VSim Reference Manual

Release 9.0.2-r2448

Tech-X Corporation

Nov 29, 2018

CONTENTS

1 Overview **1**

1.1 The Structure of the Reference Manual . . . 1

2 Visual Setup **3**

2.1 Description . . . 3
2.2 Constants . . . 3
2.3 Parameters . . . 4
2.4 Basic Settings . . . 5
2.5 Functions . . . 9
2.6 SpaceTimeFunctions . . . 10
2.7 Materials . . . 11
2.8 Geometries . . . 12
2.9 Grids . . . 16
2.10 Field Dynamics . . . 16
2.11 Particle Dynamics . . . 33
2.12 Collisions . . . 53
2.13 Histories . . . 64

3 Text Setup **71**

3.1 Global Variables . . . 71
3.2 Grid . . . 74
3.3 Decomposition . . . 79
3.4 GridBoundary . . . 81
3.5 EM Field . . . 83
3.6 ComboEmField Block . . . 93
3.7 Multifield . . . 94
3.8 ScalarDepositor and VectorDepositor . . . 186
3.9 SumRhoJ . . . 189
3.10 Fluid . . . 189
3.11 Species . . . 191
3.12 Reactions . . . 270
3.13 ImpactCollider / ImpactCollision . . . 295
3.14 Monte Carlo Interactions (DEPRECATED) . . . 303
3.15 History . . . 356
3.16 External Circuit Model . . . 397
3.17 Functions . . . 403
3.18 Slab Block . . . 450
3.19 Macros . . . 451
3.20 Postprocessing Tools . . . 590

4 Engine (Vorpal) Execution **595**
4.1 Vorpal Command Line Options 595
4.2 Customizing Environment Variables 597

5 Analyzers **599**
5.1 Guide to Analyzers 599
5.2 Available Analyzers 599

6 Trademarks and licensing **629**

Index **631**

CHAPTER

ONE

OVERVIEW

The *VSim Reference* manual describes in detail all Visual Setup parameters, all Vorpal input file blocks, and all macros that can be used.

VSim *[VSi]* is an arbitrary dimensional, electromagnetics and plasma simulation code consisting of two major components:

- VSimComposer, the graphical user interface.
- Vorpal *[NC04]*, the VSim Computational Engine.

VSim also includes many more items such as Python, MPI, data analyzers, and a set of input simplifying macros.

1.1 The Structure of the Reference Manual

The Reference Manual has three main sections, which are Visual Setup, Text Setup, and Analyzers.

The Visual Setup section will outline all available parameters in the VSim composer, with the commands themselves organized by input property. The Text Setup section follows, and is where Vorpal blocks and macros are found, as well as postprocessing tools.

The general format of each block's description is:

- Block name
- Summary of the block's purpose
- List of the block's parameters
- Example of the block as used in a Vorpal input file

We note whenever a block can or should contain another block. For the purpose of this document, a kind or a function can be considered a parameter of a block if the kind or function can be used to complete the description of the block or modify block characteristics.

Wherever possible, we describe all of the parameters in the same section as the block. While this requires that we repeat some information for different blocks that use the same parameters, you do not need to go elsewhere to find the rest of the block's description. In some cases, where a block's parameters are themselves blocks, we list the parameters' names, however we completely describe the sub-block or block that is nested in its own section.

Still within the Text Setup section, the macros will follow all the Vorpal block descriptions, and will be organized by VSim version. Postprocessing tools can also be found in Text Setup.

The final major section of *VSim Reference* is Analyzers, which includes details about all of the VSim (post engine run) analyzer scripts. Each analyzer script section will generally outline the following:

- Analysis script name

- Summary of usage with script options
- Output

Some analyzer script sections also include sections like examples, usage and testing, and other useful information.

VSim Reference is a quick-reference manual for VSim users to look up specific features and syntax for the computational engine, Vorpal. To learn about the complete VSim simulation process, including real-word physics models, please refer to *VSim Examples* and *VSim User Guide*. To learn about executing Vorpal from the command line, please refer to the *VSim User Guide*.

CHAPTER

TWO

VISUAL SETUP

2.1 Description

The *Description* element holds basic user-supplied text information about the simulation. Most of the parameters are most useful to advanced users who are creating and placing input files in the *Examples* directory of VSimComposer. The examples directory can be found in [VorpalInstallDirectory]\Contents\Examples.

`short description`: This string is the title of the example in the *New –> From Example* selection. This also sets the title that appears at the top of the VSimComposer window when the simulation is open.

`description`: This is the description given in the window on the right side in the examples window upon selecting an example.

`long description`: This text block will be visible above the image in text-based setup. It's generally used to give a description of what the simulation does, and what will happen when key parameters are modified.

`image`: The image parameter should give the name of a picture, located in the same directory as the input file, that will be given on the right-hand side of the *Editor* pane in the **Setup** tab for text-based setup. Frequently, this image is used to illustrate key parameters such as dimensions of a physical structure. 400 x 500 pixels is a good image size.

`thumbnail`: This is the small image that is visible when you select an example file, located in the same directory as the input file. 250 x 250 pixels is a good image size.

`version`: (not editable) The version of VSim used to create this example.

2.2 Constants

The *Constants* element contains a set of pre-defined physical constants that can be used in other elements of the simulation. Constants are just a number. You may add new constants by clicking the *Add* button under the *Elements Tree* while the *Constants* element (in the *Elements Tree*) is highlighted. For information on the *Elements Tree*, see the images in the VSim User Guide: Setup Window for Visual-setup Simulations page.

`kind` (not editable) The kind of constant; either a built-in kind or a User Defined kind.

`description` (only for User Defined constants) A descriptive name of the constant.

`value` The value of the constant. This is the user-supplied value of the constant if `kind = User Defined`. Otherwise it is a non-editable value supplied by VSim.

2.2.1 Built-In Constants:

- **PI**: The ratio of the circumference of a circle to its diameter.
  - 3.141592653589793
- **PIO2**: π divided by two.
  - 1.5707963267948966
- **TWOPI**: π multiplied by two.
  - 6.283185307179586
- **LIGHTSPEED**: The speed of light in vacuum (in meters/second).
  - 299792458.0
- **MU0**: The permeability of free space (in newtons/amp^2 or (tesla meters)/amp) .
  - 1.2566370614359173e-06
- **ELEMCHARGE**: The fundamental unit of charge (in coulombs).
  - 1.602176487e-19
- **ELECMASS**: The mass of an electron (in kilograms).
  - 9.10938215e-31
- **PROTMASS**: The mass of a proton (in kilograms).
  - 1.672621637e-27
- **MUONMASS**: The mass of a muon (in kilograms).
  - 1.8835313e-28
- **KB**: Boltzmann's constant in (joules/kelvin)
  - 1.3806504e-23
- **EPSILON0**: The permittivity of free space (in farads/meters or (amps^2 seconds^4)/(meters^3 kilograms)).
  - 8.854187817620389e-12
- **C2**: The speed of light squared (in meters^2/seconds^2).
  - 8.987551787368176e+16
- **ELECCHARGE**: The charge of an electron (in coulombs).
  - (-1.602176487e-19)
- **ELECMASSEV**: The mass of an electron (in eV/c^2).
  - 510998.90984764055

2.3 Parameters

The *Parameters* element is a location for evaluated, user defined variables that can be used in other elements of the simulation. A parameter is a mathematical combination of constants and other parameters. You may add a new parameter using the *Add* button under the *Elements Tree*.

`kind` (not editable): The kind of constant; a User Defined kind.

`description`: A descriptive name of the parameter.

`expression`: This is the user-supplied expression that will be calculated to determine the value of the parameter. It can include any pre-defined *Constants* as well as real numbers and some functions. Use a "^" to raise numbers to a power. Available functions include:

- **`abs(x)`**: takes the absolute value of "x".
- **`rint(x)`**: rounds "x" to an integer.
- **`sqrt(x)`**: take the square root of "x".
- **`sin(x)`**: take the sine of "x", where "x" is in radians.
- **`cos(x)`**: take the cosine of "x", where "x" is in radians.
- **`tan(x)`**: take the tangent of "x", where "x" is in radians.
- **`asin(x)`**: take the arcsine of "x", where "x" is in radians.
- **`acos(x)`**: take the arccosine of "x", where "x" is in radians.
- **`atan(x)`**: take the arctangent of "x", where "x" is in radians.
- **`sinh(x)`**: take the hyperbolic sine of "x", where "x" is in radians.
- **`cosh(x)`**: take the hyperbolic cosine of "x", where "x" is in radians.
- **`tanh(x)`**: take the hyperbolic tangent of "x", where "x" is in radians.
- **`log(x)`**: take the natural log of "x".
- **`log10(x)`**: take the base 10 log of "x".
- **`exp(x)`**: raise Euler's constant to the power "x".

`value`: The VSim calculated value of the expression.

2.4 Basic Settings

The *Basic Settings* element contains a group of property/value pairs that define the basic setup of the simulation.

`surface meshing tolerance`: Determines the relative size at which small cells from a meshed geometry surface are dropped. Set to 1.0 for simulations that do not contain any geometries.

`cfl number`: If *time step* is set to zero, the time step is automatically calculated, and for EM simulations, is reduced proportionately with the cfl number. The *cfl number* is the ratio of time step to Courant limit.

`time step`: If set to a value that is non-zero, this will be used as the simulation time step. If set to zero, the time step is calculated for you based on a number of factors.

- If it is an ES simulation without particles, the time step is set to 1.0.
- If it is an ES simulation with particles, the time step is set to the minimum of $(2/electronplasmafrequency$ or $1/(DLI * electronthermalvelocity)$ where $DLI = \sqrt{1/DX^2 + 1/DY^2 + 1/DZ^2}$ in 3 dimensions.
- If it is an EM simulation, time step is equal to the minimum of ES calculated time step or $1/LIGHTSPEED * DLI$ and multiplied by the *cfl number* and *surface meshing tolerance*.

number of steps: The number of time steps to run the simulation.

steps between dumps: The number of time steps between sequential dumps of data to hdf5 format files.

dump in groups of: If set to 1, no change. If 3, data is dumped at the period specified in steps between dumps, with an extra two dumps after each of the following 2 timesteps. For example, if steps between dumps = 20, data is written to hdf5 format files at timesteps 20,21,22,40,41,42 etc. When using this option Dump Periodicity must not be set as a Runtime Option in the Run pane.

precision: The precision of the real numbers in the simulation. For greater precision, use `double`.

- **double**
- **float**

length unit: The unit of length used for setting parameters.

- **meter**

use GPU (if found): If a GPU is found, do calculations based on faster GPU updaters.

- **false**

verbosity: The level of informative text to be output during the simulation run. The levels are listed below in an increasing order. So, the `information` level will be the least verbose, and `debug level 3` is the most verbose.

- **information**
- **emergency**
- **alert**
- **critical**
- **error**
- **warning**
- **notice**
- **debug level 1**
- **debug level 2**
- **debug level 3**

dimensionality: Set to 3, 2, or 1 to indicate how many dimensions in which to run the simulation.

- **3**
- **2**
- **1**

Note: For cylindrical coordinates, only two-dimensional electrostatic simulations are currently available in visual setup. In a 2 or 1 dimensional simulation, boundary conditions and volumes can be set in higher dimensions, but will be ignored.

grid spacing: The spacing of the grid cells.

- **uniform**

`restore geometries`: This parameter is typically set to true. Due to a bug in memory management on Windows 10 systems, it should be set to false if working with complex geometries.

`MPI decomposition`: Typically set to default, but can be specified to decompose along a single axis. Useful for simulations with significantly more cells in one direction than others.

- **`axis 0`**
- **`axis 1`**
- **`axis 2`**

`coordinate system`: The type of coordinate system to work in.

- **`cartesian`**
- **`cylindrical`**

Note: For cylindrical coordinates, only two-dimensional electrostatic simulations are currently available in visual setup.

`field solver`: The field solver determines which equations will be used to calculate the fields.

- **`electrostatic`**

  `number of guard cells`: For information about guard cells, see VSim User Guide: Simulation Concepts.

- **`electromagnetic`**

  `background permittivity`: The background permittivity of the simulation, typically 1.0

  `dielectric solver`: The type of solver to use for any dielectrics in the simulation.

  - **`point permittivity`**: Standard, sets permittivity of computational cells based on a stair step method.
  - **`permittivity averaging`**: Will calculate the average permittivity in a cell that has two dielectric objects.

  `Cerenkov Filter`: Electromagnetic problems allow for the selection of a numerical Cerenkov noise filter. These filters come in 4 varieties, or no filter.

  - **`none`**: No Filter
  - **`weak`**: Filters with a formula of $(1-x)*(1+x)$. Should be used in problems with cavities or resonant structures.
  - **`medium`**: Filters with a formula of $(1-x)*(1+2x)(1-x)$. Should be used in problems with cavities or resonant structures.
  - **`strong`**: Filters with a formula of $(1-x)$. Should be used in problems with high numbers of relativistic particles.
  - **`extreme`**: Filters with a formula of $(1-x)*(1-x)$. Should be used in problems with high numbers of relativistic particles.

  Strong filters will execute the fastest, while medium will execute the slowest. These are all **Godfrey Filters**, which use additional curl-curl operations on the electric field to update the field at the next time step, removing short wavelengths. **Friedman Filters** are another type of filter that remove high frequency noise rather than short wavelengths. Please contact Tech-X if you wish to implement this class of filter.

- **`prescribed fields`**: The prescribed fields solver is used with a defined electric and magnetic field that is allowed to have some time dependence. It is most commonly used for multipacting simulations where it can provide very high resolution of the electric field near PEC objects.
  - **`number of guard cells`**: For information about guard cells, see VSim User Guide: Simulation Concepts.
- **`no field solver`**: For pure particle movement simulations with no fields.

`periodic directions`: The directions of the simulation which should be modeled as periodic, if any. Phase shifting boundaries are used for structures which do not have a full period in the simulation space. Phase shifting boundaries cannot be used in particle simulations.

- **`no periodicity`**
- **`periodic x`**
- **`periodic y`**
- **`periodic z`**
- **`periodic x and y`**
- **`periodic x and z`**
- **`periodic y and z`**
- **`periodic x, y, and z`**
- **`phase shifting periodic x`**

  `phase shift x`: Fields will be phase shifted in the x direction by this many radians.
- **`phase shifting periodic y`**

  `phase shift y`: Fields will be phase shifted in the y direction by this many radians.
- **`phase shifting periodic z`**

  `phase shift z`: Fields will be phase shifted in the z direction by this many radians.
- **`phase shifting periodic x and y`**

  `phase shift x`: Fields will be phase shifted in the x direction by this many radians.

  `phase shift y`: Fields will be phase shifted in the y direction by this many radians.
- **`phase shifting periodic x and z`**

  `phase shift x`: Fields will be phase shifted in the x direction by this many radians.

  `phase shift z`: Fields will be phase shifted in the z direction by this many radians.
- **`phase shifting periodic y and z`**

  `phase shift y`: Fields will be phase shifted in the y direction by this many radians.

  `phase shift z`: Fields will be phase shifted in the z direction by this many radians.
- **`phase shifting periodic x, y, and z`**

  `phase shift x`: Fields will be phase shifted in the x direction by this many radians.

  `phase shift y`: Fields will be phase shifted in the y direction by this many radians.

  `phase shift z`: Fields will be phase shifted in the z direction by this many radians.

`particles`: Whether or not to include particles in the simulation.

- **`no particles`**

- **`include particles`**: If particles are included in the simulation, the following two properties are used to help calculate the time step.

    `estimated max electron density`: an estimate of the maximum electron density for setting a default timestep

    `estimated min electron temperature (eV)`: an estimate of the minimum electron temperature for setting a default timestep

    `dump nodal fields`: If True, the fields used to calculate particle pushes will be written to memory. If false, they will not. This can save hard disk space in large simulations.

`collisions framework`: Collisions framework to be used in the simulations.

- **`no collisions`**
- **`reduced`**: Reduced collisions framework. Also called the "Impact Collider" framework.
- **`monte carlo`**: Legacy Monte-Carlo interactions from previous VSim versions.
- **`reactions`**: Full featured particle and fluid interactions most commonly used.

    `collision order`: Either *random*, *constant*, or *rotate*. A random order will perform each collision in a random order, constant in the order specified, and rotate will move down the list of collisions, performing a new one first, each time. As each particle can only be involved in one collision per timestep this can effect simulation results.

`moving window`: Whether or not to use a moving window, which allows the simulation window to move at the speed of light in the chosen direction. Useful for simulations such as laser pulse or particle beam moving at a velocity close to the speed of light. Can only set a moving window in an electromagnetic simulation without phase shifting boundary conditions.

- **`no moving window`**
- **`with moving window`**

    `shift position fraction`: This determines the time at which the window will begin to move. The window will begin to move at a time equal to the *shift position fraction* times the size of the simulation grid divided by the speed of light.

    `shift speed fraction`: This is the relativistic β factor which determines the speed v at which the window will move, where $v = \beta c$.

2.5 Functions

The *Functions* element is a location for writing user defined functions that can be used in simplifying the definition of a *SpaceTimeFunction*. When used in writing a *SpaceTimeFunction*, the arguments of the function can be set to user defined constants and parameters. See the A6 Magnetron Modes Example for an example of the use of a *Function*. You have options to create your own *User Defined* function or use the built-in *turn on* function. The *turn on* function has the mathematical formula $H(t) * 0.5 * (1 - \cos(\pi t/T)) * H(T - t)$ where $H(t)$ is the Heaviside Step Function.

`kind` (not editable): The kind of constant; a User Defined kind.

`description`: A string describing the function.

`arguments`: The arguments that are used in the expression. The function can contain any number of arbitrary arguments and is not limited to the default values of x,y.

`expression`: This is the user-supplied expression that is a function of the arguments given in the argument property. It can include any pre-defined *Constants*, *Parameters*, or *Functions*, as well as real numbers and Python operators.

2.6 SpaceTimeFunctions

The *SpaceTimeFunctions* element is a location for writing user defined functions that specifically depend on the spatial and temporal variables x, y, z, and t. A space time function can use *Parameters* and *Constants* by just typing them directly in as a value of the property.

- **`User Defined`**. **This option is deprecated. Use expression instead.**
- **`expression`** This is the user-supplied expression that is a function of x, y, z, or t. It can include any pre-defined *Constants*, *Parameters*, or *Functions*, as well as real numbers.
- **`python`** This space time function will allow access to a function defined in a Python file to be used in place of a user-defined function.

  `name` This is the name of the Python function to be accessed. The Python file must be in the same directory as the runspace.
- **`feedback`** This space time function is used to take the value from a history and use that value in the next timestep, allowing feedback.

  `expression` The initial value to be used in the feedback loop. This expression will be multiplied from the value of the history in the previous output.

  `history` The name of the history from which to take values; pseudo-potential and absorbed particle current histories are supported.

  `history goal` The value of the history that should be obtained.

  `time constant` Defines how quickly the feedback responds to a difference in the measured and desired value. If too small, the measured value will oscillate near the desired value, if too large it will take a long time to reach the desired value.
- **`chirpWavePulse`** Produces a plane wave modulated by a pulse envelope. For more information, see *chirpWavePulse*.
- **`cosineFlattop`** Flat top function. See *cosineFlattop*.
- **`cosineRamp`** Function for an initial ramp. See *cosineRamp*.
- **`gaussian`** Produces a Gaussian function. See *gaussian*.
- **`gaussianPulse`** Creates a sinusoidal pulse in the form of a Gaussian beam, modulated by a Gaussian envelope longitudinally. See *gaussianPulse*.
- **`halfSinePulse`** Function for a sinusoidal pulse in the form of a Gaussian beam, modulated by a longitudinal half-sine function. See *halfSinePulse*.
- **`leakychannel`** Function that is parabolic in radius, then drops linearly to zero. See *leakychannel*.
- **`planeWavePulse`** Creates a plane wave that's modulated by a Gaussian transversely and by a half-sine function longitudinally. See *planeWavePulse*.
- **`radialCosChannel`** Function for an initial ramp into a region of a channel. See *radialCosChannel*.
- **`sinePlaneWave`** Generates a plane wave pulse that is based on a sine wave. See *sinePlaneWave*.
- **`sum function`** This is the sum of two previously defined space time functions. The functions used *must* be defined before their use in the sum function. Sum functions may be nested. For example, you could have a second sum function that accepts a previously defined sum function in order to sum three or more space time functions.

  `sumFunction1` The first function to be summed.

  `sumFunction2` The second function to be summed.

- **`product function`** This is the product of two previously defined space time functions. The functions used *must* be defined before their use in the product function. Product functions may be nested, for example, a second product function can be used to accept a previously defined product function in order to multiply three or more space time functions together.

  `prodFunction1` The first function to be summed.

  `prodFunction2` The second function to be summed.

2.7 Materials

The *Materials* element stores information about any materials used in the simulation. To use one of VSim's pre-defined materials, highlight the *Materials* element and then switch from *3D View* to *Database* in the Geometry View (see VSim User Guide: Setup Window for Visual-setup Simulations for a picture of the Geometry View). To add one of the pre-defined materials from the table, highlight the material then press *Add To Simulation* button in upper right hand corner of the VSim Composer window. The material will now be listed under the *Materials* element. To access a wider selection of materials you may load the *emthermal.vmat* file by right-clicking on the *Materials* element and selecting *Import Materials*. You may also import your own customized material (see *Customizing Materials* below).

The editable properties of the materials are:

`kind` **(not editable)** The kind of material (eg dielectric, conductor, particle absorber, permeable, etc), as defined in the .vmat file.

`heat capacity` The heat capacity of the material.

`thermal conductivity` The thermal conductivity of the material.

`resistance` The resistance of the material.

`conductivity` The conductivity of the material.

`relative permittivity` The relative permittivity of the material.

2.7.1 Customizing Materials

Custom materials properties can be created in a text editor and imported into VSim. To import a custom material:

1. Go to the "data" folder at the top level of the VSim installation directory. On Windows this will be in "Program Files\Tech-X (Win64)\VSim-9.0\data". On Mac and Linux, there is no standard location.

2. Create a new text file that ends in the .vmat extension. For example: *emthermalcustom.vmat*.

3. Open the default materials file *emthermal.vmat*.

4. Copy a block from *emthermal.vmat* to *emthermalcustom.vmat* to use as a sample.

5. Edit the title and properties of the block as needed. For example:

```
<Material FreshWater>
  <strings>
  kind = "dielectric"
  </strings>
  <params>
  heat capacity = 4.184
  conductivity = 0.005
  relative permittivity = 80.4
  thermal conductivity = 0.6065
```

(continues on next page)

(continued from previous page)

```
  </params>
</Material>
```

6. Save the new file.

7. Back in VSim, click on the *Materials* element in your simulation and then click *Add –> Import Materials*. A file browser will appear with the *data* directory already open.

8. Select your custom vmat file. If you do not see your vmat file in the directory, navigate to where you saved it.

9. The view will change to the *Database* tab and the materials added will be available here.

2.8 Geometries

The *Geometries* element contains information about any geometries that are in the simulation. You can import a file, or create your own with CSG using the primitive shapes described below.

Be sure to assign a *Material* to a geometry object before it will appear elsewhere in the simulation (i.e. as an option for a boundary condition, particle emitter, particle absorber, etc.).

2.8.1 CSG

Constructive Solid Geometry can be used to build your own complex geometry.

`kind` (not editable) Construction Group

`tessellation` The size of triangles used to construct the shape. This is a unitless number that is the maximum deviation of a facet from the curved surface divided by the diameter of the facet.

`Sphere`

- **`kind` (not editable)** Sphere
- **`material`** The material to use for the shape. Chosen from a list of imported materials in your simulation.
- **`radius`** The radius of the sphere.
- **`x position`** The location of the center of the sphere in the x direction.
- **`y position`** The location of the center of the sphere in the y direction.
- **`z position`** The location of the center of the sphere in the z direction.

`Box`

- **`kind` (not editable)** Box
- **`material`** The material to use for the shape. Chosen from a list of imported materials in your simulation.
- **`length`** The length of the box.
- **`height`** The height of the box.
- **`width`** The width of the box.
- **`x position`** The location of the center of the box base in the x direction.
- **`y position`** The location of the center of the box base in the y direction.

z position The location of the center of the box base in the z direction.

width direction x Set to 1 to make the width parameter of the box correspond to the x direction.

width direction y Set to 1 to make the width parameter of the box correspond to the y direction.

width direction z Set to 1 to make the width parameter of the box correspond to the z direction.

angle The angle of the box.

Cylinder

kind (not editable) Cylinder

material The material to use for the shape. Chosen from a list of imported materials in your simulation.

length The length of the cylinder.

radius The radius of the cylinder.

x position The location of the base of the cylinder in the x direction.

y position The location of the base of the cylinder in the y direction.

z position The location of the base of the cylinder in the z direction.

axis direction x Set to 1 to make the axial direction of the cylinder x; if set to -1, the length parameter will extend in the negative direction.

axis direction y Set to 1 to make the axial direction of the cylinder y; if set to -1, the length parameter will extend in the negative direction.

axis direction z Set to 1 to make the axial direction of the cylinder z; if set to -1, the length parameter will extend in the negative direction.

Cone

kind (not editable) Cone

material The material to use for the shape. Chosen from a list of imported materials in your simulation.

height The height of the cone.

radius The radius of the base of the cone.

x position The location of the center of the cone base in the x direction.

y position The location of the center of the cone base in the y direction.

z position The location of the center of the cone base in the z direction.

axis direction x Set to 1 to make the axial direction of the cone x.

axis direction y Set to 1 to make the axial direction of the cone y.

axis direction z Set to 1 to make the axial direction of the cone z.

Torus

kind (not editable) Torus

material The material to use for the shape. Chosen from a list of imported materials in your simulation.

`major radius` The radius to the center of the torus.

`minor radius` The radius from the center of the torus to the outside of the torus.

`x position` The location of the center of the torus in the x direction.

`y position` The location of the center of the torus in the y direction.

`z position` The location of the center of the torus in the z direction.

`axis direction x` Set to 1 to make the axial direction of the torus x.

`axis direction y` Set to 1 to make the axial direction of the torus y.

`axis direction z` Set to 1 to make the axial direction of the torus z.

`Pipe`

`kind` (not editable) Pipe

`material` The material to use for the shape. Chosen from a list of imported materials in your simulation.

`length` The length of the pipe.

`inner radius` The inner radius of the pipe.

`outer radius` The outer radius of the pipe.

`x position` The location of the base of the pipe in the x direction.

`y position` The location of the base of the pipe in the y direction.

`z position` The location of the base of the pipe in the z direction.

`axis direction x` Set to 1 to make the axial direction of the pipe x; if set to -1, the length parameter will extend in the negative direction.

`axis direction y` Set to 1 to make the axial direction of the pipe y; if set to -1, the length parameter will extend in the negative direction.

`axis direction z` Set to 1 to make the axial direction of the pipe z; if set to -1, the length parameter will extend in the negative direction.

`TruncCone`

`kind` (not editable) TruncCone (Truncated Cone)

`material` The material to use for the shape. Chosen from a list of imported materials in your simulation.

`height` The height of the cone.

`radius1` The radius of the base of the cone.

`radius2` The radius of the top of the cone.

`x position` The location of the center of the cone base in the x direction.

`y position` The location of the center of the cone base in the y direction.

`z position` The location of the center of the cone base in the z direction.

`axis direction x` Set to 1 to make the axial direction of the cone x.

`axis direction y` Set to 1 to make the axial direction of the cone y.

`axis direction z` Set to 1 to make the axial direction of the cone z.

`Wedge`

kind (not editable) Wedge

material The material to use for the shape. Chosen from a list of imported materials in your simulation.

length1 One length of the wedge.

length2 The second length of the wedge.

height The height of the wedge.

width The width of the wedge. Extrudes the wedge from a two dimensional to three dimensional object.

x position The location of the base of the wedge in the x direction.

y position The location of the base of the wedge in the y direction.

z position The location of the base of the wedge in the z direction.

width direction x Set to 1 to apply the width parameter in the x direction.

width direction y Set to 1 to apply the width parameter in the y direction.

width direction z Set to 1 to apply the width parameter in the z direction.

2.8.2 Boolean Operations

CSG Primitives may be combined in three different ways:

Subtract This will subtract the second selected primitive from the first selected primitive. Denoted by $-$.

Union This will combine the two primitives into a single object. For use if the combined object is set to be a particle sink. Denoted by $\cup$.

Intersect This will leave only the volume of the two primitives that intersect as an object. Denoted by $\cap$.

Boolean operations may be nested, for example two primitives may be combined in a union, and then with a third primitive in a second union. To combine primitive shapes, you must first add two or more shapes to your simulation. Once your primitive shapes are added, highlight the two shapes you wish to combine and right-click and select *Boolean Operation* then the operation you wish to perform.

2.8.3 Imported

You can import a geometry of type .stl, .ply, .vtk, .stp, .step, or .p12 by right-clicking the *Geometries* element and selecting *Import Geometries*.

kind (not editable)

- TriangSolid
- OceStepFromFile

filename The name and location of the imported file.

scale A factor to scale the imported geometry by. (Not available in 8.0.)

tessellation The size of triangles used to construct the shape. This is a unitless number that is the maximum deviation of a facet from the curved surface divided by the diameter of the facet.

2.9 Grids

`kind` (not editable) The kind of grid used

`xMin` The domain extent in the negative x-direction.

`xMax` The domain extent in the positive x-direction.

`yMin` The domain extent in the negative y-direction.

`yMax` The domain extent in the positive y-direction.

`zMin` The domain extent in the negative z-direction.

`zMax` The domain extent in the positive z-direction.

`xCells` The number of cells in the x-direction.

`yCells` The number of cells in the y-direction.

`zCells` The number of cells in the z-direction.

`planeShift` (in 1D or 2D) When working in one or two dimensions, this selects the xy plane in which your simulation is defined.

`lineShift` (in 1D) When working in one dimension, this selects the line (parallel to the x-axis) along which your simulation is defined.

2.10 Field Dynamics

2.10.1 FieldDynamics for Electromagnetic Simulations

This page describes the Field options available for electromagnetic simulations, that is, when the `field solver` in the *Basic Settings* element is set to `electromagnetic`.

Fields

`Electric Field` The Electric Field

`Magnetic Field` The magnetic field.

`Current Density` The current density field. Must be added by user by right clicking "Fields", hovering over "Add Field", then choosing "Current Density". Shows up as "J0"

`External Field` To allow external fields to either be added to the electric, magnetic or current fields. An external field will be added after the field solve and effect particle movements in the simulation. Must be added by user by right clicking "Fields", hovering over "Add Field", then choosing "External Field"

`description` A space to provide a descriptive comment for the field.

`field type` The type of field, electric, current or magnetic.

`field specification` Either import h5 file or function defined.

- **`import h5 file`** A vis schema compliant h5 file. It does require that the file be in the same directory as the simulation. An error message will be provided if the file fails to import.

  `filename:` The name of the .hdf5 file to be imported. Typical convention is simulation-Name_fieldName_dumpNum.h5

`lower bound 0:` The cell index of the 0th component lower bound of the source field, to be imported to the matching index in the simulation. If left not applicable the entire field will be imported.

`lower bound 1:` The cell index of the 1st component lower bound of the source field, to be imported to the matching index in the simulation. If left not applicable the entire field will be imported.

`lower bound 2:` The cell index of the 2nd component lower bound of the source field, to be imported to the matching index in the simulation. If left not applicable the entire field will be imported.

`upper bound 0:` The cell index of the 0th component upper bound of the source field, to be imported to the matching index in the simulation. If left not applicable the entire field will be imported.

`upper bound 1:` The cell index of the 1st component upper bound of the source field, to be imported to the matching index in the simulation. If left not applicable the entire field will be imported.

`upper bound 2:` The cell index of the 2nd component upper bound of the source field, to be imported to the matching index in the simulation. If left not applicable the entire field will be imported.

- **`function defined`** Allows for manual specification of each component of the field

  `component 0:` The function defining the field in the 0th component. Can be a time varying function

  `component 1:` The function defining the field in the 1st component. Can be a time varying function

  `component 2:` The function defining the field in the 2nd component. Can be a time varying function

  `time dependent:` Set to true if any of the functions are time varying. The function will then be recalculated at each time step.

`nodalE` This is a node centered electric field, used for calculating particle movements. It cannot be added to a simulation but is created automatically and will be plot-able post run if *dump nodal fields* = true.

`nodalB` This is a node centered magnetic field, used for calculating particle movements. It cannot be added to a simulation but is created automatically and will be plot-able post run if *dump nodal fields* = true.

`invEps` This is a field that stores the inverse values of dielectrics in a simulation. It cannot be added in the *Fields* tab but is created automatically if necessary and can be visualized.

`D` This is the displacement field, only created if a dielectric is present in the simulation. It cannot be added in the *Fields* tab but is created automatically if necessary and can be visualized.

Initial Condition

To add an Initial Condition to a field, right-click on the field and select *Add FieldInitialCondition –> Initial Condition*.

`kind` (not editable) Initial Condition

`expression` The value of the initial condition. Can be assigned a *Constant*, *Parameter*, or *SpaceTime-Function* by right-clicking.

`component` Can be 0, 1 or 2 for the first, second, or third component of the field.

Field Boundary Conditions

To add a Boundary Condition, right-click on *FieldBoundaryConditions* and select your choice from *Add FieldBoundaryCondition*. Your choices for dimensionality and field solver in the vsimcomposer-basic-settings element will determine which Boundary Conditions are available to add to your simulation.

`Boundary Launcher` A boundary launcher will set the chosen `field` to the value given in the `applied field` functions of space and time.

> **`field`** The `field` to which the boundary condition applies.
>
> **`applied fields`** The location and orientation of the applied fields. The location is chosen from the simulation domain boundaries. Depending on your choice for the applied field orientation (i.e., the Value of the applied fields Property), you can set 2 of the following fields as a function of space and time.
>
> - **`Fx(x,y,z,t)`**
> - **`Fy(x,y,z,t)`**
> - **`Fz(x,y,z,t)`**

`Coaxial Waveguide`

> This is a port launcher boundary condition, with the functions defining it preset to create a coaxial cable. See the VSimEM - Antennas example, "Coaxial Loop Antenna", for a demonstration of its use. For proper operation, a physical coaxial cable must be constructed in *Geometries* to match the specified cable here.
>
> **`inner radius`** The radius of the inner conductor.
>
> **`outer radius`** The radius of the outer conductor.
>
> **`frequency`** The frequency of the signal.
>
> **`voltage`** The voltage of the signal.
>
> **`relative permittivity`** The relative permittivity of the dielectric insulator.
>
> **`start time`** The time at which to turn on the coaxial waveguide.
>
> **`stop time`** The time at which to turn off the coaxial waveguide.
>
> **`turn on time`** The amount of time to bring the coaxial waveguide up to full power. Typically, 2.5 periods of the carried signal.
>
> **`coaxial waveguide surface`** The simulation domain boundary from which the coaxial waveguide enters the simulation. Depending on the selected boundary, two of the following three options are allowed.
>
> - **`X-center coordinate`** The center of the coaxial waveguide in X.
> - **`Y-center coordinate`** The center of the coaxial waveguide in Y.
> - **`Z-center coordinate`** The center of the coaxial waveguide in Z.

`Matched Absorbing Layer` A boundary condition that adds a Matched Absorbing Layer (MAL) to the specified face. A matched absorbing layer is an adiabatic absorber that uses isotropic electric and magnetic damping profiles to absorb the incident wave. This is unlike a PML (Perfectly Matched Layer), which uses the same electric and magnetic damping profiles, but is anisotropic. MAL boundaries are more stable, as an anisotropic boundary condition can become unstable when the incident wave has a non-zero imaginary part to its normal wavenumber (e.g., fringing fields from nearby structure, or particles entering the layer).

> **`thickness`** The thickness of the MAL in meters. This value must be greater than the length of a computational cell in the direction of the boundary condition.

`surface` The simulation domain surface on which the MAL boundary condition should be set.

- **`lower x`**
- **`lower y`**
- **`lower z`**
- **`upper x`**
- **`upper y`**
- **`upper z`**

`Open` A boundary condition that is "open" allowing EM waves to freely exit. This is a *[Mur98]* absorbing boundary condition. The open boundary condition works best for waves normal to the surface.

`surface` The simulation domain surface on which the open boundary condition should be set.

- **`lower x`**
- **`lower y`**
- **`lower z`**
- **`upper x`**
- **`upper y`**
- **`upper z`**

`Perfect Electric Conductor` A boundary condition that sets parallel components of the electric field to zero. For example, if the PEC boundary condition is added to the lower x surface, the y and z components of the electric field are set to zero.

`surface` The simulation domain surface on which the PEC boundary condition should be set.

- **`lower x`**
- **`lower y`**
- **`lower z`**
- **`upper x`**
- **`upper y`**
- **`upper z`**

`Perfect Magnetic Conductor` A boundary condition that sets parallel components of the magnetic field to zero. For example, if the PMC boundary condition is added to the lower x surface, the y and z components of the magnetic field are set to zero.

`surface` The simulation domain surface on which the PEC boundary condition should be set.

- **`lower x`**
- **`lower y`**
- **`lower z`**
- **`upper x`**
- **`upper y`**
- **`upper z`**

`Perfectly Matched Layer` A perfectly matched layer (PML) boundary condition. PMLs provide boundary conditions for the Yee algorithm that allow outgoing waves to leave without reflections (ideally). However in practice, there are problems with reflections in some materials, particularly photonic crystals. It is recommended to use the *Matched Absorbing Layer* (MAL) instead *[OZAJ08]*. PMLs can also fail when combined with other active boundary conditions, like ports, when there are particles present, or when structures exist at the PML boundary which are not normal to the boundary. For additional options within Text Setup, see *PmlRegion* in VSim Reference.

`thickness` The thickness of the PML in meters. This must correspond to a value greater than the length of one computational cell in the direction of the boundary condition.

`sigma` The strength of the PML conductivity. Typically 3.0 or $5.0 * LIGHTSPEED/DL$, where DL is the cell size in the normal direction.

`exponent` The exponent in the PML conductivity. Typically 1.5.

`surface` The simulation domain surface on which the PML boundary condition should be set.

- **`lower x`**
- **`lower y`**
- **`lower z`**
- **`upper x`**
- **`upper y`**
- **`upper z`**

`Port` A Port boundary condition is a tuned phase-velocity boundary condition. It can be used as an open or outgoing boundary condition, where waves traveling at exactly the specified phase velocity will exit the simulation with no reflection at all. Waves traveling at other phase velocities will partially exit and partially reflect, with a power reflection coefficient of $\rho = (v_{p,wave} - v_{p,bc})^2/(v_{p,wave} + v_{p,bc})^2$.

`phase velocity` The phase velocity tuning parameter in meters/second.

`surface` The simulation domain surface on which the MAL boundary condition should be set.

- **`lower x`**
- **`lower y`**
- **`lower z`**
- **`upper x`**
- **`upper y`**
- **`upper z`**

`Port Launcher`

A Port Launcher boundary condition will add a `Port` boundary condition to the chosen surface as well as setting the D field to the value given in the `applied field` functions of space and time.

`phase velocity` The phase velocity tuning parameter in meters/second.

`applied fields` The location and orientation of the applied fields. The location is chosen from the simulation domain boundaries. Depending on your choice for the applied field orientation, you can set two of the following fields as a function of space and time.

- **`Dx(x,y,z,t)`**
- **`Dy(x,y,z,t)`**
- **`Dz(x,y,z,t)`**

`Rectangular Waveguide` This is a port launcher boundary condition, with the functions defining it preset to create a rectangular waveguide. See the VSimEM example Rectangular Waveguide for an demonstration of its use. For proper operation, a physical waveguide must be constructed in *Geometries* at the location specified here.

`frequency` The frequency of the signal.

`voltage` The voltage of the signal.

`relative permittivity` The relative permittivity of the dielectric insulator.

`start time` The time at which to turn on the rectangular waveguide.

`stop time` The time at which to turn off the rectangular waveguide.

`turn on time` The amount of time to bring the rectangular waveguide up to full power. If this time is set to less than 2.5 periods of the carried signal, it will automatically be increased to 2.5 periods.

`waveguide surface` The simulation domain boundary from which the rectangular waveguide enters the simulation. Depending on the selected boundary, two of the following three options are allowed, as well as the polarization direction.

- **`X-center coordinate`** The center of the rectangular waveguide in X
- **`Y-center coordinate`** The center of the rectangular waveguide in Y
- **`Z-center coordinate`** The center of the rectangular waveguide in Z
- **`polarization direction`** The polarization direction of the waveguide. This will correspond to the *height* parameter of the waveguide.

`waveguide type` The type of waveguide used. Several commonly used waveguides are included, as well as the option to define your own.

- **`User Defined`**
  - **`height`** The height of the waveguide. This will correspond to the polarization direction of the waveguide
  - **`width`** The width of the waveguide.
- **`Included Waveguides`** The eight predefined waveguides are WR-90, WR-340, WR-284, WR-229, WR-187, WR-159, WR-137, and WR-112.
  - **`WG equivalent` (not-editable)** The RCSC designation of this waveguide type.
  - **`standard frequency range` (not-editable)** The frequencies over which this waveguide operates. A warning will be given if an operating frequency is given outside this range.
  - **`height` (not-editable)** The height of the waveguide. This will correspond to the polarization direction of the waveguide.
  - **`width` (not-editable)** The width of the waveguide.

Current Distributions

`Dipole Current` A dipole current is a current source centered around the user specified location and going ± 1 cell around the specified location.

`kind` (not editable) Dipole Current

`description` A space to provide a descriptive comment for the current.

`expression` The expression used to define the dipole. Can be assigned a *Constant*, *Parameter*, or *Space-TimeFunction* by right-clicking. This should be either a constant or a function of time only.

`component` The component of the current density field to be set by the expression.

`coordinate` The physical location (in meters) where the dipole current should be set.

`Distributed Current` A distributed current sets the values of the current density in the specified volume.

`kind` (not editable) Distributed current.

`description` A space to provide a descriptive comment for the current.

`J0(x,y,z,t)` If any, the expression for the current source in the x-direction.

`J1(x,y,z,t)` If any, the expression for the current source in the y-direction.

`J2(x,y,z,t)` If any, the expression for the current source in the z-direction.

`volume`

`xMin` The domain extent in the negative x-direction.

`xMax` The domain extent in the positive x-direction.

`yMin` The domain extent in the negative y-direction.

`yMax` The domain extent in the positive y-direction.

`zMin` The domain extent in the negative z-direction.

`zMax` The domain extent in the positive z-direction.

RCSBox Properties

The RCS box is available for electromagnetic simulations. It can be used to define a box and wave for calculation of radar cross sections. The wave defined will be perfectly absorbed at the other edge of the box, while waves from scattering off an object inside of the box will be allowed to exit. This allows for a relatively easy calculation of the radar cross section using the *Far-Field Box Data* History.

`kind` Only “Radar Cross Section” is available and is not editable.

`description` A space to provide a descriptive comment for the RCSBox.

`amplitude` Amplitude of the incident wave.

`frequency` Frequency of the incident wave.

`x component incident direction` Incident angle in the x direction in radians.

`y component incident direction` Incident angle in the y direction in radians.

`z component incident direction` Incident angle in the z direction in radians.

`polarization direction x real` Polarization of the real component of the wave in the x direction.

`polarization direction y real` Polarization of the real component of the wave in the y direction.

`polarization direction z real` Polarization of the real component of the wave in the z direction.

`polarization direction x imaginary` Polarization of the imaginary component of the wave in the x direction.

`polarization direction y imaginary` Polarization of the imaginary component of the wave in the y direction.

`polarization direction z imaginary` Polarization of the imaginary component of the wave in the z direction.

`volume`

xMin The domain extent in the negative x-direction.

xMax The domain extent in the positive x-direction.

yMin The domain extent in the negative y-direction.

yMax The domain extent in the positive y-direction.

zMin The domain extent in the negative z-direction.

zMax The domain extent in the positive z-direction.

2.10.2 FieldDynamics for Electrostatic Simulations

This page describes the Field options available for electrostatic simulations, that is, when the `field solver` in the *Basic Settings* element is set to `electrostatic`.

Fields

Phi The electric potential field. This field is un-editable, and is calculated automatically.

Charge Density The charge density field. This field is un-editable, and is calculated automatically.

Electric Field The Electric Field. This field is un-editable, and is calculated automatically.

Background Charge Density The background charge density field. This field can be added to a simulation by by right clicking "Fields", hovering over "Add Field", then clicking "Background Charge Density". You may add many Background Charge Densities to your simulation.

- **expression** The value of the background charge density field. Can be a constant, a parameter, or a function of space and time.

External Field An External Field can be added to a simulation by by right clicking "Fields", hovering over "Add Field", then clicking "External Field". An external field will be added after the field solve and effect particle movements in the simulation.

description A space to provide a descriptive comment for the field.

field type In electrostatic simulations only magnetic fields may be added.

field specification Either import h5 file or function defined.

- **import h5 file** A vis schema compliant h5 file. It does require that the file be in the same directory as the simulation. An error message will be provided if the file fails to import.

  filename: The name of the .hdf5 file to be imported. Typical convention is simulationName_fieldName_dumpNum.h5

  lower bound 0: The cell index of the 0th component lower bound of the source field, to be imported to the matching index in the simulation. If left not applicable the entire field will be imported.

  lower bound 1: The cell index of the 1st component lower bound of the source field, to be imported to the matching index in the simulation. If left not applicable the entire field will be imported.

  lower bound 2: The cell index of the 2nd component lower bound of the source field, to be imported to the matching index in the simulation. If left not applicable the entire field will be imported.

upper bound 0: The cell index of the 0th component upper bound of the source field, to be imported to the matching index in the simulation. If left not applicable the entire field will be imported.

upper bound 1: The cell index of the 1st component upper bound of the source field, to be imported to the matching index in the simulation. If left not applicable the entire field will be imported.

upper bound 2: The cell index of the 2nd component upper bound of the source field, to be imported to the matching index in the simulation. If left not applicable the entire field will be imported.

- **function defined** Allows for manual specification of each component of the field

  component 0: The function defining the field in the 0th component. Can be a time varying function

  component 1: The function defining the field in the 1st component. Can be a time varying function

  component 2: The function defining the field in the 2nd component. Can be a time varying function

  time dependent: Set to true if any of the functions are time varying. The function will then be recalculated at each time step.

nodalE This is a node centered electric field, used for calculating particle movements. It cannot be added to a simulation but is created automatically and will be visible if *dump nodal fields* = true.

nodalB This is a node centered magnetic field, used for calculating particle movements. It cannot be added to a simulation but is created automatically and will be visible if *dump nodal fields* = true.

invEps This is a field that stores the inverse values of dielectrics in a simulation. It cannot be added in the *Fields* tab but is created automatically if necessary and can be visualized.

D This is the displacement field, only created if a dielectric is present in the simulation. It cannot be added in the *Fields* tab but is created automatically if necessary and can be visualized.

Initial Condition

To add an Initial Condition to a field, right-click on the field and select *Add FieldInitialCondition –> Initial Condition*.

kind (not editable) Initial Condition

expression The value of the initial condition. Can be assigned a *Constant*, *Parameter*, or *SpaceTimeFunction* by right-clicking.

component Can be 0, 1 or 2 for the first, second, or third component of the field.

Field Boundary Conditions

To add a Boundary Condition, right-click on *FieldBoundaryConditions* and select your choice from *Add FieldBoundaryCondition*. Your choices for dimensionality and field solver in the vsimcomposer-basic-settings element will determine which Boundary Conditions are available to add to your simulation.

Dirichlet Use a Dirichlet boundary condition to set the value of the potential field, Phi, on the surface.

value The value of the boundary condition. Can be assigned a *Constant*, *Parameter*, or *SpaceTimeFunction* by right-clicking.

surface The surface on which the Dirichlet boundary condition should be set.

- **`lower x`** The lower x boundary of the simulation domain.
- **`lower y`** The lower y boundary of the simulation domain.
- **`lower z`** The lower z boundary of the simulation domain.
- **`upper x`** The upper x boundary of the simulation domain.
- **`upper y`** The upper y boundary of the simulation domain.
- **`upper z`** The upper z boundary of the simulation domain.
- **`partial lower x`** A Dirichlet boundary condition applied only to part of the lower x simulation boundary. The entire lower x boundary must be filled with a boundary condition. The lower coordinate is included in the simulation and the upper bound is exclusive. Also note that at the intersection of the Y or Z simulation boundary the first computational cell of the X boundary is included in the Y or Z simulation boundary condition.

  `lower y coordinate` The inclusive lower y coordinate.

  `upper y coordinate` The exclusive upper y coordinate.

  `lower z coordinate` The inclusive lower z coordinate.

  `upper z coordinate` The exclusive upper z coordinate.
- **`partial lower y`** A Dirichlet boundary condition applied only to part of the lower y simulation boundary. The entire lower y boundary must be filled with a boundary condition. The lower coordinate is included in the simulation and the upper bound is exclusive. Also note that at the intersection of the X or Z simulation boundary the first computational cell of the Y boundary is included in the X or Z simulation boundary condition.

  `lower x coordinate` The inclusive lower x coordinate.

  `upper x coordinate` The exclusive upper x coordinate.

  `lower z coordinate` The inclusive lower z coordinate.

  `upper z coordinate` The exclusive upper z coordinate.
- **`partial lower z`** A Dirichlet boundary condition applied only to part of the lower z simulation boundary. The entire lower z boundary must be filled with a boundary condition. The lower coordinate is included in the simulation and the upper bound is exclusive. Also note that at the intersection of the X or Y simulation boundary the first computational cell of the Z boundary is included in the X or Y simulation boundary condition.

  `lower x coordinate` The inclusive lower x coordinate.

  `upper x coordinate` The exclusive upper x coordinate.

  `lower y coordinate` The inclusive lower y coordinate.

  `upper y coordinate` The exclusive upper y coordinate.
- **`partial upper x`** A Dirichlet boundary condition applied only to part of the upper x simulation boundary. The entire upper x boundary must be filled with a boundary condition. The lower coordinate is included in the simulation and the upper bound is exclusive. Also note that at the intersection of the Y or Z simulation boundary the first computational cell of the X boundary is included in the Y or Z simulation boundary condition.

  `lower y coordinate` The inclusive lower y coordinate.

  `upper y coordinate` The exclusive upper y coordinate.

  `lower z coordinate` The inclusive lower z coordinate.

upper z coordinate The exclusive upper z coordinate.

- **partial upper y** A Dirichlet boundary condition applied only to part of the upper y simulation boundary. The entire upper y boundary must be filled with a boundary condition. The lower coordinate is included in the simulation and the upper bound is exclusive. Also note that at the intersection of the X or Z simulation boundary the first computational cell of the Y boundary is included in the X or Z simulation boundary condition.

  lower x coordinate The inclusive lower x coordinate.

  upper x coordinate The exclusive upper x coordinate.

  lower z coordinate The inclusive lower z coordinate.

  upper z coordinate The exclusive upper z coordinate.

- **partial upper z** A Dirichlet boundary condition applied only to part of the upper z simulation boundary. The entire upper z boundary must be filled with a boundary condition. The lower coordinate is included in the simulation and the upper bound is exclusive. Also note that at the intersection of the X or Y simulation boundary the first computational cell of the Z boundary is included in the X or Y simulation boundary condition.

  lower x coordinate The inclusive lower x coordinate.

  upper x coordinate The exclusive upper x coordinate.

  lower y coordinate The inclusive lower y coordinate.

  upper y coordinate The exclusive upper y coordinate.

- **shape surface** Use a shape surface to specify a Dirichlet boundary condition on the surface of a geometry in your simulation.

  object name: Choose from any pre-defined geometries in the simulation.

Neumann Use a Neumann boundary condition to set the value of the derivative of the potential field, Phi, on the surface.

value The value of the derivative. Can be assigned a *Constant*, *Parameter*, or *SpaceTimeFunction* by right-clicking.

surface The surface on which the Dirichlet boundary condition should be set.

- **lower x** The lower x boundary of the simulation domain.
- **lower y** The lower y boundary of the simulation domain.
- **lower z** The lower z boundary of the simulation domain.
- **upper x** The upper x boundary of the simulation domain.
- **upper y** The upper y boundary of the simulation domain.
- **upper z** The upper z boundary of the simulation domain.
- **partial lower x** A Neumann boundary condition applied only to part of the lower x simulation boundary. The entire lower x boundary must be filled with a boundary condition. The lower coordinate is included in the simulation and the upper bound is exclusive. Also note that at the intersection of the Y or Z simulation boundary the first computational cell of the X boundary is included in the Y or Z simulation boundary condition.

  lower y coordinate The inclusive lower y coordinate.

  upper y coordinate The exclusive upper y coordinate.

  lower z coordinate The inclusive lower z coordinate.

`upper z coordinate` The exclusive upper z coordinate.

- **`partial lower y`** A Neumann boundary condition applied only to part of the lower y simulation boundary. The entire lower y boundary must be filled with a boundary condition. The lower coordinate is included in the simulation and the upper bound is exclusive. Also note that at the intersection of the X or Z simulation boundary the first computational cell of the Y boundary is included in the X or Z simulation boundary condition.

  `lower x coordinate` The inclusive lower x coordinate.

  `upper x coordinate` The exclusive upper x coordinate.

  `lower z coordinate` The inclusive lower z coordinate.

  `upper z coordinate` The exclusive upper z coordinate.

- **`partial lower z`** A Neumann boundary condition applied only to part of the lower z simulation boundary. The entire lower z boundary must be filled with a boundary condition. The lower coordinate is included in the simulation and the upper bound is exclusive. Also note that at the intersection of the X or Y simulation boundary the first computational cell of the Z boundary is included in the X or Y simulation boundary condition.

  `lower x coordinate` The inclusive lower x coordinate.

  `upper x coordinate` The exclusive upper x coordinate.

  `lower y coordinate` The inclusive lower y coordinate.

  `upper y coordinate` The exclusive upper y coordinate.

- **`partial upper x`** A Neumann boundary condition applied only to part of the upper x simulation boundary. The entire upper x boundary must be filled with a boundary condition. The lower coordinate is included in the simulation and the upper bound is exclusive. Also note that at the intersection of the Y or Z simulation boundary the first computational cell of the X boundary is included in the Y or Z simulation boundary condition.

  `lower y coordinate` The inclusive lower y coordinate.

  `upper y coordinate` The exclusive upper y coordinate.

  `lower z coordinate` The inclusive lower z coordinate.

  `upper z coordinate` The exclusive upper z coordinate.

- **`partial upper y`** A Neumann boundary condition applied only to part of the upper y simulation boundary. The entire upper y boundary must be filled with a boundary condition. The lower coordinate is included in the simulation and the upper bound is exclusive. Also note that at the intersection of the X or Z simulation boundary the first computational cell of the Y boundary is included in the X or Z simulation boundary condition.

  `lower x coordinate` The inclusive lower x coordinate.

  `upper x coordinate` The exclusive upper x coordinate.

  `lower z coordinate` The inclusive lower z coordinate.

  `upper z coordinate` The exclusive upper z coordinate.

- **`partial upper z`** A Neumann boundary condition applied only to part of the upper z simulation boundary. The entire upper z boundary must be filled with a boundary condition. The lower coordinate is included in the simulation and the upper bound is exclusive. Also note that at the intersection of the X or Y simulation boundary the first computational cell of the Z boundary is included in the X or Y simulation boundary condition.

  `lower x coordinate` The inclusive lower x coordinate.

upper x coordinate The exclusive upper x coordinate.

lower y coordinate The inclusive lower y coordinate.

upper y coordinate The exclusive upper y coordinate.

PoissonSolver

kind (not editable) Poisson Solver

relative permittivity Function giving the relative permittivity.

solver VSim allows the use of the direct solver SuperLU or one of the current choices for iterative solvers.

- **SuperLU** Use the SuperLU solver. This is a direct matrix solver which is not recommended for use in a simulation with a grid larger than a few thousand cells. For grids with more than a few thousand cells, use an iterative solver.
- **conjugate gradient** An iterative solver for symmetric positive definite matrices.

  tolerance The size of the error vector for stopping the iteration to solution.

  max iterations The maximum number of iterations allowed to achieve the solution. Anything above 30 is questionable.

  convergence metric This is described in the AztecOO user guide, p. 17, currently here. Metrics are one of:

  - **r0** Use the ratio of the L2 norms of the residual and the initial residual.
  - **rhs** Use the ratio of the L2 norms of the residual and the right-hand side.
  - **Anorm** Use the ratio of the L2 norm of the residual and the maximum absolute row sum of the matrix (L-infinity norm).
  - **noscaled** Use the ratio of the L2 norm of the residual.
  - **sol** Use the ratio of the maximum component of the residual and the sum of (L-infinity norm of the matrix times the maximum absolute component of the first iterate to the solution + the maximum absolute component of the right-hand side).
- **generalized minimum residual** An iterative solver. Actually the restarted GMRES. A very robust solver.

  Krylov vector space size Number of vectors used in Krylov subspace.

  max iterations The maximum number of iterations allowed to achieve the solution. Anything above 30 is questionable.

  tolerance The size of the error vector for stopping the iteration to solution.

  convergence metric This is described in the AztecOO user guide, p. 17, currently here. Metrics are one of:

  - **r0** Use the ratio of the L2 norms of the residual and the initial residual.
  - **rhs** Use the ratio of the L2 norms of the residual and the right-hand side.
  - **Anorm** Use the ratio of the L2 norm of the residual and the maximum absolute row sum of the matrix (L-infinity norm).
  - **noscaled** Use the ratio of the L2 norm of the residual.

  - **sol** Use the ratio of the maximum component of the residual and the sum of (L-infinity norm of the matrix times the maximum absolute component of the first iterate to the solution + the maximum absolute component of the right-hand side).

  orthogonalization How to orthogonalize the Krylov subspace. Options are:

  - **classic** Uses two steps of the classical Gram-Schmidt orthogonalization (Default).
  - **modified** Uses one step of the Modified Gram-Schmidt orthogonalization.

- **conjugate gradient squared**

  tolerance The size of the error vector for stopping the iteration to solution.

  max iterations The maximum number of iterations allowed to achieve the solution. Anything above 30 is questionable.

  convergence metric This is described in the AztecOO user guide, p. 17, currently here. Metrics are one of:

  - **r0** Use the ratio of the L2 norms of the residual and the initial residual.
  - **rhs** Use the ratio of the L2 norms of the residual and the right-hand side.
  - **Anorm** Use the ratio of the L2 norm of the residual and the maximum absolute row sum of the matrix (L-infinity norm).
  - **noscaled** Use the ratio of the L2 norm of the residual.
  - **sol** Use the ratio of the maximum component of the residual and the sum of (L-infinity norm of the matrix times the maximum absolute component of the first iterate to the solution + the maximum absolute component of the right-hand side).

- **biconjugate gradient stabilized**

  tolerance The size of the error vector for stopping the iteration to solution.

  max iterations The maximum number of iterations allowed to achieve the solution. Anything above 30 is questionable.

  convergence metric This is described in the AztecOO user guide, p. 17, currently here. Metrics are one of:

  - **r0** Use the ratio of the L2 norms of the residual and the initial residual.
  - **rhs** Use the ratio of the L2 norms of the residual and the right-hand side.
  - **Anorm** Use the ratio of the L2 norm of the residual and the maximum absolute row sum of the matrix (L-infinity norm).
  - **noscaled** Use the ratio of the L2 norm of the residual.
  - **sol** Use the ratio of the maximum component of the residual and the sum of (L-infinity norm of the matrix times the maximum absolute component of the first iterate to the solution + the maximum absolute component of the right-hand side).

- **transpose-free quasi-minimal residual**

  tolerance The size of the error vector for stopping the iteration to solution.

  max iterations The maximum number of iterations allowed to achieve the solution. Anything above 30 is questionable.

  convergence metric This is described in the AztecOO user guide, p. 17, currently here. Metrics are one of:

  - **r0** Use the ratio of the L2 norms of the residual and the initial residual.

- **`rhs`** Use the ratio of the L2 norms of the residual and the right-hand side.
- **`Anorm`** Use the ratio of the L2 norm of the residual and the maximum absolute row sum of the matrix (L-infinity norm).
- **`noscaled`** Use the ratio of the L2 norm of the residual.
- **`sol`** Use the ratio of the maximum component of the residual and the sum of (L-infinity norm of the matrix times the maximum absolute component of the first iterate to the solution + the maximum absolute component of the right-hand side).

`preconditioner`

- **`no preconditioner`**
- **`multigrid`** Multigrid has a large number of parameters. In most cases, the defaults are good. If one is using particles, the linear solve often does not matter in overall timing. More information is available in the VSim User Guide: VSim User Guide: Simulation Concepts: Fields

  `mg defaults`

  - **`SA`** Works best with symmetric matrices. Symmetric matrices occur when all boundary conditions in the problem are periodic.
  - **`DD`** Works best in general.
  - **`DD-ML`**
  - **`Maxwell`**

  `maximum levels` The maximum number of levels of multigrid smoother application before doing a direct solve on the smaller problem. For highly parallel problems this should be limited to 3-4 to minimize communication time.

  `smoother type` The smoother to apply at each multigrid level. The default is a good choice, but the user may try other smoothers to achieve best solving times. The choices are:

  - **`Gauss Seidel`**
  - **`symmetric variable block Gauss Seidel`**
  - **`Jacobi`**
  - **`Chebyshev`**
  - **`Aztec`**

  `smoother sweeps` The number of times to apply the smoother at each multigrid level. A good choice is 3, but the user may try other values to achieve best solving times.

  `when to smooth` Generally one wants to smooth both. The before and after choices are provided to the user to try other values to achieve best solving times.

  - **`both`**
  - **`before`**
  - **`after`**

  `coarse type` The type of solver to use at the coarsest level.

  - **`Jacobi`**
  - **`KLU`**

  `damping factor` Damping factor for smoothed aggregation. The default is a good choice. Users may try other values to achieve best solving times.

threshold This determines the multigrid aggregation at each level.

increase or decrease If set to increasing, level 0 will correspond to the finest level. If set to decreasing, max levels - 1 will correspond to the finest level. Should not affect convergence. The choices are:

- **increasing**
- **decreasing**

2.10.3 FieldDynamics for Prescribed Fields Simulations

This page describes the Field options available for prescribed field simulations, that is, when the `field solver` in the *Basic Settings* element is set to `prescribed fields`.

Fields

externalElectricField The Electric Field as prescribed by the user.

kind (un-editable) External Electric Field

description A space to provide a descriptive comment for the field.

field type electric field(cannot be changed)

field specification Either import h5 file or function defined.

- **function defined** Allows for manual specification of each component of the field

  component 0: The function defining the field in the 0th component. Can be a time varying function

  component 1: The function defining the field in the 1st component. Can be a time varying function

  component 2: The function defining the field in the 2nd component. Can be a time varying function

- **import h5 file** A vis schema compliant h5 file. It does require that the file be in the same directory as the simulation. An error message will be provided if the file fails to import.

  filename: The name of the .hdf5 file to be imported. Typical convention is simulationName_fieldName_dumpNum.h5

  lower bound 0: The cell index of the 0th component lower bound of the source field, to be imported to the matching index in the simulation. If left not applicable the entire field will be imported.

  lower bound 1: The cell index of the 1st component lower bound of the source field, to be imported to the matching index in the simulation. If left not applicable the entire field will be imported.

  lower bound 2: The cell index of the 2nd component lower bound of the source field, to be imported to the matching index in the simulation. If left not applicable the entire field will be imported.

  upper bound 0: The cell index of the 0th component upper bound of the source field, to be imported to the matching index in the simulation. If left not applicable the entire field will be imported.

upper bound 1: The cell index of the 1st component upper bound of the source field, to be imported to the matching index in the simulation. If left not applicable the entire field will be imported.

upper bound 2: The cell index of the 2nd component upper bound of the source field, to be imported to the matching index in the simulation. If left not applicable the entire field will be imported.

externalMagneticField The Magnetic Field as prescribed by the user.

kind (un-editable) External Magnetic Field

description A space to provide a descriptive comment for the field.

field type magnetic (cannot be changed)

field specification Either import h5 file or function defined.

- **function defined** Allows for manual specification of each component of the field

  component 0: The function defining the field in the 0th component. Can be a time varying function

  component 1: The function defining the field in the 1st component. Can be a time varying function

  component 2: The function defining the field in the 2nd component. Can be a time varying function

- **import h5 file** A vis schema compliant h5 file. It does require that the file be in the same directory as the simulation. An error message will be provided if the file fails to import.

  filename: The name of the .hdf5 file to be imported. Typical convention is simulationName_fieldName_dumpNum.h5

  lower bound 0: The cell index of the 0th component lower bound of the source field, to be imported to the matching index in the simulation. If left not applicable the entire field will be imported.

  lower bound 1: The cell index of the 1st component lower bound of the source field, to be imported to the matching index in the simulation. If left not applicable the entire field will be imported.

  lower bound 2: The cell index of the 2nd component lower bound of the source field, to be imported to the matching index in the simulation. If left not applicable the entire field will be imported.

  upper bound 0: The cell index of the 0th component upper bound of the source field, to be imported to the matching index in the simulation. If left not applicable the entire field will be imported.

  upper bound 1: The cell index of the 1st component upper bound of the source field, to be imported to the matching index in the simulation. If left not applicable the entire field will be imported.

  upper bound 2: The cell index of the 2nd component upper bound of the source field, to be imported to the matching index in the simulation. If left not applicable the entire field will be imported.

Time Dependence Properties

Time Dependence is available for prescribed field simulations.

`multiplier` Time varying multiplier to be applied to the prescribed fields.

2.11 Particle Dynamics

2.11.1 Kinetic Particles

Charged Particles

Charged Particles can be used to define any kinetic particle with given mass and charge.

To add Charged Particles, right click on the "KineticParticles" element, hover over the "Add ParticleSpecies" and choose "Charged Particles".

`kind` (not editable) Charged Particles

`nominal density` A positive value suggesting the nominal density for the particles. This will be used in conjunction with the weight setting to compute the density, weights, and number of particles in a macro particle for your kinetic particles.

`description` A space to provide a descriptive comment for the particle species.

`particle dynamics` Whether to use relativistic or non-relativistic particles.

- **`relativistic`**: Use the relativistic particle pushing algorithm to update the particle positions and velocities by including a gamma term.
- **`non-relativistic`**: Use the non-relativistic particle pushing algorithm to update the particle positions and velocities.

`particle weights` Whether to use constant or variable weight particles.

- **`variable weights`**: The weights of the macroparticles can vary throughout the simulation.
- **`constant weights`**: The weights of the macroparticles are constant throughout the simulation.
- **`managed weights`**: Variable weight particles that are managed to allow for maximum and minimum weights, and maximum and minimum number of macroparticles per cell.

  `macroparticles per cell for splitting` If more than this many macroparticles are in a cell splitting will not occur.

  `macroparticles per cell for combining` If fewer than this many macroparticles are in a cell combining will not occur.

  `minimum split particle weight` If the split particles would weigh under this value, splitting will not occur.

  `maximum combined particle weight` If the combined particle would weigh over this value, combination will not occur.

  `splitting periodicity` Number of time steps between assessing if particles should be split.

  `combining periodicity (not editable)` Number of time steps between assessing if particles should be combined.

  `splitting algorithm` Algorithm to use in determining split particle weights.

  `combining algorithm` Algorithm to use in determining combined particle weights.

`weight setting` Whether to use computed weights or explicitly set weights.

- **computed weights:** Let VSim calculate your macroparticle weights for you based on the number of macroparticles per cell you specify as well as the nominal density. The weights are calculated such that the number of particles in a macro particle is equal to the nominalDensity * cellVolume / macroparticles per cell.

  macroparticles per cell: The number of macroparticles per cell.

- **explicitly set weights:** Declare the number of particles in a macro particle explicitly.

  particles per macroparticle: The number of particles in a macroparticle.

molecule Molecule of the charged particle. A custom ion is available for those not pre-defined.

- **mass [amu]** The mass of a single real particle in atomic mass units.
- **charge number** The charge number of a single particle, multiple of the fundamental charge.
- **ionization energy [eV]** The ionization energy of the molecule in electron volts. This value will be used by particle interactions set thresholds and determine energy losses.

Additional Features:

Emitters:

- *Shape Settable Flux Emitter*
- *Slab Settable Flux Emitter*

Boundary Conditions:

- *Properties of Particle Boundary Conditions*

Loaders:

- *Particle Loader*

Electrons

Electrons are pre-defined to have a charge = -1.602176487e-19 C and a mass = 9.10938215e-31 kg.

To add Electrons, right click on the "KineticParticles" element, hover over the "Add ParticleSpecies" and choose "Electrons".

kind (not editable): Electrons

nominal density: A positive value suggesting the nominal density for the particles. This will be used in conjunction with the weight setting to compute the density, weights, and number of particles in a macro particle for your kinetic particles.

description A space to provide a descriptive comment for the particle species.

particle dynamics: Whether to use relativistic or non-relativistic particles.

- **relativistic**: Use the relativistic particle pushing algorithm to update the particle positions and velocities by including a gamma term.
- **non-relativistic**: Use the non-relativistic particle pushing algorithm to update the particle positions and velocities.

particle weights: Whether to use constant or variable weight particles.

- **variable weights**: The weights of the macroparticles can vary throughout the simulation.
- **constant weights**: The weights of the macroparticles are constant throughout the simulation.

- **`managed weights`**: Variable weight particles that are managed to allow for maximum and minimum weights, and maximum and minimum number of macroparticles per cell.

  `macroparticles per cell for splitting:` If more than this many macroparticles are in a cell splitting will not occur.

  `macroparticles per cell for combining:` If fewer than this many macroparticles are in a cell combining will not occur.

  `minimum split particle weight:` If the split particles would weigh under this value, splitting will not occur.

  `maximum combined particle weight:` If the combined particle would weigh over this value, combination will not occur.

  `splitting periodicity:` Number of time steps between assessing if particles should be split.

  `combining periodicity (not editable):` Number of time steps between assessing if particles should be combined.

  `splitting algorithm:` Algorithm to use in determining split particle weights.

  `combining algorithm:` Algorithm to use in determining combined particle weights.

`weight setting:` Whether to use computed weights or explicitly set weights.

- **`computed weights:`** Let VSim calculate your macroparticle weights for you based on the number of macroparticles per cell you specify as well as the nominal density. The weights are calculated such that the number of particles in a macro particle is equal to the nominalDensity * cellVolume / macroparticles per cell.

  `macroparticles per cell`: The number of macroparticles per cell.

- **`explicitly set weights:`** Declare the number of particles in a macro particle explicitly.

  `particles per macroparticle`: The number of particles in a macroparticle.

Additional Features:

Emitters:

- *Shape Settable Flux Emitter*
- *Secondary Electron Emitter*
- *Slab Settable Flux Emitter*

Boundary Conditions:

- *Properties of Particle Boundary Conditions*

Loaders:

- *Particle Loader*

Neutral Particles

Neutral Particles of the *gas kinds* listed below are pre-defined with the appropriate charge and mass values. Users can also add custom neutral species.

To add Neutral Particles, right click on the "KineticParticles" element, hover over the "Add ParticleSpecies" and choose "Neutral Particles".

`kind` (not editable) Neutral Particles

`nominal density` A positive value suggesting the nominal density for the particles. This will be used, in conjunction with the weight setting, to compute the density, weights, and number of particles in a macro particle for your kinetic particles.

`description` A space to provide a descriptive comment for the particle species.

`particle dynamics` Whether to use relativistic or non-relativistic particles

- **`relativistic`**: Use the relativistic particle pushing algorithm to update the particle positions and velocities by including a gamma term.
- **`non-relativistic`**: Use the non-relativistic particle pushing algorithm to update the particle positions and velocities.

`particle weights` Whether to use constant or variable weight particles.

- **`variable weights`** The weights of the macroparticles can vary throughout the simulation.
- **`constant weights`** The weights of the macroparticles are constant throughout the simulation.
- **`managed weights`** Variable weight particles that are managed to allow for maximum and minimum weights, and maximum and minimum number of macroparticles per cell.

  `macroparticles per cell for splitting` If more than this many macroparticles are in a cell splitting will not occur.

  `macroparticles per cell for combining` If fewer than this many macroparticles are in a cell combining will not occur.

  `minimum split particle weight` If the split particles would weigh under this value, splitting will not occur.

  `maximum combined particle weight` If the combined particle would weigh over this value, combination will not occur.

  `splitting periodicity` Number of time steps between assessing if particles should be split.

  `combining periodicity (not editable)` Number of time steps between assessing if particles should be combined.

  `splitting algorithm` Algorithm to use in determining split particle weights.

  `combining algorithm` Algorithm to use in determining combined particle weights.

`weight setting` Whether to use computed weights or explicitly set weights.

- **`computed weights`** Let VSim calculate your macroparticle weights for you, based on the number of macroparticles per cell you specify as well as the nominal density. The weights are calculated such that the number of particles in a macro particle is equal to the nominalDensity * cellVolume / macroparticles per cell.

  `macroparticles per cell`: The number of macroparticles per cell.

- **`explicitly set weights`** Declare the number of particles in a macro particle explicitly.

  `particles per macroparticle`: The number of particles in a macroparticle.

`molecule` Molecule of the neutral particle. Options are:

- **`H`**: Hydrogen (atomic)
- **`H2`**: Hydrogen (molecular)
- **`He`**: Helium

- **`Ar`**: Argon
- **`Xe`**: Xenon
- **`Rn`**: Radon
- **`Kr`**: Krypton
- **`O`**: Oxygen (atomic)
- **`O2`**: Oxygen (molecular)
- **`Ne`**: Neon
- **`N`**: Nitrogen (atomic)
- **`N2`**: Nitrogen (molecular)
- **`custom molecule`** Set the mass (in amu) and ionization energy (in eV) of a custom gas.
  - **`mass [amu]`** The mass of a single real particle in atomic mass units.
  - **`ionization energy [eV]`** The ionization energy of the molecule in electron volts.

Additional Features:

Emitters:

- *Shape Settable Flux Emitter*
- *Sputter Emitter*
- *Slab Settable Flux Emitter*

Boundary Conditions:

- *Properties of Particle Boundary Conditions*

Loaders:

- *Particle Loader*

Field Scaling Electrons

These are a special type of electron used in testing. Most notably, they consist of only one physical particle per macroparticle. They are in fact given a variable weight, which is used to track whether the particles have been created via secondary emission. Each particle also has a scaling value, allowing for multiple voltage levels to exist in the simulation simultaneously. Since there is only one physical particle per macroparticle, these particles will have negligible effect on the simulation.

The primary use of these particles is to scan a structure for multipacting resonances across a number of power levels at the same time.

To add Field Scaling Electrons, right click on the “KineticParticles” element, hover over the “Add TestParticleSpecies” and choose “Field Scaling Electrons”.

Field Scaling Electron Loader

This particle loader is very similar to a standard particle species loader, with some notable exceptions.

`description`: A space to provide a descriptive comment for the particle species.

load density: Only the relative density of particles may be specified, as this is the same as the physical density.

> **physical density**: This will specify the number of particles per cell.

particle load placement: At this time particles may only be loaded according to a bit-reversed algorithm.

load duration: Whether to only load the particles at the beginning of the simulation or repeatedly.

- **initialize only**: Particles will only be loaded at the beginning of the simulation.
- **repeat loading**: Particles will be loaded according to the parameters below.

  start time: The time at which to start loading particles.

  stop time: The time at which to stop loading particles.

  load after initialization: Loading will repeat in all cells during the loading period.

  load upon shift: Load particles into cells brought into the simulation by a moving window.

scaling factor: This will determine the minimum value, maximum value, and scaling factor applied to the field scaling electrons. So for example with a minimum value of 10, and a scaling factor of 3, that individual electron is charged to 30V.

> **minimum scaling factor**: The minimum voltage of the particles loaded.
>
> **maximum scaling factor**: The maximum voltage of the particles loaded.
>
> **number of scale factors**: The number of scale factors, or steps, between the minimum scaling factor and maximum scaling factor.

volume: The volume in which to load particles.

- **cartesian 3d slab**
- **cylindrical 2d slab**

Additional Features:

Fewer additional features are available for Field Scaling Electrons. The only emitter available for Field Scaling Electrons is a secondary emitter.

- *Secondary Electron Emitter*

And the only available particle boundary conditions are: Absorbing, Boundary Absorb and Save, Cut-Cell Absorb and Save, Interior Absorb and Save, and Reflecting.

- *Properties of Particle Boundary Conditions*

Particle Emitters

Shape Settable Flux Emitter

All particle types may emit from a shape settable flux emitter. Certain emission specifications are only available based on particle type and particle weights specification. Available in cartesian coordinate simulations only.

start time Time to start emitting particles.

stop time Time to stop emitting particles.

emission specification: Specification of the emitted particles, note that the specification options vary for constant or variable/managed weight particles.

- **`emission current density:`** Only available with variable/managed weight specified particles.

  `emission current density`: Specify the current density of the emitter (amps/meter^2). Can be a spatial profile.

  `velocity coordinate system`: Either **global** or **surface**. A global coordinate system will specify the emission velocities according to global axis. A surface coordinate system will set the emission directions according to the normal of the emission object. A positive value will emit particles away from the shape and a negative value will emit particles into the shape.

  - **`average velocity 0`**: The average (mean) speed of particles in the x-direction when *velocity coordinate system* is set to "global". If set to "surface" then this will be the average velocity for the direction normal to the emitting surface.
  - **`average velocity 1`**: The average (mean) speed of particles in the y-direction when *velocity coordinate system* is set to "global". If set to "surface" then this will be the average velocity for a direction perpendicular to the emitting surface.
  - **`average velocity 2`**: The average (mean) speed of particles in the z-direction when *velocity coordinate system* is set to "global". If set to "surface" then this will be the average velocity for a direction perpendicular to the emitting surface.
  - **`thermal velocity 0`**: A spread (standard deviation) for particle speeds in the 0 direction.
  - **`thermal velocity 1`**: A spread (standard deviation) for particle speeds in the 1 direction.
  - **`thermal velocity 2`**: A spread (standard deviation) for particle speeds in the 2 direction.

- **`emission flux:`** Only available with variable/managed weight specified particles.

  `emission flux`: Specify the flux of the emitter (particles/meter^2). Can be a spatial profile.

  `velocity coordinate system`: Either **global** or **surface**. A global coordinate system will specify the emission velocities according to global axis. A surface coordinate system will set the emission directions according to the normal of the emission object. A positive value will emit particles away from the shape and a negative value will emit particles into the shape.

  - **`average velocity 0`**: The average (mean) speed of particles in the x-direction when *velocity coordinate system* is set to "global". If set to "surface" then this will be the average velocity for the direction normal to the emitting surface.
  - **`average velocity 1`**: The average (mean) speed of particles in the y-direction when *velocity coordinate system* is set to "global". If set to "surface" then this will be the average velocity for a direction perpendicular to the emitting surface.
  - **`average velocity 2`**: The average (mean) speed of particles in the z-direction when *velocity coordinate system* is set to "global". If set to "surface" then this will be the average velocity for a direction perpendicular to the emitting surface.
  - **`thermal velocity 0`**: A spread (standard deviation) for particle speeds in the 0 direction.
  - **`thermal velocity 1`**: A spread (standard deviation) for particle speeds in the 1 direction.
  - **`thermal velocity 2`**: A spread (standard deviation) for particle speeds in the 2 direction.

- **`emission current:`** Only available with constant weight particles.

  `emission current`: Specify the total emitted current per second from the emitter (amps/second).

  `velocity coordinate system`: Either **global** or **surface**. A global coordinate system will specify the emission velocities according to global axis. A surface coordinate system will set the emission directions according to the normal of the emission object. A positive value will emit particles away from the shape and a negative value will emit particles into the shape.

  - **`average velocity 0`**: The average (mean) speed of particles in the x-direction when *velocity coordinate system* is set to "global". If set to "surface" then this will be the average velocity for the direction normal to the emitting surface.
  - **`average velocity 1`**: The average (mean) speed of particles in the y-direction when *velocity coordinate system* is set to "global". If set to "surface" then this will be the average velocity for a direction perpendicular to the emitting surface.
  - **`average velocity 2`**: The average (mean) speed of particles in the z-direction when *velocity coordinate system* is set to "global". If set to "surface" then this will be the average velocity for a direction perpendicular to the emitting surface.
  - **`thermal velocity 0`**: A spread (standard deviation) for particle speeds in the 0 direction.
  - **`thermal velocity 1`**: A spread (standard deviation) for particle speeds in the 1 direction.
  - **`thermal velocity 2`**: A spread (standard deviation) for particle speeds in the 2 direction.

- **`emission rate:`** Only available with constant weight particles.

  `emission rate`: Specify the total emitted particles per second from the emitter (particle/second).

  `velocity coordinate system`: Either **global** or **surface**. A global coordinate system will specify the emission velocities according to global axis. A surface coordinate system will set the emission directions according to the normal of the emission object. A positive value will emit particles away from the shape and a negative value will emit particles into the shape.

  - **`average velocity 0`**: The average (mean) speed of particles in the x-direction when *velocity coordinate system* is set to "global". If set to "surface" then this will be the average velocity for the direction normal to the emitting surface.
  - **`average velocity 1`**: The average (mean) speed of particles in the y-direction when *velocity coordinate system* is set to "global". If set to "surface" then this will be the average velocity for a direction perpendicular to the emitting surface.
  - **`average velocity 2`**: The average (mean) speed of particles in the z-direction when *velocity coordinate system* is set to "global". If set to "surface" then this will be the average velocity for a direction perpendicular to the emitting surface.
  - **`thermal velocity 0`**: A spread (standard deviation) for particle speeds in the 0 direction.
  - **`thermal velocity 1`**: A spread (standard deviation) for particle speeds in the 1 direction.
  - **`thermal velocity 2`**: A spread (standard deviation) for particle speeds in the 2 direction.

- **`Fowler Nordheim Emission:`** Specify particle emission according to the Fowler-Nordheim model. Only available with variable/managed weight electron particle species.

  `work function [eV]`: Work function of the material from which emission is occurring.

  `A`: Coefficient A of the Fowler-Nordheim emission model.

  `B`: Coefficient B of the Fowler-Nordheim emission model.

  `field enhancement`: Multiplies the measured electric field by this amount.

  `Cv`: Coefficient Cv of the Fowler-Nordheim emission model.

  `Cy`: Coefficient Cy of the Fowler-Nordheim emission model.

- **`Richardson Dushman Emission:`**

  Specify particle emission according to the Richardson-Dushman model. Only available with variable/managed weight electron particle species.

work function [eV]: Work function of the material from which emission is occurring. Parameter in the Richardson-Dushman model.

field evaluation offset: The offset from the surface where the field resulting from the particle is evaluated.

temperature (K): Temperature of the material from which emission is occurring. Parameter in the Richardson-Dushman model.

field enhancement: Multiplies the measured electric field by this amount.

flux multiplier: Multiplies the resulting output current by this amount.

- **Child Langmuir Emission:** Specify particle emission according to the Child Langmuir model. Only available with variable/managed weight electron particle species.
  - **average velocity 0**: The average (mean) speed of particles in the 0 direction.
  - **average velocity 1**: The average (mean) speed of particles in the 1 direction.
  - **average velocity 2**: The average (mean) speed of particles in the 2 direction.
  - **thermal velocity 0**: A spread (standard deviation) for particle speeds in the 0 direction.
  - **thermal velocity 1**: A spread (standard deviation) for particle speeds in the 1 direction.
  - **thermal velocity 2**: A spread (standard deviation) for particle speeds in the 2 direction.

emission surface The surface off of which to emit.

object name: The name of the geometry off of which to emit.

emission offset: The distance away from the object that emitted particles are placed, as a fraction of a cell length.

macroparticle emission: Only available with variable/managed weight particles. This allows for the specification of the macroparticle emission independent of the emitted particles. Used to handle computational concerns around macroparticle weight.

macroparticle rate: Number of macroparticles to emit per timestep. This value can be modified by the macroparticle emission profile.

macroparticle emission profile: Spatial profile for emission of the macroparticles. If this corresponded to half of the emissions shape, half of the number of macroparticles specified in *macroparticle rate* would be emitted, while the emission specification would be unaffected.

Slab Settable Flux Emitter

All particle types may emit from a slab settable flux emitter. Certain emission specifications are only available based on particle type and particle weights specification. Available in all coordinate simulations.

start time Time to start emitting particles in seconds.

stop time Time to stop emitting particles in seconds.

emission specification Specification of the emitted particles, note that the specification options vary for constant or variable/managed weight particles.

- **emission current density**

  emission current density Specify the current density of the emitter (amps/meter^2). Can be a spatial profile.

velocity coordinate system Either global or surface. A global coordinate system will specify the emission velocities according to global axis. A surface coordinate system will set the emission directions according to the normal of the emission object. So in a surface coordinate system a lower simulation bounds the emission velocity must be negative to emit into the simulation space, for an upper simulation boundary the particles must be positive to emit into the simulation space.

- **average velocity 0**: The average (mean) speed of particles in the x-direction when *velocity coordinate system* is set to "global". If set to "surface" then this will be the average velocity for the direction normal to the emitting surface.
- **average velocity 1**: The average (mean) speed of particles in the y-direction when *velocity coordinate system* is set to "global". If set to "surface" then this will be the average velocity for a direction perpendicular to the emitting surface.
- **average velocity 2**: The average (mean) speed of particles in the z-direction when *velocity coordinate system* is set to "global". If set to "surface" then this will be the average velocity for a direction perpendicular to the emitting surface.
- **thermal velocity 0**: A spread (standard deviation) for particle speeds in the 0 direction.
- **thermal velocity 1**: A spread (standard deviation) for particle speeds in the 1 direction.
- **thermal velocity 2**: A spread (standard deviation) for particle speeds in the 2 direction.

• **emission flux**

emission flux Specify the flux of the emitter (particles/meter^2). Can be a spatial profile.

velocity coordinate system Either global or surface. A global coordinate system will specify the emission velocities according to global axis. A surface coordinate system will set the emission directions according to the normal of the emission surface. So in a surface coordinate system a lower simulation bounds the emission velocity must be negative to emit into the simulation space, for an upper simulation boundary the particles must be positive to emit into the simulation space.

- **average velocity 0**: The average (mean) speed of particles in the x-direction when *velocity coordinate system* is set to "global". If set to "surface" then this will be the average velocity for the direction normal to the emitting surface.
- **average velocity 1**: The average (mean) speed of particles in the y-direction when *velocity coordinate system* is set to "global". If set to "surface" then this will be the average velocity for a direction perpendicular to the emitting surface.
- **average velocity 2**: The average (mean) speed of particles in the z-direction when *velocity coordinate system* is set to "global". If set to "surface" then this will be the average velocity for a direction perpendicular to the emitting surface.
- **thermal velocity 0**: A spread (standard deviation) for particle speeds in the 0 direction.
- **thermal velocity 1**: A spread (standard deviation) for particle speeds in the 1 direction.
- **thermal velocity 2**: A spread (standard deviation) for particle speeds in the 2 direction.

• **emission current**

emission current Specify the total emitted current per second from the emitter (amps/second).

velocity coordinate system Either global or surface. A global coordinate system will specify the emission velocities according to global axis. A surface coordinate system will set the emission directions according to the normal of the emission surface. So in a surface coordinate system a lower simulation bounds the emission velocity must be negative to emit into the simulation space, for an upper simulation boundary the particles must be positive to emit into the simulation space.

- **average velocity 0**: The average (mean) speed of particles in the x-direction when *velocity coordinate system* is set to "global". If set to "surface" then this will be the average velocity for the direction normal to the emitting surface.
- **average velocity 1**: The average (mean) speed of particles in the y-direction when *velocity coordinate system* is set to "global". If set to "surface" then this will be the average velocity for a direction perpendicular to the emitting surface.
- **average velocity 2**: The average (mean) speed of particles in the z-direction when *velocity coordinate system* is set to "global". If set to "surface" then this will be the average velocity for a direction perpendicular to the emitting surface.
- **thermal velocity 0**: A spread (standard deviation) for particle speeds in the 0 direction.
- **thermal velocity 1**: A spread (standard deviation) for particle speeds in the 1 direction.
- **thermal velocity 2**: A spread (standard deviation) for particle speeds in the 2 direction.

- **emission rate**

emission rate Specify the total number of particles emitted per second (particles/second)

velocity coordinate system Either global or surface. A global coordinate system will specify the emission velocities according to global axis. A surface coordinate system will set the emission directions according to the normal of the emission surface. So in a surface coordinate system a lower simulation bounds the emission velocity must be negative to emit into the simulation space, for an upper simulation boundary the particles must be positive to emit into the simulation space.

- **average velocity 0**: The average (mean) speed of particles in the x-direction when *velocity coordinate system* is set to "global". If set to "surface" then this will be the average velocity for the direction normal to the emitting surface.
- **average velocity 1**: The average (mean) speed of particles in the y-direction when *velocity coordinate system* is set to "global". If set to "surface" then this will be the average velocity for a direction perpendicular to the emitting surface.
- **average velocity 2**: The average (mean) speed of particles in the z-direction when *velocity coordinate system* is set to "global". If set to "surface" then this will be the average velocity for a direction perpendicular to the emitting surface.
- **thermal velocity 0**: A spread (standard deviation) for particle speeds in the 0 direction.
- **thermal velocity 1**: A spread (standard deviation) for particle speeds in the 1 direction.
- **thermal velocity 2**: A spread (standard deviation) for particle speeds in the 2 direction.

- **`Fowler Nordheim Emission`** Specify particle emission according to the Fowler-Nordheim model. Only available with variable/managed weight electron particle species.

  `work function [eV]`: Work function of the material from which emission is occurring.

  `A`: Coefficient A of the Fowler-Nordheim emission model.

  `B`: Coefficient B of the Fowler-Nordheim emission model.

  `field enhancement`: Multiplies the measured electric field by this amount.

  `Cv`: Coefficient Cv of the Fowler-Nordheim emission model.

  `Cy`: Coefficient Cy of the Fowler-Nordheim emission model

- **`Richardson Dushman Emission`** Specify particle emission according to the Richardson-Dushman model. Only available with variable/managed weight electron particle species.

  `work function [eV]`: Work function of the material from which emission is occurring. Parameter in the Richardson-Dushman model.

  `field evaluation offset`:

  `temperature (K)`: Temperature of the material from which emission is occurring. Parameter in the Richardson-Dushman model.

  `field enhancement`: Multiplies the measured electric field by this amount.

  `flux multiplier`: Multiplies the resulting output current by this amount.

- **`Child Langmuir Emission`** Specify particle emission according to the Child Langmuir model. Only available with variable/managed weight electron particle species.

  - **`average velocity 0`**: The average (mean) speed of particles in the 0 direction.
  - **`average velocity 1`**: The average (mean) speed of particles in the 1 direction.
  - **`average velocity 2`**: The average (mean) speed of particles in the 2 direction.
  - **`thermal velocity 0`**: A spread (standard deviation) for particle speeds in the 0 direction.
  - **`thermal velocity 1`**: A spread (standard deviation) for particle speeds in the 1 direction.
  - **`thermal velocity 2`**: A spread (standard deviation) for particle speeds in the 2 direction.

`emission surface`

- **`lower x`** The lower x simulation boundary.
- **`lower y`** The lower y simulation boundary.
- **`lower z`** The lower z simulation boundary.
- **`upper x`** The upper x simulation boundary.
- **`upper y`** The upper y simulation boundary.
- **`upper z`** The upper z simulation boundary.

`emission offset` The distance away from the object that emitted particles are placed, as a fraction of a cell length.

`macroparticle emission` Only available with variable/managed weight particles. This allows for the specification of the macroparticle emission independent of the emitted particles. Used to handle computational concerns around macroparticle weight.

`macroparticle rate` Number of macroparticles to emit per timestep. This value can be modified by the macroparticle emission profile.

macroparticle emission profile Spatial profile for emission of the macroparticles. If this corresponded to half of the emissions shape, half of the number of macroparticles specified in *macroparticle rate* would be emitted, while the emission specification would be unaffected.

Secondary Electron Emitter

Secondary electrons can be emitted from a electron species, charged particle species, or neutral particle species. They can then be emitted into a separate electron species, or into the same electron species.

emitter type

- **secondary electron emitter** This emitter is for use on simulation boundaries. An instance of *secElec*, follow the link for more information.

  particle boundary condition The particle boundary condition to emit secondary electrons from. It must be of the type *Boundary Absorb and Save* or *Cut-Cell Absorb and Save*. If selected as a particle boundary condition of a different species, particles from that species will emit into the electron species.

  material The material for computing the secondary electron yield. One of either “copper” or “stainless”.

  suppression energy Suppress emission of secondary electrons unless the emitted energy is greater than this number (in eV). Useful if electrons are emitted in a region where the local electric field would push them back into the wall in the same time step. In this case, set to a large number.

- **interior secondary electron emitter** This emitter is for use with particle boundary conditions internal to the simulation space. An instance of *secElec*, follow the link for more information. Unless the emission boundary is a plane, it is recommended to use a `secondary electron emitter`.

  particle boundary condition The particle boundary condition to emit secondary electrons from. It must be of the type *Interior Absorb and Save*. It can be from a different particle species, which will cause incident particles of that species to sputter as the neutral particle species.

  material The material for computing the secondary electron yield. One of “copper” or “stainless”.

  suppression energy Suppress emission of secondary electrons unless the emitted energy is greater than this number (in eV). Useful if electrons are emitted in a region where the local electric field would push them back into the wall in the same time step. In this case, set to a large number.

  emission axis Specifies the direction of the outward-facing normal direction of the emitter. Options are: 0 for the x direction (in cartesian, z in cylindrical); 1 for the y direction (R in cylindrical); and 2 for the z direction.

  emission direction An option to flip the sign of the normal direction. Choosing **positive** will make the emission in the positive x, y, or z direction, and a choice of **negative** will result in emission in the negative x, y, or z direction.

  emission coordinate The absolute position of the emission plane with respect to zero.

- **simple secondary electron emitter** A secondary emitter with a user specified SEY function. An instance of *simpleSec*, follow the link for more information.

  particle boundary condition The particle boundary condition to emit secondary electrons from. It must be of the type *Boundary Absorb and Save* or *Cut-Cell Absorb and Save*. If selected as a particle boundary condition of a different species, particles from that species will emit into the electron species.

suppression energy Suppress emission of secondary electrons unless the emitted energy is greater than this number (in eV). Useful if electrons are emitted in a region where the local electric field would push them back into the wall in the same time step. In this case, set to a large number.

emitted energy The energy of the emitted electrons if constant weight particles are used.

SEY(E) A function that defines the SEY curve. This function should be a function of only x, where x gets interpreted as the impact energy of the impacting electron.

- **constant probability secondary electron emitter** This emitter will emit secondary electrons according to a user defined probability. An instance of *secElec*, follow the link for more information.

  particle boundary condition The particle boundary condition to emit secondary electrons from. It must be of the type *Boundary Absorb and Save* or *Cut-Cell Absorb and Save*.

  emission probability The probability that a secondary electron is emitted, between 0.0 and 1.0.

Sputter Emitter

Neutral particle types may sputter from other particle species.

description A space to provide a descriptive comment of the emitter.

emitter type The type of emitter to use, either a sputter emitter or interior sputter emitter.

- **sputter emitter** This type of emitter should be used when emitting off a simulation particle boundary condition.

  particle boundary condition The particle boundary condition to sputter particles from. It must be of the type *Boundary Absorb and Save*. It can be a particle boundary condition from a different particle species, which will cause incident particles of that species to sputter as the neutral particle species.

  material properties This will determine the type of atom that is sputtered. Should be the same as the species to which this emitter is applied.

  sputtered velocity sigma The standard deviation of the velocity distribution of emitted particles.

- **interior sputter emitter** This emitter is for use when emitting off an interior particle boundary condition.

  particle boundary condition The particle boundary condition to sputter particles from. It must be of the type *Interior Absorb and Save* or *Cut Cell Absorb and Save*. It can be a particle boundary condition from a different particle species, which will cause incident particles of that species to sputter as the neutral particle species.

  material properties This will determine the type of atom that is sputtered. Should be the same as the species to which this emitter is applied.

  sputtered velocity sigma The standard deviation of the velocity distribution of emitted particles.

  emission axis The axis along which particles are to be emitted, 0 for the 0th dimension, 1 for the 1st dimension or 2 for the 2nd dimension.

  emission direction Set to either positive or negative, this is the direction in which particles are emitted.

  emission coordinate The coordinate at which particles are actually emitted. Effectively this is an offset from the position of the particle boundary condition.

Properties of Particle Boundary Conditions

Particle boundary conditions may be set to any particle type. If a particle boundary condition is not set on a simulation boundary, it is automatically set to absorbing.

`Absorbing:` This type of particle boundary condition will absorb incident particles. It will not save any data from them, so they may not be used in histories or secondary emitters.

`volume`

- **`lower x slab`**
- **`lower y slab`**
- **`lower z slab`**
- **`upper x slab`**
- **`upper y slab`**
- **`upper z slab`**
- **`cartesian 3d slab`**
- **`cylindrical 2d slab`**
- **`index 3d slab`**

`Boundary Absorb and Save:` This type of boundary condition will absorb incident particles and save them for other uses, such as histories or secondary emission.

`volume`: The simulation boundary over which to apply this condition.

`Boundary Accumulate:` Incident particles will be accumulated on this boundary and not allowed to move. This allows for the charging of a simulation boundary. The accumulated particles will be stored in a new particle species, appended with the prefix heavy.

`volume`: The simulation boundary to accumulate particles on.

`Boundary Diffuse Reflector:` This type of particle boundary condition will reflect particles back into the simulation space, with an user supplied velocity. It may only be applied on simulation boundaries.

- **`average velocity 0`**
- **`average velocity 1`**
- **`average velocity 2`**
- **`thermal velocity 0`**
- **`thermal velocity 1`**
- **`thermal velocity 2`**
- **`volume`**

  `lower x slab`

  `lower y slab`

  `lower z slab`

  `upper x slab`

  `upper y slab`

  `upper z slab`

  `lower r slab`

upper r slab

Cut-Cell Absorb and Save: This type of boundary condition will absorb incident particles and save them for other uses, such as histories or secondary emission. It may be applied to geometries. In VSim9.0, only one Cut-Cell Absorb and Save boundary condition may be used per particle species. If this boundary condition is used Interior Absorb and Save boundary conditions may not be used.

volume

- **object name** The name of the geometry over which to apply this condition.

Cut-Cell Accumulate: Incident particles will be accumulated on this boundary and not allowed to move. This allows for the charging of a shape The accumulated particles will be stored in a new particle species, appended with the prefix heavy.

volume: The shape to accumulate particles on.

Interior Absorb and Save: This type of boundary condition will absorb incident particles and save them for use with histories or secondary emission. It may be set to an internal volume of the simulation space.

volume: Depending on simulation properties this will be either a:

- **cylindrical 2D slab**
- **cartesian 3D slab**

Interior Accumulate: Incident particles will be accumulated on this boundary and not allowed to move. This allows for the charging of a user specified interior plane. The accumulated particles will be stored in a new particle species, appended with the prefix heavy.

volume

xMin The minimum x position of the boundary condition.

xMax The maximum x position of the boundary condition.

yMin The minimum y position of the boundary condition.

yMax The maximum y position of the boundary condition.

zMin The minimum z position of the boundary condition.

zMax The maximum z position of the boundary condition.

surface The surface of the described volume to accumulate particles on.

Interior Diffuse Reflector: This type of particle boundary condition will reflect particles back into the simulation space, with an user supplied velocity. It may only be applied on internal planes of the simulation space

average velocity 0

average velocity 1

average velocity 2

thermal velocity 0

thermal velocity 1

thermal velocity 2

orthogonal direction

Incoming direction of particles to apply the boundary condition on.

negative x

  `x coordinate` x coordinate of the boundary condition.

  `lower y coordinate` lower y coordinate of the boundary condition.

  `lower z coordinate` lower z coordinate of the boundary condition.

  `upper y coordinate` upper y coordinate of the boundary condition.

  `upper z coordinate` upper z coordinate of the boundary condition.

`negative y`

  `y coordinate` y coordinate of the boundary condition.

  `lower x coordinate` lower x coordinate of the boundary condition.

  `lower z coordinate` lower z coordinate of the boundary condition.

  `upper x coordinate` upper x coordinate of the boundary condition.

  `upper z coordinate` upper z coordinate of the boundary condition.

`negative z`

  `z coordinate` z coordinate of the boundary condition.

  `lower x coordinate` lower x coordinate of the boundary condition.

  `lower y coordinate` lower y coordinate of the boundary condition.

  `upper x coordinate` upper x coordinate of the boundary condition.

  `upper y coordinate` upper y coordinate of the boundary condition.

`positive x`

  `x coordinate` x coordinate of the boundary condition.

  `lower y coordinate` lower y coordinate of the boundary condition.

  `lower z coordinate` lower z coordinate of the boundary condition.

  `upper y coordinate` upper y coordinate of the boundary condition.

  `upper z coordinate` upper z coordinate of the boundary condition.

`positive y`

  `y coordinate` y coordinate of the boundary condition.

  `lower x coordinate` lower x coordinate of the boundary condition.

  `lower z coordinate` lower z coordinate of the boundary condition.

  `upper x coordinate` upper x coordinate of the boundary condition.

  `upper z coordinate` upper z coordinate of the boundary condition.

`positive z`

  `z coordinate` z coordinate of the boundary condition.

  `lower x coordinate` lower x coordinate of the boundary condition.

  `lower y coordinate` lower y coordinate of the boundary condition.

  `upper x coordinate` upper x coordinate of the boundary condition.

  `upper y coordinate` upper y coordinate of the boundary condition.

`Interior Partial Transmitter:` An interior partial transmitter will transmit a user specified fraction of the incident particles and absorb the remainder. It will only do this in the orthogonal direction, not if particles are to come from the other direction.

`orthogonal direction`: Incoming direction of particles to apply the boundary condition on.

`negative x`

`x coordinate` x coordinate of the boundary condition.

`lower y coordinate` lower y coordinate of the boundary condition.

`lower z coordinate` lower z coordinate of the boundary condition.

`upper y coordinate` upper y coordinate of the boundary condition.

`upper z coordinate` upper z coordinate of the boundary condition.

`negative y`

`y coordinate` y coordinate of the boundary condition.

`lower x coordinate` lower x coordinate of the boundary condition.

`lower z coordinate` lower z coordinate of the boundary condition.

`upper x coordinate` upper x coordinate of the boundary condition.

`upper z coordinate` upper z coordinate of the boundary condition.

`negative z`

`z coordinate` z coordinate of the boundary condition.

`lower x coordinate` lower x coordinate of the boundary condition.

`lower y coordinate` lower y coordinate of the boundary condition.

`upper x coordinate` upper x coordinate of the boundary condition.

`upper y coordinate` upper y coordinate of the boundary condition.

`positive x`

`x coordinate` x coordinate of the boundary condition.

`lower y coordinate` lower y coordinate of the boundary condition.

`lower z coordinate` lower z coordinate of the boundary condition.

`upper y coordinate` upper y coordinate of the boundary condition.

`upper z coordinate` upper z coordinate of the boundary condition.

`positive y`

`y coordinate` y coordinate of the boundary condition.

`lower x coordinate` lower x coordinate of the boundary condition.

`lower z coordinate` lower z coordinate of the boundary condition.

`upper x coordinate` upper x coordinate of the boundary condition.

`upper z coordinate` upper z coordinate of the boundary condition.

`positive z`

`z coordinate` z coordinate of the boundary condition.

`lower x coordinate` lower x coordinate of the boundary condition.

lower y coordinate lower y coordinate of the boundary condition.

upper x coordinate upper x coordinate of the boundary condition.

upper y coordinate upper y coordinate of the boundary condition.

Specular Reflecting: This type of particle boundary condition will reflect particles back into the simulation space. It may only be applied on simulation boundaries.

volume

- **lower x slab**
- **lower y slab**
- **lower z slab**
- **upper x slab**
- **upper y slab**
- **upper z slab**
- **lower r slab**
- **upper r slab**

Particle Loader

All kinetic particles have access to the same type of particle loader.

load density Whether to define the density of loaded particles by macroparticles or physical particles.

relative density Use a function defining the density of loaded particles in macroparticles. With constant weight particles this is a value between 0 and 1 that will be multiplied by the macroparticles per cell of the particles definition.

physical density Use a function defining the density of loaded particles in physical particles. With variable weight particles this value is divided by the *nominal density* of the particles definition and is used to modify the weight of each loaded macroparticle.

particle load placement Whether to use a bit-reversed or grid-defined placement of macroparticles in each cell.

- **grid** Place particles equidistant on the grid lines of the simulation. Useful for starting simulations "cold".
- **macro particles per direction** A vector describing the number of macroparticles to load per cell on each axis.
- **bit-reversed** Places particles according to a bit-reversed algorithm. Number of macroparticles loaded per cell is based on the value given in the *weight setting* of the particles definition.

load duration Whether to load the particles only at the beginning of the simulation or repeatedly.

- **initialize only** Particles will only be loaded at the beginning of the simulation.
- **repeat loading** Particles will be loaded according to the parameters below.
  - **start time** The time at which to start loading particles.
  - **stop time** The time at which to stop loading particles.
  - **load after initialization** Loading will repeat in all cells during the loading period.
  - **load upon shift** Load particles into cells brought into the simulation by a moving window.

velocity distribution The velocity components of particles as they are loaded.

u0 The velocity in the 0th dimension.

u1 The velocity in the 1st dimension.

u2 The velocity in the 2nd dimension.

volume The volume in which to load particles:

- **cartesian 3d slab**
- **cylindrical 2d slab**

Particle Loader from File Properties

All kinetic particles have access to the same type of particle loader. This loader will use a file with one line per particle, X,Y,Z position.

file Filename to load from. Must be located in the simulation directory.

particle shift in direction 0 A shift to apply to the 0th component of all loaded particles.

particle shift in direction 1 A shift to apply to the 1st component of all loaded particles.

particle shift in direction 2 A shift to apply to the 2nd component of all loaded particles.

density function A space time function which can modify the density of loaded particles.

load period [timesteps] If zero, particles only loaded at initialization, otherwise will load according to a specified period.

2.11.2 Fluids

Fluids Properties

kind (not editable): Neutral Fluid

fluid temperature (K): The temperature of the neutral fluid in Kelvin.

molecule: Molecule of the neutral particle. A custom molecule is available for those not pre-defined.

- **H**: Hydrogen (atomic)
- **H2**: Hydrogen (molecular)
- **He**: Helium
- **Ar**: Argon
- **Xe**: Xenon
- **Rn**: Radon
- **Kr**: Krypton
- **O**: Oxygen (atomic)
- **O2**: Oxygen (molecular)
- **Ne**: Neon
- **N**: Nitrogen (atomic)
- **N2**: Nitrogen (molecular)
- **custom molecule**: Set the mass (in amu) and ionization energy (in eV) of a custom gas.

- **mass [amu]**: The mass of a single real particle in atomic mass units.
- **ionization energy [eV]**: The ionization energy of the molecule in electron volts.

volume: This volume value will change depending on your dimensionality and coordinate system.

xMin: The minimum x position of the background fluid.

xMax: The maximum x position of the background fluid.

yMin: The minimum y position of the background fluid.

yMax: The maximum y position of the background fluid.

zMin: The minimum z position of the background fluid.

zMax: The maximum z position of the background fluid.

fluid density: Whether to use a constant or variable fluid density. A constant density will not change due to reactions with particles, while a variable will.

- **constant fluid density**: A constant fluid density is used to keep the fluid density constant after reactions with particles.
- **density**: The value of the density of the neutral fluid per cubic meter.
- **variable fluid density**: A variable fluid density is used to allow the fluid density to vary due to reactions with particles.
- **density**: The value of the density of the neutral fluid per cubic meter.

2.12 Collisions

2.12.1 Reactions Framework

The Reactions Framework is new in VSim 9. It surpasses previous interaction frameworks in both speed and flexibility in types of processes that can be added to simulations. See *Reactions Text-Setup Introduction*, as well as relevant sections in the VSim User Guide for more general information on the Reaction framework.

The Reaction framework collisions are available when `particles` in the vsimcomposer-basic-settings element is set to `include particles` and the `collision framework` is set to `reactions`. This will add a "Reactions" element to appear within the "Particle Dynamics" element.

When a collision process is added to a simulation, the user must specify each of the products and reactants from the drop down menus corresponding to each species in the chemical formula. Additionally, users must add a cross-sections to determine the reaction probability as to determine how many particles to react with in each cell for each timestep. Scroll to the bottom of this page for the reference on setting cross-sections.

Particle Particle Collisions

These collisions are for interactions between kinetically modeled particle species (for a process involving a background gas use the "Particle Fluid Collisions"). The following interactions are available by right-clicking the "Particle Particle Collisions" element and hovering the mouse pointer over the "Add CollisionType" menu.

- **Charge Exchange** A collision of the form $A + B^+ \rightarrow A^+ + B$. This is the implementation of *Charge Exchange* in the visual setup.

  energyLoss The maximum energy lost during a single inelastic collision in eV. A choice of 0 (default) will give an elastic collision.

- **`Impact Ionization`** A collision of the form $A + B \rightarrow A^{+} + B + e$. This is the implementation of *Impact Ionization* in the visual setup. If one of the reactant species is an electron, the *Electron Ionization* `productGenerator` is used. Ionization energies are taken from the appropriate species blocks.

  Note: If an electron is involved in an ionization process, the "product distribution" attribute won't affect the simulation. The scattering distribution is determined automatically by a physical model.

- **`Elastic`** A collision of the form $A + B \rightarrow A + B$. This is the implementation of *Binary Elastic* in the visual setup.

  `energyLoss` The maximum energy lost during a single inelastic collision in eV. A choice of 0 (default) will give an elastic collision.

- **`Dissociative Double Ionization`** A collision of the form $AB + C \rightarrow A^{+} + B^{+} + C + 2e$. This is the implementation of *Dissociative Ionization* in the visual setup.

  `dissociation energy` The energy required to dissociate the molecule, in eV. This value also sets a threshold for whether or not the reaction will occur. So a pair of particle will need at least this much relative energy (i.e. energy in center of momentum frame) in order to react. This can be set as a negative value to have the products gain kinetic energy.

- **`Dissociative Single Ionization``** A collision of the form $AB + C \rightarrow A^{+} + B + C + e$. This is the implementation of *Dissociative Ionization* in the visual setup.

  `dissociation energy` The energy required to dissociate the molecule, in eV. This value also sets a threshold for whether or not the reaction will occur. So a pair of particle will need at least this much relative energy (i.e. energy in center of momentum frame) in order to react. This can be set as a negative value to have the products gain kinetic energy.

- **`Dissociative Recombination`** A collision of the form $AB^{+} + e \rightarrow A + B$. This is the implementation of *Dissociative Recombination* in the visual setup.

  `dissociation energy` The threshold energy for the reaction in eV. So, a pair of reactants will need at least this much relative energy (i.e. energy in center of momentum frame) in order to react. If the reaction occurs then this much energy is lost from the products to potential energy. If this is zero, then energy will not be lost. If negative, then the products will gain kinetic energy.

- **`General Binary Reaction`** A collision of the form $A + B \rightarrow C + D$. This is the implementation of *Binary Reaction* in the visual setup.

  `threshold energy` The threshold energy for the reaction in eV. So, a pair of reactants will need at least this much relative energy (i.e. energy in center of momentum frame) in order to react. If the reaction occurs then this much energy is lost from the products to potential energy. If this is zero, then energy will not be lost. If negative, then the products will gain kinetic energy.

- **`Electron Impact Dissociation`** A collision of the form $AB + e \rightarrow A + B + e$. This is the implementation of *Electron Impact Dissociation* in the visual setup.

  `dissociation energy` The energy required to dissociate the molecule in eV.

- **`Electron Attachment`** A collision of the form $A + e \rightarrow A^{-}$. This is the implementation of *Electron Attachment* in the visual setup.

  `binding energy` The threshold energy for the reaction in eV. So, a pair of reactants will need at least this much relative energy (ie energy in center of momentum frame) in order to react. If the reaction occurs then this much energy is lost.

- **`Negative Ion Detachment`** A collision of the form $A^{-} + B \rightarrow A + B + e$. The neutral reactant must be a kinetically modeled species. This is the implementation of *Negative Ion Detachment* in the visual setup.

detachment energy The threshold energy for the reaction in eV. So, a pair of reactants will need at least this much relative energy (ie energy in center of momentum frame) in order to react. If the reaction occurs then this much energy is lost.

- **Excitation** A collision of the form $A + B \rightarrow A^* + B$. This is the implementation of *Binary Excitation* in the visual setup.

  excitation energy The potential energy, in eV, gained by the excited species (a positive value for this attribute will result in an energy loss from the simulation). This is also a threshold energy. The reaction will not occur between reactant that have a center of mass energy less than this value.

- **Inelastic Electron Scattering** A collision of the form $e + A \rightarrow e + A$. This is the implementation of *Electron Scatter* in the visual setup.

  scatter type

  - **VahediSurendra** Use the Vahedi-Surendra model to determine the scattering distrubution. See *Electron Scatter* for more information.
  - **isotropic** Use an isotropic scattering distribution.

- **Recombination** A collision of the form $A^+ + e \rightarrow A$. This is the implementation of *Binary Recombination* in the visual setup, for the specific case of electron recombination.

Particle Fluid Collisions

These collisions are for interactions between kinetically modeled particle species and a background gas. The following interactions are available by right-clicking the “Particle Fluid Collisions” element and hovering the mouse pointer over the “Add CollisionType” menu.

- **Elastic** A collision of the form $A + B \rightarrow A + B$. This is the implementation of *Binary Elastic* in the visual setup.

  energyLoss The user can choose either `fully elastic` which conserves kinetic energy, or `partially elastic` in which case the user

- **Charge Exchange** A collision of the form $A + B^+ \rightarrow A^+ + B$. This is the implementation of *Charge Exchange* in the visual setup.

  energyLoss The maximum energy lost during a single inelastic collision in eV. A choice of 0 (default) will give an elastic collision.

- **Impact Excitation** A collision of the form $A + B \rightarrow A^* + B$. This is the implementation of *Binary Excitation* in the visual setup.

  excitation energy The potential energy, in eV, gained by the excited species (a positive value for this attribute will result in an energy loss from the simulation). This is also a threshold energy. The reaction will not occur between reactant that have a center of mass energy less than this value.

- **Impact Ionization** A collision of the form $A + B \rightarrow A^+ + B + e$. This is the implementation of *Impact Ionization* in the visual setup. If one of the reactant species is an electron, the *Electron Ionization* `productGenerator` is used. Ionization energies are taken from the appropriate species blocks.

  Note: If an electron is involved in an ionization process, the “product distribution” attribute won’t affect the simulation. The scattering distribution is determined automatically by a physical model.

- **Electron Attachment** A collision of the form $A + e \rightarrow A^-$. This is the implementation of *Electron Attachment* in the visual setup.

binding energy The threshold energy for the reaction in eV. So, a pair of reactants will need at least this much relative energy (ie energy in center of momentum frame) in order to react. If the reaction occurs then this much energy is lost.

- **Negative Ion Detachment** A collision of the form $A^{-} + B \rightarrow A + B + e$. The neutral reactant must be a kinetically modeled species. This is the implementation of *Negative Ion Detachment* in the visual setup.

  detachment energy The threshold energy for the reaction in eV. So, a pair of reactants will need at least this much relative energy (ie energy in center of momentum frame) in order to react. If the reaction occurs then this much energy is lost.

- **Inelastic Scattering** A collision of the form $e + A \rightarrow e + A$. This is the implementation of *Electron Scatter* in the visual setup.

  scatter type

  - **VahediSurendra** Use the Vahedi-Surendra model to determine the scattering distrubution. See *Electron Scatter* for more information.
  - **isotropic** Use an isotropic scattering distribution.

- **General Binary Reaction** A collision of the form $A + B \rightarrow C + D$. This is the implementation of *Binary Reaction* in the visual setup.

  threshold energy The threshold energy for the reaction in eV. So, a pair of reactants will need at least this much relative energy (i.e. energy in center of momentum frame) in order to react. If the reaction occurs then this much energy is lost from the products to potential energy. If this is zero, then energy will not be lost. If negative, then the products will gain kinetic energy.

- **Dissociative Double Ionization** A collision of the form $AB + C \rightarrow A^{+} + B^{+} + C + 2e$. This is the implementation of *Dissociative Ionization* in the visual setup.

  dissociation energy The energy required to dissociate the molecule, in eV. This value also sets a threshold for whether or not the reaction will occur. So a pair of particle will need at least this much relative energy (i.e. energy in center of momentum frame) in order to react. This can be set as a negative value to have the products gain kinetic energy.

- **Dissociative Single Ionization`** A collision of the form $AB + C \rightarrow A^{+} + B + C + e$. This is the implementation of *Dissociative Ionization* in the visual setup.

  dissociation energy The energy required to dissociate the molecule, in eV. This value also sets a threshold for whether or not the reaction will occur. So a pair of particle will need at least this much relative energy (i.e. energy in center of momentum frame) in order to react. This can be set as a negative value to have the products gain kinetic energy.

Three Body Reactions

- **Three Body Recombination** A collision of the form $A^{+} + e + B \rightarrow A + B$. This is the implementation of *Three Body Recombination* in the visual setup.

Field Ionization Process

The "Fluid Field" and "Particle Field" ionization processes both use *Field Ionization* `productGenerator`. The distinction is made in the visual setup for benefit of the user.

- **Particle Field Ionization** This is the implementation of *Field Ionization* for ionization of kinetically modeled particles by electric fields in the visual setup.

  cross section type

  - **DCADK** Use this when the time step resolves the oscillations of Electric field. See *Field Ionization DCADK* for more information.
  - **Average ADK** Use this when the time step is much larger than the oscillations of Electric field. See *Field Ionization Average ADK* for more information.

- **`Fluid Field Ionization`** This is the implementation of *Field Ionization* for ionization of background gases by electric fields in the visual setup.

  `cross section type`

  - **DCADK** Use this when the time step resolves the oscillations of Electric field. See *Field Ionization DCADK* for more information.
  - **Average ADK** Use this when the time step is much larger than the oscillations of Electric field. See *Field Ionization Average ADK* for more information.

Decay Process

- **`Decay`** This is the implementation of *Decay* in the visual setup

  `lifetime` The lifetime (in seconds) of the unstable species.

Cross-Sections

Interpolated from 2Column Data

Import cross sections from a data file with the independent variable (either velocity or energy) in the first column and the cross-section (dependent variable) in the second column. The imported file should *NOT* have any header.

`cross section variable` The unit of the independent variable in the first column of the data file. Either `velocity`, or `energy`.

`cross section data file` The name of the cross section data file. This file must be in the run directory.

Constant Cross Section

This can be used to set a user defined function for the cross section.

`constant` A constant value (in m^2) for the cross section.

Power Law Cross Section

Use a cross section of the form Ax^E. See *Power Law Cross-Section* for more information.

`coefficient` The value for the constant *A* in the expression above.

`exponent` The value for the constant *E* in the expression above.

`cross section variable` The unit of the independent variable (x in the expression above). Either `velocity`, or `energy`.

Exponential Polynomial Cross Section

The cross-sections will take the functional from of the equation:

$$\sigma(y) = \frac{1}{y}\left(A\ln(y) + \frac{B\ln(y) + C*(y - "toZero")}{y + D}\right)$$

where

$$y = x/I$$

See *Exponential Polynomial Cross-Section* for more information.

Acoeff The value for the fitting constant *A* in the formula above.

Bcoeff The value for the fitting constant *B* in the formula above.

Ccoeff The value for the fitting constant *C* in the formula above.

Dcoeff The value for the fitting constant *D* in the formula above.

Icoeff The value for the fitting constant *I* in the formula above. A scaling factor for the independent variable, x.

toZero Sets the value for the `toZero` parameter in the formula above. Enter either “0” to have the cross-section go to zero at threshold, or “1” to have the cross-section go to a non-zero value.

cross section variable Set the independent variable, x, in the equation above. Options are:

- **velocity**
- **energy**

Additional Collision Attributes

update periodicity This sets how frequently the reaction is carried out. See the description of `updatePeriod` in rxns-process-Setting for more information. The *update periodicity* setting in the Visual Setup will set the `updatePeriod` when the `.sdf` is translated to a `.in` file. Options are:

every timestep The reaction update will occur every timestep.

update period *not editable; defaulted to 1*

custom Set a custom update periodicity.

update period Enter a constant to set the update periodicity.

product distribution This will apply an anisotropy to the cross-section. See the section *Anisotropy* for more information. This attribute is only available for collisions with two reactant particles (or fluids) and two product particles (or fluids) except for the Inelastic Electron Scattering process which as the anisotropy set according to the Vahedi-Surendra model.

isotropic distribution The default. Set no preference for forward or backward scattering.

anisotropic distribution Add a preference for forward or backward scattering.

anisotropy Set a value between -1 and 1 (inclusive). A value of -1 will give full backward scattering bias, and 1 will give full forward scattering bias.

2.12.2 Reduced Collisions

The "reduced" collisions framework is an implementation of the *ImpactCollider* framework in the Visual Setup. Processes in this framework are limited to kinetic particles interacting via elastic, excitation, ionization, and a set of charge exchange and momentum exchange collisions with a background gas species.

The "reduced" collisions are available when `particles` in the vsimcomposer-basic-settings element is set to `include particles` and the `collision framework` is set to `reduced`. This will add a "ReducedCollisions" element to appear with in the "Particle Dynamics" element.

Electron Neutral Fluid Collisions

To include a collision process, right-click on the *Electron Neutral Fluid Collision* in the element tree and choose *Add CollisionProcess* to choose one of either *Elastic*, *Excitation*, or *Ionization*.

`neutral gas` Select the neutral gas to use in the collision. The drop down will automatically populate with available pre-defined *BackgroundGases* elements.

`impact particles` Select the electron particles to use in the collision. The drop down will automatically populate with available pre-defined *KineticParticles* of kind = Electrons.

Elastic

`kind` (not editable) Elastic

`cross section type` The cross section for the collision must be specified. You can choose whether to use *built in* cross sections, cross sections from the *Evaluated Electron Data Library (EEDL)* or by specifying your own *user defined cross sections* in a text file.

- **`user defined cross section`** User provided cross-section data provided in a text file. For more information on the format of the file, please see *OAFunc Cross-Section Interface*.

  `cross section data file` The name of the file containing the cross-section data.

- **`product distribution`** The type of scattering.
  - **`isotropic`**
  - **`Vahedi-Surendra`**

Excitation

`kind` (not editable) Excitation

`cross section type` The cross section for the collision must be specified. You can choose whether to use *built in* cross sections, cross sections from the *Evaluated Electron Data Library (EEDL)* or by specifying your own *user defined cross sections* in a text file.

- **`user defined cross section`** User provided cross-section data provided in a text file. For more information on the format of the file, please see *OAFunc Cross-Section Interface*.

  `cross section data file` The name of the file containing the cross-section data.

- **`product distribution`** The type of scattering:
  - **`isotropic`**
  - **`Vahedi-Surendra`**

Ionization

`kind` (not editable) Ionization

`product electrons` The resultant electrons from the impact ionization collision.

`product ions` The resultant ions from the impact ionization collision.

`cross section type` The cross section for the collision must be specified. You can choose whether to use *built in* cross sections, cross sections from the *Evaluated Electron Data Library (EEDL)* or by specifying your own *user defined cross sections* in a text file.

- **`user defined cross section`** User provided cross-section data provided in a text file. For more information on the format of the file, please see *OAFunc Cross-Section Interface*

  `cross section data file` The name of the file containing the cross-section data.

- **`product distribution`** The type of scattering:
  - **`isotropic`**
  - **`Vahedi-Surendra`**

Ion Neutral Fluid Collisions

To include a collision process, right-click on the *Ion Neutral Fluid Collision* in the element tree and choose *Add CollisionProcess* to choose one of either *Charge Exchange* or *Momentum Exchange*.

`neutral gas` Select the neutral gas to use in the collision. The drop down will automatically populate with available pre-defined *BackgroundGases* elements.

`impact particles` Select the charged particles to use in the collision. The drop down will automatically populate with available pre-defined *KineticParticles* of kind = Charged Particles.

Charge Exchange

`kind` (not editable) Charge Exchange

`cross section type` The cross section for the collision must be specified. You can choose whether to use *built in* cross sections, cross sections from the *Evaluated Electron Data Library (EEDL)* or by specifying your own *user defined cross sections* in a text file.

- **`built in`** The built-in collisions utilize existing internal collision cross-section data.
- **`user defined cross section`** User provided cross-section data provided in a text file. For more information on the format of the file, please see *OAFunc Cross-Section Interface*.

  `cross section data file` The name of the file containing the cross-section data.

- **`product distribution`** The type of scattering.
  - **`backward`**

Momentum Exchange

`kind` (not editable) Momentum Exchange

cross section type The cross section for the collision must be specified. You can choose whether to use *built in* cross sections, cross sections from the *Evaluated Electron Data Library (EEDL)* or by specifying your own *user defined cross sections* in a text file.

- **built in** The built-in collisions utilize existing internal collision cross-section data.
- **user defined cross section** User provided cross-section data provided in a text file. For more information on the format of the file, please see *OAFunc Cross-Section Interface*

  cross section data file The name of the file containing the cross-section data.
- **product distribution** The type of scattering.

  isotropic

2.12.3 Monte Carlo Framework (Deprecated in VSim 9)

The particle interactions available as part if the "Monte Carlo Interactions" framework are still available, but we recommend that users update their simulations to use the new *Reactions Framework* framework. The Monte Carlo interaction will be removed in VSim 10. For more information on the Monte Carlo framework, see *Monte Carlo Interactions Introduction* as well as the section VSim User Guide: Simulation Concepts: Collisions: Monte Carlo Interactions in the VSim User Guide.

The Monte Carlo framework collisions are available when `particles` in the vsimcomposer-basic-settings element is set to `include particles` and the `collision framework` is set to `monte carlo`. This will add a "Reactions" element to appear with in the "Particle Dynamics" element.

When a collision process is added to a simulation, the user must specify each of the products and reactants from the drop down menus corresponding to each species in the chemical formula. Additionally, users must add a cross-sections to determine the reaction probability as to determine how many particles to react with in each cell in a timestep.

Particle Particle Collisions

These collisions are for interactions between kinetically modeled particle species (for a process involving a background gas use the "Particle Fluid Collisions"). The following interactions are available by right-clicking the "Particle Particle Collisions" element and hovering the mouse pointer over the "Add CollisionType" menu.

- **Charge Exchange** A collision of the form $A^+ + A \to A + A^+$. Reactants must be of the same species. This is the implementation of *binaryChargeExchange* in the visual setup.

  Note: Caution should be exercised when using binaryChargeExchange reactions in the Monte Carlo framework with variable-weight species kinds; the results may be unreliable. Consider using the newer Reactions framework instead.

- **Recombination** A collision of the form $A^+ + e \to A + \gamma$. The energy released in the form of a photon is not tracked. This is the implementation of *binaryRecombination* in the visual setup.
- **Impact Ionization** A collision of the form $A + e \to A^+ + 2e$. The neutral reactant must be a kinetically modeled species. This is the implementation of *binaryIonization* in the visual setup.
- **Excitation Loss** A collision of the form $A + e \to A^* + e$. Where A^* is an excited state of the A species. The excited species cannot be tracked, by the user specified `excitation threshold` energy will be lost from the simulation. This is the implementation of *binaryExcitation* in the visual setup.

  excitation threshold The energy to excite the `incoming particles` species (in eV). This energy is lost from the simulation if the process occurs.

- **`Elastic`** A collision of the form $A + B \to A + B$. There is an option to add an energy loss when the reaction occurs. This is the implementation of *binaryElastic* in the visual setup.

  Note: Caution should be exercised when using binaryElastic reactions in the Monte Carlo framework with variable-weight species kinds; the results may be unreliable. Consider using the newer Reactions framework instead.

  `energy loss` The energy (in eV) lost when the reaction occurs. This is also a threshold energy for the process to occur.

- **`Impact Dissociation`** A collision of the form $e + AB \to A + B + e$. There is an option to add a threshold energy. This is the implementation of *binaryDissociation* in the visual setup.

  `threshold energy` Dissociation threshold energy used to calculate the energy of the secondary electron.

Particle Fluid Collisions

These collisions are for interactions between kinetically modeled particle species and a background gas. The following interactions are available by right-clicking the “Particle Fluid Collisions” element and hovering the mouse pointer over the “Add CollisionType” menu.

- **`Elastic`** A collision of the form $A + B \to A + B$. This is the implementation of *impactElastic* in the visual setup.
- **`Charge Exchange`** A collision of the form $A^+ + A \to A + A^+$. The neutral reactant must be a fluid species and the same species as the incoming ion. This is the implementation of *chargeExchange* in the visual setup.
- **`Impact Excitation`** This is the implementation of *impactExcitation* in the visual setup.
- **`Impact Ionization`** This is the implementation of *impactIonization* in the visual setup.
- **`Electron Attachment`** This is the implementation of *electronAttachment* in the visual setup.
- **`Negative Ion Detachment`** This is the implementation of *negativeIonDetachment* in the visual setup.
- **`Inelastic Scattering`** This is the implementation of *nullBgAbsorber* in the visual setup. Follow the link in the previous sentence for a description of the `material mass`, `electron temperature fluid`, `radiation length`, `atomic ratio`, `multiple scattering model`, and energy straggling‘‘ parameters.

Three Body Reactions

These collisions are for interactions between kinetically modeled particle species (for a process involving a background gas use the “Particle Fluid Collisions”). The following interactions are available by right-clicking the “Particle Particle Collisions” element and hovering the mouse pointer over the “Add CollisionType” menu.

- **`Three Body Recombination`** This is the implementation of *threeBodyRecombination* in the visual setup. Follow the link for a description of the `thermal velocity neutrals`, `alpha`, and `tempExpFactor` parameters.

Field Ionization Process

- **`Particle Field Ionization`** This is the implementation of *fieldIonization* in the visual setup. Follow the link for a description of the `DCADK` and `Average ADK` cross section types.

- **`Fluid Field Ionization`** This is the implementation of *nullFieldIonization* in the visual setup. Follow the link for a description of the `DCADK` and `Average ADK` cross section types.

Decay Process

- **`Decay`** This is the implementation of *oneBodyDecay/oneBodyVADecay* in the visual setup

  `lifetime` The lifetime (in seconds) of the unstable species.

Cross-Sections

Interpolated from 2Column Data

Import cross sections from a data file with the independent variable (either velocity or energy) in the first column and the cross-section (dependent variable) in the second column.

`cross section variable` The unit of the independent variable in the first column of the data file. Either `velocity`, or `energy`.

`cross section data file` The name of the cross section data file. This file must be in the run directory.

function

This can be used to set a user defined function for the cross section.

`cross section variable` The unit of the independent variable in the expression for the cross section. Either `velocity`, or `energy`.

`expression` The expression for the cross-section. Can be set to a constant, parameter, or SpaceTimeFunction.

LXcatFile

Use cross sections with LXCat headers. The unique headers before each set of two-column cross section data sets allows. See `https://fr.lxcat.net/data/set_type.php` to obtain cross sections.

`cross section data file` The name of the cross section data file. This file must be in the run directory.

`cross section variable` The unit of the independent variable in the first column of the data file. Either `velocity`, or `energy`.

`process` A string from the LXCat header that will determine which set of cross sections from with in the `cross section data file` to use for the interaction. Enter the text that follows the "PROCESS:" line in the header file.

Evaluated Electron Data Library (eedl)

This option is only available for some reactions. The eedl contains cross-sections for many processes involving electrons. An electron must be involved in the collision to use this set of cross-sections. The eedl.dat file (along with many others) is installed with VSim. On Windows, it is installed into `C:\Program Files\Tech-X\VSim-9\Contents\engine\share\data`. This data file must be copied into the run directory. The inputs from the user are matched with the headers in the eedl file to determine which cross-sections to use.

cross section data file The name of the file containing the eedl headers and cross-sections. Must be copied into run directory.

2.13 Histories

Histories provide data from each time step of a simulation. They can provide useful diagnostics to make sure your simulation is proceeding as intended. Some histories are only available with certain simulation setups (e.g. only available in electromagnetic simulation, or only available in simulations with particles).

To add a history, right-click the "Histories" element of the setup tree then navigate to the history to be added to the simulation

2.13.1 Array History

An *Array History* will output an array of data for each time-step.

Field Slab Data Store the value of a field at every timestep within a specified 3D volume.

kind (not editable) Field Slab Data

description A comment to describe the history.

field Choose the field to record. Options for electromagnetic simulations are:

- **Electric Field**
- **Magnetic Field**

Options for electromagnetic simulations are:

- **Phi**
- **Charge Density**
- **Electric Field**

volume The volume inside of which to collect the field data.

- **cartesian 3d slab**
  - **xMin** The minimum x position of the box.
  - **xMax** The maximum x position of the box.
  - **yMin** The minimum y position of the box.
  - **yMax** The maximum y position of the box.
  - **zMin** The minimum z position of the box.
  - **zMax** The maximum z position of the box.

Particle Momentum Calculate the total momentum for a particular set of particles in the whole simulation domain. All three components of the momentum are recorded. Thus, for some simulations in 1D or 2D, some components of the momentum may always be zero.

kind (not editable) Particle Momentum

particles Select any of the previously defined *KineticParticles* in your simulation.

Far-Field Observation **ONLY AVAILABLE IN ELECTROMAGNETIC SIMULATIONS**

kind (not editable) Far-Field Observation `time delay` The simulation time at which the history begins recording.

`duration` The simulation time length during which the history records.

`number of time points` The number of time points.

`Kirchhoff sphere radius` The radius of the Kirchhoff sphere.

`number of polar angles` The number of polar angles represented on the Kirchhoff sphere.

`number of azimuthal angles` The number of azimuthal angles represented on the Kirchhoff shpere.

`number of line integral segments` The number of integration points around the far-field sphere.

`Kirchhoff sphere center` The center of the Kirchhoff sphere.

`Far-Field Box Data` **ONLY AVAILABLE IN ELECTROMAGNETIC SIMULATIONS**

`kind` **(not editable)** Far-Field Box Data

`start time` The simulation time at which the history begins recording.

`end time` The simulation time at which the history ends recording.

`volume` The volume to use for the box.

- `cartesian 3d slab`
  - `xMin` The minimum x position of the box.
  - `xMax` The maximum x position of the box.
  - `yMin` The minimum y position of the box.
  - `yMax` The maximum y position of the box.
  - `zMin` The minimum z position of the box.
  - `zMax` The maximum z position of the box.

2.13.2 Combo History

Combo Histories are used to do operations on other histories. The operation is done at every time step and the resulting values are recorded as a new history. The output will be a 1D array of the value vs time. Beware of dividing by zero.

`Binary Combination History` A Binary Combination History can be used to combine any two *Field* or *Particle* histories.

`kind` **(not editable)** Binary Combination History

`history 1` Select one previously defined *Field* or *Particle* History.

`history 2` Select one previously defined *Field* or *Particle* History.

`history1 coefficient` A multiplying factor on the first history.

`history2 coefficient` A multiplying factor on the second history.

`combination` Which operation to use for combining the histories.

- `add` (history1coefficient*history 1) + (history2 coefficient*history 2)
- `subtract` (history1 coefficient*history 1) - (history2 coefficient*history 2)
- `multiply` (history1 coefficient*history 1) * (history2 coefficient*history 2)
- `divide` (history1 coefficient*history 1) / (history2 coefficient*history 2)

2.13.3 Field History

Field Histories record on a per time-step basis. Field histories are used to measure quantities such as the value or energy of the field at a location. The output will be a 1D array of the value vs time.

`Accelerating Voltage` This history creates a test electron and measures the accelerating voltage received by an electron traveling at a fixed velocity across a gap in a cavity structure. See *acceleratingVoltage* for a reference defining ‘acclerating voltage’.

`kind` **(not editable)** Accelerating Voltage

`description` A comment to describe the history.

`start coordinate 0` The starting position of the test electron in the x-direction in Cartesian simulations (or z-direction in cylindrical simulations).

`start coordinate 1` The starting position of the test electron in the y-direction in Cartesian simulations (or r-direction in cylindrical simulations).

`start coordinate 2` The starting position of the test electron in the z-direction in Cartesian simulations (or phi-direction in cylindrical simulations).

`end coordinate 0` The starting position of the test electron in the x-direction in Cartesian simulations (or z-direction in cylindrical simulations).

`end coordinate 1` The starting position of the test electron in the y-direction in Cartesian simulations (or r-direction in cylindrical simulations).

`end coordinate 2` The starting position of the test electron in the z-direction in Cartesian simulations (or phi-direction in cylindrical simulations).

`velocity` The velocity of the test electron. By default this is the speed of light.

`Electric Field Energy` Calculate the total energy of the electric field in the specified volume.

`kind` **(not editable)** Electric Field Energy

`volume` The region over which to calculate the field energy.

- **`simulation region`** Use the entire simulation domain.
- **`index 3d slab`** A user-defined volume based on cell indices.

  `lower indices` The lower indices of the volume.

  `upper indices` The upper indices of the volume.

`EM Field Energy` Calculate the total energy of the electromagnetic field in the specified volume. Only available in electromagnetic simulations.

`kind` **(not editable)** EM Field Energy

`volume` The region over which to calculate the field energy.

- **`simulation region`** Use the entire simulation domain.
- **`index 3d slab`** A user-defined volume based on cell indices.

  `lower indices` The lower indices of the volume.

  `upper indices` The upper indices of the volume.

  `shape` A volume based on a previously defined geometry.
- **`object name`** Select from a previously defined geometry.

`Magnetic Field Energy` Calculate the total energy of the magnetic field in the specified volume.

kind (not editable) Magnetic Field Energy

volume The region over which to calculate the field energy.

- **simulation region** Use the entire simulation domain.
- **index 3d slab** A user-defined volume based on cell indices.

  lower indices The lower indices of the volume.

  upper indices The upper indices of the volume.

Field at Position Record the specified field at the specified coordinates. All components of the field are recorded into an array.

kind (not editable) Field at Position

field Select the desired field.

coordinate0 The position coordinate in the 0th dimension, x in cartesian coordinates, z in cylindrical.

coordinate1 The position coordinate in the 1st dimension, y in cartesian coordinates, r in cylindrical.

coordinate2 The position coordinate in the 2nd dimension, z in cartesian coordinates.

Poynting Flux **ONLY AVAILABLE IN ELECTROMAGNETIC SIMULATIONS**

Calculates the integrated Poynting vector (energy flux) through the area defined by the min and max values.

kind (not editable) Poynting Vector

surface The plane to use.

- **yz**

  offset The x offset from zero, in meters.

  yMin The location of the y minimum, in meters.

  yMax The location of the y maximum, in meters.

  zMin The location of the z minimum, in meters.

  zMax The location of the z maximum, in meters.

- **xz**

  offset The y offset from zero, in meters.

  xMin The location of the x minimum, in meters.

  xMax The location of the x maximum, in meters.

  zMin The location of the z minimum, in meters.

  zMax The location of the z maximum, in meters.

- **xy**

  offset The z offset from zero, in meters.

  xMin The location of the x minimum, in meters.

  xMax The location of the x maximum, in meters.

  yMin The location of the y minimum, in meters.

  yMax The location of the y maximum, in meters.

Pseudo-potential **This option is deprecated. Use ‘Pseudo-potential at Coordinates’ or ‘pseudo-potential at Indices’ instead.**

kind (not editable) Pseudo-potential

start indices The indices of the cells for the starting location.

end indices The indices of the cells for the ending location.

Pseudo-potential at Coordinates Calculates the pseudo-potential difference, in Volts, between two points. The start point would correspond to the measure point, while the end point would correspond to the reference.

kind (not editable) Pseudo-potential

start coordinate 0 The coordinate of the start point in the 0th dimension.

start coordinate 1 The coordinate of the start point in the 1st dimension.

start coordinate 2 The coordinate of the start point in the 2nd dimension.

end coordinate 0 The coordinate of the end point in the 0th dimension.

end coordinate 1 The coordinate of the end point in the 1st dimension.

end coordinate 2 The coordinate of the end point in the 2nd dimension.

Pseudo-potential at Indices Calculates the pseudo-potential difference, in Volts, between two points, specified by grid index.

kind (not editable) Pseudo-potential

description A description of the potential difference.

start indices The indices of the cells for the starting location.

end indices The indices of the cells for the ending location.

2.13.4 Log History

A *Log History* will record data on a per-event basis rather than at each time step.

Absorbed Particle Log Record information about each and every particle that strikes a chosen absorbing surface. The output will be a 1D array of the value.

kind (not editable) Absorbed Particle Log

particle absorber Select the previously defined particle absorbing boundary condition. This must be a ParticleBoundaryCondition that can *Save*.

particle quantity What information about the particle is to be recorded. For vector-like quantities (position, velocity, and weight), you must select which component of the vector you wish to record in the *component* option (0 –> x, 1 –> y, 2 –> z).

particle time The time the particle strikes the absorber.

particle position The position of the particle when it is absorbed.

particle velocity The velocity of the particle when it is absorbed.

particle weight The weight of the particle when it is absorbed.

particle energy The total energy of all the particles that are absorbed.

particle current The total current of all the particles that are absorbed.

absorbed particle mass The mass of the particle absorbed in each time step.

2.13.5 Particle History

Particle Histories record on a per time-step basis. Particle histories are used to measure quantities such as the total number of particles in a simulation at each step, or the current absorbed at chosen absorbing surface at each step. The output will be a 1D array of the value vs time.

`Absorbed Particle Current` Calculates the absorbed current on a specified particle absorber.

> **`kind`** **(not editable)** Absorbed Particle Current
>
> **`particle absorber`** Select the previously defined particle absorbing boundary condition. This must be a ParticleBoundaryCondition that can *Save*.

`Absorbed Particle Power` Calculates the power absorbed on a specified particle absorber.

> **`kind`** **(not editable)** Absorbed Particle Power
>
> **`particle absorber`** Select the previously defined particle absorbing boundary condition. This must be a ParticleBoundaryCondition that can *Save* particle data.

`Emitted Current` Records the emitted current from the specified particle emitter

> **`kind`** **(not editable)** Emitted Current
>
> **`particle emitter`** Select the previously defined particle emitting boundary condition.

`Number of Macroparticles` Calculates the total number of macroparticles in the simulation domain for the specified *KineticParticle*.

> **`kind`** **(not editable)** Number of Macroparticles
>
> **`particles`** Select the name of the previously defined *KineticParticles*.

`Number of Physical Particles` Calculates the total number of real particles in the simulation domain for the specified *KineticParticle*.

> > **`kind`** **(not editable)** Number of Physical Particles
> >
> > **`particles`** Select the name of the previously defined *KineticParticles*.

`Particle Energy` Calculates the total energy in the simulation domain for the specified *KineticParticle*.

> **`kind`** **(not editable)** Particle Energy
>
> **`particles`** Select the name of the previously defined *KineticParticles*.

`Particle Energy Change from Boundary` Calculates the energy change in a particle species due to a diffuse reflector boundary condition.

> **`kind`** **(not editable)** Particle Energy Change from Boundary
>
> **`particle absorber`** Select the name of the boundary diffuse reflector particle boundary condition.

CHAPTER

THREE

TEXT SETUP

3.1 Global Variables

3.1.1 Global Variables

floattype (*string*)
Variable that defines the precision of real numbers in the simulation. For greater precision, use the value `double`, otherwise use the option `float`. You must define *floattype* in an input file.

dimension (*integer*)
Variable that defines the dimensionality (1D, 2D, or 3D) of the simulation. The variable dimension indicates how many dimensions this simulation will use. The `-dim` command line parameter overrides this variable. You must define *dimension* in an input file or on the command line.

dt (*real*)
Step size to use in the simulation. When choosing the step size for your simulation, you must consider stability requirements. For example, for electromagnetic simulations, you should specify a step size that satisfies the Courant condition. You can override the *dt* variable using the `-dt` command line option. You must define *dt* in an input file.

nsteps (*integer*)
Number of steps to take. (In the case of a restart, *nsteps* is the number of additional steps.) This can be overridden with the `-n` command line parameter. You must define *nsteps* in an input file or on the command line.

dumpPeriodicity (*integer*)
How often to dump the data; indicates data is to be dumped whenever the time step has increased by this amount. The command line parameter -d overrides this variable. Must have this defined in an input file or on the command line.

dumpSteps (*integer*)
An alternative to dumpPeriodicity allowing an irregular specification of times steps in which to dump. Steps should be in increasing order, but do not need to be evenly spaced. If the simulation runs successfully, or is ended by clicking "Dump and Stop" a final dump will be written so that the simulation can be restarted from the final dump. If the simulation ends with a crash, dumpSteps will not write a final dump, and it will not be possible to restart the simulation from a previous dump.

This option can also be applied to individual field blocks to set the steps at which the particular field is dumped. If dumpSteps is used in this way, care must be taken when restarting the simulation from a previous dump.

dumpSteps (*Expression*)
Alternative to dumpPeriodicity in which an expression block of name dumpSteps is defined. Can be used to provide an expression for dumpsteps. For example:

```
<Expression dumpSteps>
  expression =  or(mod(n+1, VPMW_DUMP_PERIOD) == 1 , mod(n-1, VPMW_DUMP_PERIOD)␣
↪== 1 ), mod(n, VPMW_DUMP_PERIOD) == 1)
</Expression>
```

Will dump at the specified VPMW_DUMP_PERIOD, as well as at timestep = VPMW_DUMP_PERIOD + 1 and VPMW_DUMP_PERIOD + 2

verbosity (*integer*)
Specifies the amount of text output from a simulation run. Increasing the value of *verbosity* may aid in debugging a simulation. The value can be $2^N - 1$, with $0 <= N <= 10$. The values, $1 <= N <= 4$ are for severe problems that will stop the simulation. The values, $N >= 8$ are for increasing levels of debug information. The other levels are:

- `verbosity=0`, $N = 0$, *NONE*, no messages output.
- `verbosity=31`, $N = 5$, *WARNING*, include warnings in the output, which should be examined by the user in order to fix any errors that could be affecting the simulation.
- `verbosity=63`, $N = 6$, *NOTICE*, include notices in the output, which are not necessarily issues that need to be fixed, but should still be examined by the user.
- `verbosity=127`, $N = 7$, *INFO*, normal output for informing the user of simulation progress.

randomSeeding (*string*, *default seedDeterministicallyByDefault*)
Specifies how Vorpal seeds the (pseudo) random number generators use, e.g., to determine initial particle velocities or whether a Monte Carlo collision occurs. The recommended option, `seedDeterministicallyByDefault` allows different random number generators to be used for different input blocks, with the seeds chosen deterministically based on the input block (fully qualified…i.e., including names of parent blocks) name and *randomSeed*. Another option, `seedRandomlyByDefault` generates seeds from a separate random number generator, which is itself seeded according to the current time; if run twice (with enough time separation to be resolved by the internal timer), a simulation will use different random number sequences. For these options, each MPI rank in a parallel simulation uses a different seed. If random seeds are explicitly specified in an input file block, the specified seeds are used. The `seedGlobalRng` option (deprecated) uses the same random number generator for many (but not all) objects.

randomSeed (*integer*, *between -1 and 2147483647*, *default 1*)
A seed for generating (pseudo) random numbers. If *randomSeeding* = `seedDeterministicallyByDefault`, then changing this value will change the seeds for all random number generators used in a simulation. If this is set to -1 (the recommended option), then a *randomSeed* will be chosen based on the current time. This option has a similar effect for *randomSeeding* = `seedGlobalRng`, except it affects only the global random number generator (excluding input blocks that take `seed` or `randomSeed` options). This option has no effect for *randomSeeding* = `seedRandomlyByDefault`.

maxcellxing (*integer*)
In particle simulations in which the particles could ordinarily cross more than one cell in a time step and cause out-of-bounds memory access (such as can happen in electrostatic simulations with non-relativistic dynamics), you can impose a velocity limit to prevent the out-of-bounds memory access from occurring by using the maxcellxing variable. For example, `maxcellxing = 1` will limit the particle velocity so that the particle cannot cross more than one cell per step. That is, if for any direction i, velocity_i * dt > dx_i, then the velocity is reduced in magnitude so that velocity_i * dt<= dx_i for all i. Since imposing a velocity limit introduces error into the simulation, for cases in which you must use the velocity limit for a significant number of particles, you should lower the time step for the simulation.

sortPtcls (*boolean*, *default = true*)
Specifies whether or not you want Vorpal to sort the particles based on an algorithm that intelligently selects the

time step at which to sort. This can affect performance and may be a parameter that you want to examine on your own for its effect on performance.

`dnSortMin` (*integer*)
This parameter is used to instruct Vorpal what is the minimum number of time steps before sorting of the particles. This allows sorting to be deterministic and thus repeatable.

`dnSortMax` (*integer*)
This parameter is used to instruct Vorpal what is the maximum number of time steps before sorting of the particles. This allows sorting to be deterministic and thus repeatable.

`useGridBndryRestore` (*boolean*)
Switch to control the method for obtaining grid boundary information on restarts. The default (true) is to input boundary information from the dump files. Setting this switch to false means that the grid boundary information is re-calculated, which can be computationally more intensive.

`useHistoryRestore` (*boolean*)
The default(true), restores histories so that the dump and restart yields the same result as simply continuing simulation (also important for feedback); if false, treat history dump file as if empty, and append data only with no regard for previous data

`copyHistoryAtEachDump` (*boolean*)
The default(true), copies the History file at each dump in a HistoryOld file before appending the new history data. This is a safety measure in case the history file gets corrupted. If false, the data is simply appended to the History file, without creating a back up (or copied) file.

`stepPrintPeriodicity` (*integer*)
How often to print out the "Taking step n at clock time…" message as the simulation runs. By default, this option is 1 and the message is printed every step. If this option is 0, the message will never be printed. If this option is 10, the message will be printed for every tenth step. This option is useful for reducing the amount of output of simulations that run many time steps.

`timingAnalysisPeriodicity` (*integer*)
How often to calculate (and print) rough timing diagnostics to evaluate the speed of a simulation. This is useful for analyzing load-balancing and parallel scaling or for charactarizing the computational performance of simulations.

By default, this option is -1, and no timings are calculated; if 0, timings are printed out for the entire simulation. If a positive integer, timings will be printed out periodically. For example, if this option is 1000, the timing for each set of 1000 time-steps will be printed out (as well as a summary for the entire simulation).

The output is repeated in a single-line format that can be converted into a *.csv* (comma separated values) file using the command line utility `grep`. For example, to extract the output of *`timingAnalysisPeriodicity`* from *mySimulation.out* into a new file *timingAnalysis.csv* that can be opened by most spreadsheet software, one would execute `grep -oP '(?<=timingAnalysis:,  ).*' mySimulation.out > timingAnalysis.csv`.

Timings are calculated from wallclock-time and will be affected by other activity. The timing resolution varies from system to system. Short times (much less than one second) may be inaccurate.

Global Variables Specific To Moving Windows

The two global parameters *`downShiftDir`* and *`downShiftPos`* apply to Vorpal's moving window feature. The moving window feature allows the simulation window to move at the speed of light in the chosen direction. This feature is used to reduce the size of the simulation box while following the physics phenomenon of interest such as a laser pulse or a particle beam propagating at a velocity close to the speed of light.

downShiftDir (*integer*)
Control variable for the **moving window** option that sets the direction of the moving window. If the *downShiftDir* parameter is not specified or set to `-1` for, Vorpal does not use the moving window. Set to a number between `0` and *dimension* to cause a down shift (moving window) in that direction.

downShiftDir Parameter Values

- *null* (If no value specified, Vorpal does not use the moving window.)
- `-1` (If `-1` is specified, Vorpal does not use the moving window)
- `0` (*x* direction)
- `1` (*y* direction)
- `2` (*z* direction)

downShiftPos (*real*)
Distance at which Vorpal should activate the **moving window** feature. Vorpal calculates the time equal to this distance divided by the speed of light to determine when to turn on the moving window. The *downShiftPos* parameter may be set to a variable such as *DSHFTPOS*, which has been assigned a real number value such as 0.95**LX*, where *LX* is the length of the grid along the *x* axis.

OAFunc shiftSpeed (block)
Function (of one variable) that returns the velocity of the moving window (in m.s $^{-1}$) as a function of time. When using `kind = expression` in an *OAFunc* block `x` is the default variable, to use `t` instead, specify `variable = t` in the *OAFunc* block [see *expression (OAFunc)*]. You can still use `x` but be aware that `x` represents the time in this case.

Example Moving Window Code

```
# Moving window
# Shift along direction 0
downShiftDir   = 0
# Start shifting at t = DSHFTPOS/(speed of light)
downShiftPos   = DSHFTPOS

<OAFunc shiftSpeed>
   kind = expression
   expression = LIGHTSPEED *(BETA - ALPHA *t)
   variable = t
</OAFunc>
```

3.2 Grid

3.2.1 Grid

Determines the simulation size and relationship of physical coordinates to cell indices.

Every object in Vorpal interacts with a grid defined by the Grid block. If there is more than one Grid block in the input file, Vorpal uses the grid named `globalGrid` and ignores any others. If there are no Grid blocks named `globalGrid`, the last Grid block will be used and all preceding Grid blocks will be ignored.

Grid defaults to a uniform Cartesian grid if you omit the kind parameter or when you set *kind* to `uniCartGrid`, regardless of the presence or value of `coordinateSystem` parameter.

Note: There are no restrictions on the ratio of grid sizes among the dimensions. However, Tech-X recommends novice users unfamiliar with effects of large aspect ratio cells, keep the grid sizes, `DX`, `DY`, and `DZ`, within a factor of 5 from each other.

By convention, Vorpal lays out its axes as follows:

x: left to right y: front to back z: bottom to top

Grid Parameters

`maxIntDepHalfWidth` (*integer*, *optional*, *default = 1*)
Maximum half-width of either the interpolator or depositor stencil.

Please see *Additional Attributes for Particle Simulations* for more information.

`maxCellXings` (*integer*, *optional*, *default = 1*)
The maximum number of cells a particle is allowed to cross in one time step. This does not override the `maxcellxing` parameter specified in the species block, but must be equal to or larger than any species block maxcellxing value. This is used in calculating the number of guard cells found in the simulation. Please see *Additional Attributes for Particle Simulations* for more information.

`kind` (*string*)
Specifies type of grid to use, and is one of the following options.

- `uniCartGrid` (default) Works with VSimBase, VSimEM, VSimPD, VSimPA, and VSimMD licenses.

  `numCells` (integer vector, required) Sets number of cells in the x, y, and z directions. Should contain at least as many values as there are dimensions in the grid. Example usage:

  ```
    numCells = [5 10 15]
  or
    numCells = [NX NY NZ]
  ```

  `lengths` (float vector, required) Sets lengths of the simulation space in meter units. Should contain at least as many values as there are dimensions in the grid. Example usage:

  ```
    lengths = [0.1 0.1 0.1]
  or
    lengths = [LX LY LZ]
  ```

  `startPositions` (float vector, required, default = `[0 0 0]`) Specifies where the starting coordinates in the simulation space are in meter units. Should contain at least as many values as there are dimensions in the grid. Example usage:

  ```
    startPositions = [-0.05 0.0 0.0]
  or
    startPositions = [XBGN YBGN ZBGN]
  ```

- `coordProdGrid` Works with VSimPD and VSimMD licenses.

  `coordinateSystem` (optional, default=Cartesian) Sets the coordinate system of the grid. One of:

  – `Cartesian` (default), e.g. x,y,z

- `Cylindrical`, e.g. z, r, phi
- `Polar`, e.g. r, phi,z
- `Tubular`, e.g. phi, z, r

`includeCylAxis` (integer, optional, default = `0`) Whether or not to include the lower bound in the radial direction that corresponds to `r = 0` in the simulation. If includeCylAxis is true, one needs to make sure that the starting position in the r direction (dir1) is set to `0.0`. If includeCylAxis is false, one needs to make sure that the starting position in the r direction (dir1) is set to greater than `0.0`. Setting includeCylAxis to true in the Grid block sets a flag that allows for proper evaluation of the nodal volume element and Ez area element for the first cell. One should also take care to set includeCylAxis in any necessary FieldUpdater blocks. See: *FieldUpdater* for the different kinds of CoordProd updaters that are available.

`coordinateGrid` (block, required) Specifies the lower and upper bounds and cell size of each dimension of the coordinate product grid independently in its own CoordinateGrid block. See *CoordinateGrid*.

Example Uniform Cartesian Grid Block

```
<Grid globalGrid>
  kind = uniCartGrid
  numCells = [40 20 20]
  lengths = [5e-06 5e-05 5e-05]
  startPositions = [0.0 -2.5e-05 -2.5e-05]
</Grid>
```

Example Cylindrical Grid Block

```
<Grid globalGrid>
  kind = coordProdGrid
  coordinateSystem = Cylindrical
  includeCylAxis = 1
  <CoordinateGrid dir0>
    sectionBreaks = [ZBGN ZEND]
    deltaAtBreaks = [DZ DZ]
  </CoordinateGrid>
  <CoordinateGrid dir1>
    sectionBreaks = [RBGN REND]
    deltaAtBreaks = [DR DR]
  </CoordinateGrid>
</Grid>
```

3.2.2 CoordinateGrid

`CoordinateGrid`:

Specifies each dimension of the coordinate product grid independently in its own CoordinateGrid block. (There is not a boundary condition to treat the origin of polar coordinates.)

Valid CoordinateGrid names include:

- x
- y

- z
- r
- phi
- dir0 (generic axes designation for specified coordinate system)
- dir1 (generic axes designation for specified coordinate system)
- dir2 (generic axes designation for specified coordinate system)

The exact coordinate of each grid point is internally known by Vorpal and is not available to the user. Instead, you can select break points to describe the grid. Vorpal automatically fills the sections between breakpoints. The non-uniform grid will be produced according to the following rules:

1. There will be a grid line exactly at the specified sectionBreaks.
2. There grid spacing to either side of the sectionBreaks will be approximately the corresponding value from deltaAtBreaks.
3. The grid will be filled in between sectionBreaks with a smoothly varying grid spacing based upon geometric expansion or contraction.

CoordinateGrid Parameters

`sectionBreaks`

List of breakpoints to construct the grid points along the coordinate axis associated with each block. There is no upper limit on the number of break points specified in sectionBreaks, and there must be a value in deltaAtBreaks for each break point specified in sectionBreaks. The coordinate axis will span from the first value to the last value specified in sectionBreaks, so you must specify a minimum of two sectionBreaks.

Required Parameters:

`deltaAtBreaks` (required)

Grid spacing between breakpoints.

`defRadius` (required,default=1meter)

Default radius with which to compute physical lengths or areas for Tubular coordinates; expressed in meters. In the case of a 1D or 2D simulation using Tubular coordinates, the radial direction is ignored.

Sample usage:

```
defRadius = 1.00
```

Example CoordinateGrid Block

```
<CoordinateGrid z>
    sectionBreaks = [0.000 0.050 0.300 0.400]
    deltaAtBreaks = [0.005 0.005 0.020 0.020]
</CoordinateGrid>
```

3.2.3 Cylindrical Coordinates

To use cylindrical coordinates, set `coordinateSystem = Cylindrical` in the Grid block. In the case of a cylindrical grid, expressions that normally call for *x*, *y* coordinates (see *ParticleSource* blocks, *STFunc* blocks, etc.) instead refer to *z*, *r* respectively, however you should still use *x* and *y* in the actual expressions.

Example

```
# The cylindrical grid
<Grid globalGrid>
    kind=coordProdGrid
    coordinateSystem=Cylindrical
    includeCylAxis=1
    <CoordinateGrid dir0>
        sectionBreaks=[ZBGN ZEND]
        deltaAtBreaks=[DZ DZ]
    </CoordinateGrid>
    <CoordinateGrid dir1>
        sectionBreaks=[RBGN REND]
        deltaAtBreaks=[DR DR]
    </CoordinateGrid>
</Grid>
```

Also see

- *STFunc Block*.
- *ParticleSource*.

3.2.4 Additional Attributes for Particle Simulations

In some types of particle simulations, a user may want to add one or two additional Grid block attributes, both of which affect the number of guard cells.

maxIntDepHalfWidth

In particle simulations in which you use higher-order particle depositors or interpolators, you should invoke the `maxIntDepHalfWidth` parameter with a value no less than `(int)(particleOrder/2)+1`. This is because higher order particle depositors will spread a particle's charge over more than just the nearest neighbor grid nodes.

maxCellXings/maxcellxings

It is not recommended to allow particles to move more than a cell in a single time step. By skipping over more than a single cell, particles will not be accelerated in a completely self-consistent way (since they will experience slightly non-local fields), which will introduce inaccuracies in the simulation.

However, if there are a few fast-moving particles at the top end of a speed distribution which are unimportant to the overall physics of the simulation, it would be inconvenient to have to set a time step short enough to keep these fast particles from crossing more than a cell per time step.

The `maxCellXings` attribute in the grid block combined with the `maxcellxings` (note the difference in case) attribute in the species block can be used in such a situation. Both of these attributes should be set to the same value. These two attributes must be set appropriately if particles are able to travel more than a cell in a time step.

The `maxCellXings` attribute in the grid block will (along with the `maxIntDepHalfWidth` attribute, see *Guard Cell Calculation and Setting Overlap in Fields* below) sets the number of "guard cells" around the simulation domain which absorb particles leaving the simulation. By default, there is one guard cell surrounding the simulation domain. If particles can travel more than a cell per time step, they can skip over this absorber and cause a segmentation fault when they ask for the field values from outside the simulation domain. For example, a setting of `maxCellXings = 3` will set a layer of three guard cells around the simulation grid.

The `maxcellxings` attribute in the species block will impose a speed limit on each component of the velocity, independently. So, if `maxcellxings = 3` in a simulation, then the speed limit for the x-component of velocity will be $3 * DX/DT$, the speed limit for the y-component will be $3 * DY/DT$, and the speed limit in z will be $3 * DZ/DT$. Any particles with a velocity component larger than the limits will have the speed reduced to the speed limit for each velocity component in violation of it's respective speed limit. This will likely result in the modification of both the magnitude and direction of the velocity of particles in an unphysical way. For simulations in which a significant number of particles have their speeds limited, you should lower the time step for the simulation.

To see the effect of the speed limit, set the verbosity level to "NOTICE." This will print out the numbers of particles with limited speeds in the run log. Additionally, the phase space plot can be checked. If there are a significant number of particles piled up along a vertical or horizontal line in the plot, reduce the time step.

Guard Cell Calculation and Setting Overlap in Fields

The number of guard cells is determined based on the preceding parameters. Each field in the `multiField` block should include a two-component vector attribute, `overlap`, which is set according to the following.

For a non-dep field:

- overlap[0]=maxIntDepHalfWidth + (maxCellXings - 1)
- overlap[1]=maxIntDepHalfWidth + (maxCellXings - 1)

This defaults to overlap = [1 1]

For a dep field:

- overlap[0]=maxIntDepHalfWidth + (maxCellXings - 1)
- overlap[1]=maxIntDepHalfWidth + (maxCellXings - 1) + 1

This defaults to overlap = [1 2]

Please see *Field* for more details on manually overriding the number of guard cells for an individual field.

See also

- *Grid*

3.3 Decomposition

3.3.1 Decomp

Determines the domain decomposition and periodicity. The Decomp block determines how the simulation is broken up into domains for parallel processing and which directions are periodic.

If the domain boundaries lead to a specification of a number of processors that is not found at runtime, Vorpal will reconfigure the domain decomposition to match the runtime number of processors. That is, if you do not specify a decomposition, Vorpal will try to calculate an appropriate domain decomposition for you.

The decomposition algorithm used by Vorpal proceeds by first doing a prime factorization of the number of processors. It then uses that list of factors to divide up each dimension using the largest remaining factor on the dimension with the largest number of cells.

For example, for a simulation using 30 processors and a domain size of `200 x 100 x 100`, Vorpal will perform the following calculations:

```
30 = 5*3*2
```

The *x* direction has the largest number of cells (200) and 5 is the largest factor of the number of processors, so we divide *x* into 5 regions, each

```
40x100x100
```

With the *x* direction done, *y* is the now the direction with the largest number of cells (100), so Vorpal uses the next largest factor to divide it into 3 sections:

```
40x{33 or 34}x100
```

Finally, Vorpal partitions *z* with the remaining factor is 2, dividing it into 2 pieces.

```
40x{33 or 34}x50.
```

That is the size of our final domains.

In practice, for parallelism, Vorpal simulations should have approximately 20 - 40 cells in each dimension. Otherwise, the amount of messaging per computation increases and the benefits of using multiple processors is outweighed by the cost of the communication between them. The 20 - 40 rule depends on the complexity of the calculations being done. With a large number of particles per cell (20 or more) you could use approximately 20 cells in each direction.

The decomposition may affect what features can be used; for example, guard cells require a domain of at least four cells in every direction. If the number of processors and dimensions of the simulation chosen cause this requirement to be violated, you will see errors in the simulation.

Decomp Parameters

`kind` (*string*)
: **(deprecated in v8.0)** `regular` only allowed at present. Version 8.0+ should simply omit the kind parameter.

`periodicDirs` (*integervector*)
: Directions to have periodic boundary conditions.

`decomp0` (*integervector*)
: Domain boundaries along direction 0.

`decomp1` (*integervector*)
: Domain boundaries along direction 1.

`decomp2` (*integervector*)
: Domain boundaries along direction 2.

Example of Default Decomposition Generation

In this example, Vorpal uses the aforementioned algorithm (prime factorization) to automatically generate a decomposition for a parallel simulation. The *y*- and *z*- directions will be periodic as indicated by use of *`periodicDirs`*.

```
<Decomp decomp>
   periodicDirs = [1 2]
 </Decomp>
```

Example of User-specified Decomposition

In this example, assuming there is a 200 x 200 mesh grid as described in the section on Grid cell specification, the user requests domain boundaries in the x-direction at 50, 100, and 150, and in the y-direction for 100. This will produce 8 domains, requiring that 8 processors be used for the simulation. If a different number of processors is actually used, Vorpal will default to the auto-generated (prime factorization) decomposition as discussed in the general description of Decomp, above.

```
<Decomp  decomp>
    periodicDirs = [1 2]
    decomp0 = [50 100 150]
    decomp1 = [100]
</Decomp>
```

3.4 GridBoundary

3.4.1 GridBoundary Blocks

GridBoundary

Block for modeling curved surfaces (such as spheres or cylinders). A GridBoundary can be referenced by constructs such as *Initial and Boundary Conditions*, *ParticleSource*, *ParticleSink*, and other constructs.

To restore a GridBoundary on restart of a simulation without recalculating it, see *useGridBndryRestore*.

GridBoundary Parameters

kind
One of:

- **funcGridBndry** Works with VSimEM, VSimPD, and VSimMD licenses.

  Defines a region with an *STFunc Block* block. The inside is where the function value is non-negative. See funcGridBndry and the example *Example GridBoundary Block*.

- **rgnGridBndry** Works with VSimEM, VSimPD, VSimMD, and VSimSD licenses.

  A rgnGridBndry defines a GridBoundary by using one or multiple **STRgn** blocks. For more information about the types of regions available, see *STRgn*. A rgnGridBoundary is a good way to combine multiple regions into one large region using unions or intersections. See rgnGridBndry.

- **gridRgnBndry** Works with VSimEM, VSimPD, and VSimMD licenses.

  Defines a region with a STereoLithography (stl) file. See gridRgnBndry.

calculateVolume (*optional*)
Forces the grid boundary to calculate various auxiliary quantities that are used by cut-cell particle boundaries. Examples include the cut-cell volume and normal to the surface.

subMesh (integer, default = 7): When *calculateVolume* is true, the **GridBoundary** will calculate area/volume fractions for cells cut by the boundary (i.e., the fraction of the cell inside the boundary). Sometimes a **GridBoundary** is too complicated to allow accurate calculation of volume fractions (such as a surface that isn't smooth with small curvature compared to a cell size, or when multiple surfaces cut through a cell). If **GridBoundary** realizes that the surface is too complicated, it will estimate the inside fraction by sampling `subMesh` raised to NDIM power points regularly-spaced within each cell to see whether they are inside or outside the boundary. Sampling is a less accurate but more robust method for calculating volume fractions.

dmFrac (*float*)
Fraction of the Courant time step for which the simulation will be stable when using the Dey-Mittra boundary algorithm with Yee electromagnetics. For example, if `dmFrac = 0.25`, then your time step for the simulation must be 25% of the Courant time step.

boundaryPrecision (*float*, *default = 1.e-5*)
Precision to which lengths (such as the length of a cell edge inside a **GridBoundary**) are calculated.

gridBndryFuncIsSmooth (*bool*, *optional*)
If true then Vorpal assumes the boundary is a smooth function of x,y,z which allows for faster calculations of certain quantities.

dumpPeriodicity (*integer*, *optional*)
How often to dump the data; indicates data is to be dumped whenever the time step has increased by this amount. The command line parameter -d overrides this variable. Must have this defined in an input file or on the command line.

For more dumping options, see the Dumping Fields, Particles, and GridBoundaries section in Output Data in the VSim User Guide.

Example GridBoundary Block

```
<GridBoundary plane>
  boundaryPrecision = 1e-14
  calculateVolume = true
  subMesh = SUBMESH
  kind = funcGridBndry
  gridBndryFuncIsSmooth = true

  <STFunc function>
    kind = expression
    expression = (x/PI + y/2. + LZ/exp(3.)) - z
  </STFunc>

</GridBoundary>
```

3.4.2 STRgn Blocks

STRgn

Block for modeling curved surfaces as Constructive Solid Geometry (CSG) objects. This allows complex shapes to be created by combining geometry primitives with the three operations of union, difference and intersections. This is primarily for creating geometries to use with the *GridBoundary* block to create complex boundaries.

Note: The name of the STRgn block defined can only be set to “region” (see example below). If there are multiple nested STRgn blocks, only the first STRgn block must be named “region”.

Also see *stRgnMask* for information on using STRgn to assign properties of a fluid to an area or volume.

STRgn Example Block in a GridBoundary Block

```
<GridBoundary magTestPillBox3DparaShortPecShapes>
 kind = rgnGridBndry
 calculateVolume = 1
 dmFrac = 0.5

 <STRgn region>  ##### block name must be 'region'
  kind = stFuncRgn
  <STFunc function>
   kind = expression
   expression = H(z-(0.5))*H((0.4)-z)*H(0.4^2-x^2-y^2)
  </STFunc>
 </STRgn>
</GridBoundary>
```

3.5 EM Field

3.5.1 EmField Block

EmField

Blocks define the properties of the electromagnetic (EM) fields in the simulation. An EmField can be one of many types, with the most commonly used being the explicitly time-advanced which can be constructed with the `emMultiField` and the *basicEM.mac*, in which the basic components of the field are on the Yee mesh, and these fields are then averaged to cell nodes for use by other objects. The basic setup of the Yee EM field is illustrated below. In this setup, electric fields are located on the edges of grid cells, and magnetic fields are on the faces of the cell surfaces.

This EmField section describes general parameters for use with all EmField kinds, followed by descriptions of each individual EmField kind. Parameters specific to each individual kind are discussed within the description of each kind. **`InitialConditions`** and **`BoundaryConditions`**, including `outGoing` and kinds for EM fields only, are discussed in *Initial and Boundary Conditions*.

Other fields include specified constant or functional fields and electrostatic fields. Multiple EM fields can co-exist in a simulation. Since each field will have a unique name, other objects in the simulation can reference them individually.

Electromagnetic fields are described by EmField blocks. Defining the EM field (EmField) input block involves defining the kind parameter, as well as nested input blocks that describe the electromagnetic boundary conditions.

Vorpal has several different algorithms that can be used to model EmFields. You specify which algorithm you would like Vorpal to use by using the kind parameter. For each different kind, different parameters apply.

All EmField kinds can specify initial and boundary conditions. Periodic boundary conditions are defined in the **`Decomp`** input block.

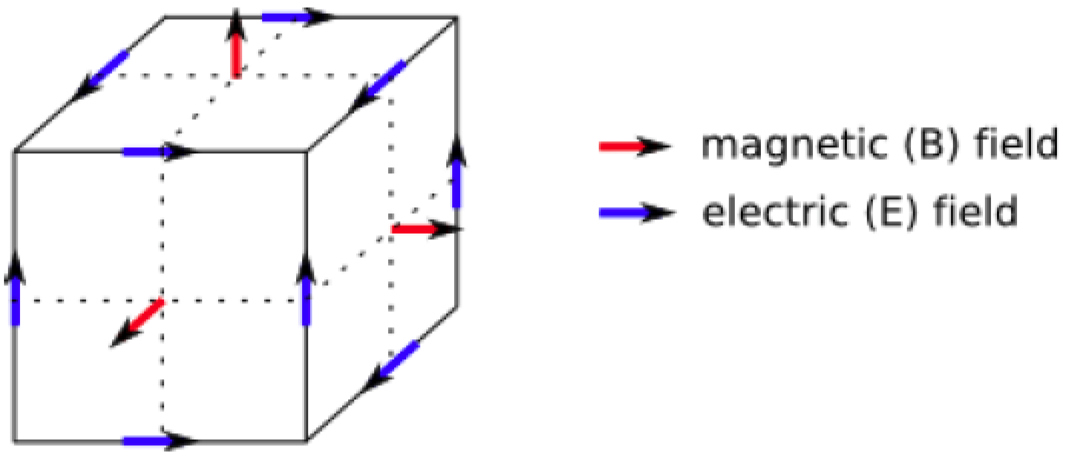


Fig. 3.1: Yee model for placing fields on the grid

EmField Parameters

EmField: Electromagnetic field implementation algorithm.

`kind`

One of the following:

- `constEmField`: Works with VSimBase, VSimEM, VSimMD, VSimPA, and VSimPD licenses.

  EM field in which components are held constant throughout the simulation.

- `emMultiField`: Works with VSimBase, VSimEM, VSimMD, VSimPA, and VSimPD licenses.

  Electromagnetic field described with the **`MultiField`** syntax. To construct standard Yee mesh explicit update use the basicEM macro.

- `funcEmField`: Works with VSimBase, VSimEM, VSimMD, VSimPA, and VSimPD licenses.

  Values for this field are set by a space/time function using the **`STFunc`** object.

- `yeeStaticEmField`: Works with VSimBase, VSimEM, VSimMD, VSimPA, and VSimPD licenses.

  Contains an electric field only. The **`yeeStaticEmField`** field is determined by Poisson solve.

`rhojweighting` (*string*)

Option for current/charge deposition when using higher-order particles.

`gridBoundary` (string, default = `sphere`)

Defines the boundaries for an EM Field in the simulation. One of:

- `sphere` (default)
- `universe`
- `pycavity`
- `stlfile`
- `user-defined boundary`

`interpolation` (string)

Defines the interpolation used with higher-order particles.

- **linearFromNodalFields:** (default in case in which the interpolating fields are on the nodes)
- **esirk1stOrder (default in case in which the** Interpolating fields are on the Yee mesh): Corresponds to bilinear or trilinear interpolation from the Yee mesh points. With a VSimPA or VSimPD license up to 7th order interpolation (esirk7thOrderCP) is available.
- **esirk1stOrderCP (default in case in which the** Interpolating fields are on the Yee mesh): Corresponds to bilinear or trilinear interpolation from the Yee mesh points. With a VSimPA or VSimPD license up to 7th order interpolation (esirk7thOrderCP) is available. This variant is intended for use with coordProd (CP) grids.

interpOrder (string, default = linear)

String used in conjunction with the polynomial interpolation algorithm only. interpOrder describes the order of the interpolation from the Yee mesh. Interpolation options include:

- linear (default)
- quadratic
- cubic
- quartic

Note: Most of the time, *rhojweighting* and interpolation parameters for EmField will be the same. However, nodal fields use linearFromNodalFields and the Yee field uses esirk1stOrder. That is:

```
rhojweighting = esirk1stOrder
interpolation = esirk1stOrder
```

However, in cases in which:

```
rhojweighting = areaWeighting
```

then:

```
interpolation = linearFromNodalFields
```

which is the default for fields in which:

```
offset=none
```

Otherwise the default for fields with any other offset is:

```
interpolation = esirk1stOrder
```

In the case in which:

```
rhojweighting = bilinear
```

then:

```
interpolation = linearFromNodalFields
```

which is the default for fields in which:

```
offset=none
```

Otherwise the default for fields with any other offset is:

```
interpolation = esirk1stOrder
```

Every gridField object has a default interpolation algorithm.

In the example below, the interpolation is set in a higher-level object and is valid for all gridField objects created by that EmField:

```
<EmField ...>
   kind = emMultiField
   interpolation=...
   ...
</EmField>
```

The following code demonstrates how to change the interpolation for each gridField object individually:

```
<MultiField ...>
  <Field ...>
    kind = gridField
    interpolation=...
  </Field>
  ...
</MultiField>
```

See also

- *Initial and Boundary Conditions*
- *yeeStaticEmField*

EmField Block Kinds

constEmField

constEMField:

Kind of **EmField** that defines a constant field.

constEMField Parameters

elecField (*float vector*)
The electric field.

magField (*float vector*)
The magnetic field.

hasB (*option, default=true*)
Flag to specify that the magnetic field (if present) will be used in the species update step.

Example constEMField Block

```
<EmField constEM>
    kind = constEmField
    elecField = [0. 0. 0.]
    magField = [0. 0. 0.]
</EmField>
```

See also

- *EmField*

emMultiField

The **emMultiField** kind of **EmField** updates all the electromagnetic fields using the **MultiField** feature. Please see the *Multifield* section of this manual for detailed information about **MultiField** and **MultiField** parameters.

Use `kind = emMultiField` to create a **MultiField** that uses the **EmField** interface. Any valid **MultiField** parameters are valid for **emMultiField** except for `depFields`.

If **emMultiField** is going to be used with particles, the particle current/density must be brought in through `externalFields = [SumRhoJ]`.

If **emMultiField** is going to be used with particles, the particles must be told which fields are to be used for the particle push algorithm. You have 3 options:

- Use the `emField =` syntax and define nodal fields specifically named `nodalE` and `nodalB`.
- Use the `emField =` syntax, change the interpolation to something other than `linearFromNodalFields`, and also specifically name your Yee fields `ElecMultiField` and `MagMultiField`.
- Use the `fields=[]` syntax and specify which fields are to be used for the particle push.

emMultiField Parameters

hasB (*option*, *default=true*)
: Flag to specify that the magnetic field (if present) will be used in the species update step.

needsRho (*option*, *default=false*)
: Flag to specify that the charge density is to be used for update step.

needsJ (*option*, *default=true*)
: Flag to specify that the current density is to be used for update step.

funcEmField

Defines an EM field using a block to describe the characteristics of the EM field. **funcEmField** requires use of a code block.

funcEmField Block Parameters

STFunc (block)

The function that sets the value for a specific field component. The name of the block determines which component. E0,E1,E2 sets the x,y and z components of the electric field and B0,B1,B2 does the same for the magnetic field. See *STFunc Block*.

Example funcEmField Block

```
<EmField myExternalField>
  kind = funcEmField
  <STFunc E0>
    kind = expression
    expression = EX_1 * cos(K_PE * x) * H( DRIVE_TIME - t )
  </STFunc>
</EmField>
```

yeeStaticEmField

EM field that solves Poisson's equation for electrostatic potential.

Defining an EM field of `kind = yeeStaticEmField` requires that you also define nested input blocks for:

`Solver:` See *Solver*.

`BoundaryCondition:` See below for more information on kinds of electrostatic boundary conditions.

yeeStaticEmField Parameters

`affectedBySpeciesList` (*integer vector*)
List of species affected by the ES solver. Optional parameter.

`hasB` (*option*, *default=true*)
Flag to specify that the magnetic field (if present) will be used in the species update step.

`needsJ` (*option*, *default=false*)
Flag to specify that the current density is to be used for update step.

yeeStaticEmField Boundary Condition Parameters

`kind`

Electrostatic boundary condition. One of:

- **`dirichlet:`** Specifies a value for the potential.
- **`neumann:`** Specifies a value for the derivative of the potential.
- **`periodic:`** Transparent boundary condition; does not emit, reflect, or absorb in the same manner as electromagnetic simulations. Specifying periodic allows particles to "wrap around" the computational domain. `periodic` is the default boundary condition used by Vorpal when an electrostatic boundary

condition is not explicitly specified. Periodic boundary conditions are defined in the **Decomp** input block.

Example yeeStaticEmField Block

```
<EmField  myEsField>
    kind = yeeStaticEmField
    <Solver mysolver>
        kind = gmres
        precond = sym_CS
        # Convergence criterium:
        tolerance = 1e-8
        # Maximum number of iterations
        #maxIter = 1000
        # Other choices for output are:
        # none, all, warnings, last, 1, 2, 3,  ...
        #
        # If the following is set to true, the
        # potential is written into a separate file
        dumpPotential = true
    </Solver>
```

```
# Set V = 0 V on far left (-x) wall
  <BoundaryCondition leftWall>
      kind = dirichlet
      minDim = 1
      lowerBounds = [0 0 0]
      upperBounds = [1 NY1 NZ1]
  <STFunc function>
      kind = constantFunc
      amplitude = 0.
  </STFunc>
  #     maxApplyTime = 10.
  </BoundaryCondition>
```

```
# Set V = 0 on right (+x) wall
  <BoundaryCondition rightWall>
      kind = dirichlet
      minDim = 1
      lowerBounds = [NX 0 0]
      upperBounds = [NX1 NY1 NZ1]
          <STFunc function>
              kind = constantFunc
              amplitude = 0.
          </STFunc>
  #     maxApplyTime = 10.
  </BoundaryCondition>
```

```
# Set V = 0 V on front wall (-y)
  <BoundaryCondition frontCathode>
      kind = dirichlet
      minDim = 2
      lowerBounds = [0 0 0]
      upperBounds = [NX1 1 NZ1]
      <STFunc function>
```

(continues on next page)

(continued from previous page)

```
          kind = constantFunc
          amplitude = 0.
      </STFunc>
  #     maxApplyTime = 10.
  </BoundaryCondition>
```

```
  # Set V = 0 V on back wall (+y)
    <BoundaryCondition backCathode>
        kind = dirichlet
        minDim = 2
        lowerBounds = [0 NY 0]
        upperBounds = [NX1 NY1 NZ1]
        <STFunc function>
            kind = constantFunc
            amplitude = 0.
        </STFunc>
    #     maxApplyTime = 10.
    </BoundaryCondition>
</EmField>
```

ES Simulations and Curved Boundaries

ES simulations can only deal with curved boundaries using a stair-step approximation of the curved surface. To use either of these methods, the user must specify the **GridBoundary** to be used. The **GridBoundary** is specified with the `gridBoundary` parameter in the **EmField** block.

Example of Curved Boundaries with ES Fields

```
<EmField myESField>
    kind = yeeStaticEmField
    gridBoundary = concentricsphere
    # Set potential on the grid boundary
    <STFunc boundaryFunc>
        kind = expression
        expression = -50000.0*H(ROUT*ROUT-x*x-y*y-z*z)
    </STFunc>
</EmField>
```

Solver

Solver: Required by a **yeeStaticEmField** as a nested Solver input block.

Adjusting Solver parameters requires experience and a feel for the simulation that you are modeling. Determining which solver parameters to change and to what they should be changed is an iterative process because not all solvers always give the same result. Tech-X recommends beginning by using the `gmres` solver with the `multigrid` pre-conditioner as a starting point because it tends to yield good convergence.

Note: Vorpal implements the boundary conditions in the solver in such a way that the matrix in the resulting system is non-symmetric (except for the fully periodic case). You must choose a solver that can deal with non-symmetric matrices. All the solvers with the exception of the cg solver work on non-symmetric matrices. You can use the `cg` solver in the fully periodic case (no explicit boundary conditions).

For more information about Vorpal's solvers and pre-conditioners, see the Trilinos Project Web site (http://trilinos.sandia.gov/)

Solver Parameters

`dumpPotential` (*boolean*, *default = false*)
By default, Vorpal does not write out the potential to an `.h5` file. You can save the potential to a file by setting the parameter *`dumpPotential`* to true.

`enforceZeroNetCharge` (*string*)
In a fully periodic electrostatic simulation, the total charge must be 0 (zero). If the solver parameter *`enforceZeroNetCharge`* is set to true, Vorpal spreads charges throughout the domain to ensure that the total charge is 0.

`kind` (*string*)
One of the solver types:

- `bicgstab`: Bi-conjugate gradients stabilized.
- `cgs`: Conjugate gradient squared.
- `gmres`: Generalized minimal residual.
- `cg`: Conjugate gradients.
- `tfqmr`: Transpose-free quasi minimal residual solver.

`precond` (*string*)
One of pre-conditioner types chosen to improve convergence behavior:

- `none`: No preconditioner
- `sym_CS`: Symmetric Cholesky Solution
- `Jacobi`: Point and block Jacobi
- `sym_GS`: Symmetric Gauss-Seidel
- `Neumann`: Neumann preconditioner
- `ls`: Least-squares polynomials
- `dom_decomp`: Domain decomposition
- `multigrid`: Multi-grid pre-conditioner

`smoother` (*string*)
Type of smoother:

- `GaussSeidel`: Gauss-Seidel
- `VBlockSymGaussSeidel`: Symmetric variable block Gauss-Seidel
- `Jacobi`: Jacobi smoother

`tolerance` (*real*, *default=1e5*)
Convergence criterion. When the residual reduction compared to the initial residual reaches the value specified for this parameter, the solver stops iterating.

maxIter (*integer*, *default=1000*)
Maximum number of solver iterations. If the solver never reaches tolerance value, it will continue to iterate. Solver stops after specified number of iterations.

Note: The following parameters are specific to the multi-grid pre-conditioner.

nLevels (*integer*)
Maximum number of coarser grid levels that multigrid pre-conditioner will construct. It is safest to choose, say 10 levels, to ensure that for larger problems enough levels are used. For smaller problems, only a few of the 10 levels will actually be used.

threshold (*real*)
Weight at which multigrid pre-conditioner considers two nodes to be connected. The textbook threshold value for well-behaved problems is 0.08.

getPrevStep (*boolean*, *default=false*)
If true, the solution at the previous time step will be used as the starting point for the current time step. This is useful to decrease the time to solution.

Example Solver Block

```
<Solver mysolver>
    kind = gmres
    # Other choices for electrostatic solvers are:
    # cgs, gmres, cg, tfqmr
    # For fully periodic systems with particles,
    # try this option:
    # enforceZeroNetCharge = true
    # Replaced sym_CS with multigrid
    precond = multigrid
    # Other choices for preconditioners are:
    # none, Jacobi, Neumann, ls, dom_decomp, sym_CS
    #
    # The following options are valid only for
    # multigrid pre-conditioner.
    #
    # Comment out when precond is NOT multigrid
    smoother = GaussSeidel
    # Other choices for smoothers are:
    #  VBlockSymGaussSeidel, Jacobi
    #
    # Number of grid levels
    # Comment out when precond is NOT multigrid
    nLevels = 10
    # threshold for refinement
    # Comment out when precond is NOT multigrid
    threshold = 0.08
    # End of options specific to multigrid
    # pre-conditioner.
    # convergence criterion:
    tolerance = 1e-8
    # maximum number of iterations; default is 1000
    # maxIter = 1000
    # Other choices for output are:
```

(continues on next page)

(continued from previous page)

```
    # none, all, warnings, last, 1, 2, 3, ...
</Solver>
```

3.6 ComboEmField Block

3.6.1 ComboEmField

The ComboEmField block combines other types of **`EmField`** descriptions defined within a simulation. For example, use ComboEmField to add a static functionally defined magnetic field to a dynamic electromagnetic field. Being able to combine different types of EmField descriptions is useful in the delta-f particle simulation method for adding in the background, varying electromagnetic field to the self-consistently generated electromagnetic field.

ComboEmField Parameters

`kind` (*string*)
Type of ComboEmField. Currently `comboEmField` is the only valid type.

`comboEmField` works with VSimBase, VSimEM, VSimMD, VSimPA, and VSimPD licenses.

`emField1` (*string*)
User-defined EmField.

`emField2` (*string*)
User-defined EmField.

`dumpField` (*integer*)
If non-zero, dump the values of the combined field into an HDF5 format output file.

`hasB` (*option*, *default=false*)
Flag to specify that the magnetic field (if present) will be used in the species update step.

`needsRho` (*option*, *default=false*)
Flag to specify that the charge density is to be used for update step.

`needsJ` (*option*, *default=false*)
Flag to specify that the current density is to be used for update step.

Example ComboEmField Block

```
<ComboEmField scaledEmSum>
  kind=comboEmField
  emField1=myESField
  emField2=scaledSextupoleField
</ComboEmField>
```

See also

- *EmField*

3.7 Multifield

3.7.1 Multifield Block

MultiField

Advanced method for simulations that enables precise user control over fields and algorithms.

A **MultiField** object contains a trio of sub-objects:

- *`Field`*
- *`FieldUpdater`*
- `UpdateStep`

The **MultiField** object facilitates development and application of new algorithms by providing separate specification of the data (*`Field`*), the update operations on those fields (*`FieldUpdater`*), and the algorithmic sequencing of those update operations (`UpdateStep`). A crucial feature of `UpdateStep` use is that instances of `UpdateStep` also govern communication during parallel processing. Additional special cases of these sub-objects also exist, including `InitialUpdateStep` (an update operation done once at the begining), *`FieldMultiUpdater`* (a special treatment of fields with more than one component), and *`PmlRegion`* (a special absorber boundary object for electromagnetic simulation).

This additional level of control proves useful in two general circumstances. First, it allows the user to make significant alterations in the sequencing of the traditional electrostatic and electromagnetic algorithms. These changes might include custom sources and boundary conditions, custom fields for particle forces, diagnostics, or post processing purposes, and/or custom combinations of different solvers in the same simulation. One example is to swap out the traditional electromagnetic solvers in favor of GPU accelerated solvers. A second useful circumstance is that it provides a means to create arbitrary partial differential equaion (PDE) solvers, beyond the commonly used electrostatic and electromagnetic simulations. One example provided with the software shows how to construct a solver for the heat conduction problem.

Electromagnetics Using MultiField

The traditional electromagnetic algorithm contains an electric *`Field`* and a magnetic *`Field`*, two instances of *`FieldUpdater`*: Ampere (which updates electric field) and Faraday (which updates magnetic field), and two instances of `UpdateStep`, one for each of the *`FieldUpdater`*. An `UpdateStep` can designate one-and-only-one field for parallel-process messaging. Therefore `UpdateStep` must be used twice, once to update-and-message the electric field, and again to update-and-message the magnetic field.

From this backbone electromagnetic algorithm, several commonly used enhancements arise and are found in the following examples:

Metallic Boundaries and the `FieldUpdater` deyMittraUpdater When there are metallic boundaries, an additonal *`FieldUpdater`* (deyMittraUpdater) applies a cut-cell boundary condition to the magnetic field. This additional *`FieldUpdater`* is usually grouped in the same `UpdateStep` as the Faraday *`FieldUpdater`*, so there remain only two UpdateSteps.

***Particles and the Magnetic Field* `UpdateStep`** When there are particles, the magnetic field `UpdateStep` is usually split, so that half is done at the end of the cycle, in order to have electric and magnetic fields at the same time during the cycle for particle forces. The remaining half of the update is then done at the beginning of the next cycle, resulting in three of `UpdateStep` instead of two.

Particles and Assisted Particle Force Computation with `Field` Definitions When there are particles, a second set of electric and magnetic *`Field`* definitions centered at the nodes may be introduced to assist in computing particle forces. The `edgeToNodeVec` *`FieldUpdater`* and `faceToNodeVec` *`FieldUpdater`* provide this functionality, and two new uses of `UpdateStep`, one for each new *`Field`*, are placed at the end of the cycle.

Outgoing Wave Boundary Conditions with an Open `FieldUpdater` Outgoing wave boundary conditions can be applied to the electric field with an open *`FieldUpdater`*. This additional *`FieldUpdater`* is usually grouped in the same `UpdateStep` as the Ampere *`FieldUpdater`*.

Dielectric Loss, Matched magnetic loss, and Traditional Bulk Electrical Conductivity Dielectric loss (complex dielectric), matched magnetic loss, and traditional bulk electrical conductivity, are added as additional uses of *`FieldUpdater`*, usually instances of *`STFuncUpdater`*, just before and just after Ampere and Faraday, and usually grouped together with the applicable field `UpdateStep`.

Electric Voltage and Current Sources Addition Electric voltage and current sources are added as additional uses of *`FieldUpdater`*, usually instances of *`STFuncUpdater`*, grouped with and applied just before or just after the Ampere `UpdateStep`.

Blocks Contained Inside MultiField Blocks

A **MultiField** is described between <MultiField *nameOfThisMultiField*> and </MultiField> tags. In addition to **MultiField** parameters, the **MultiField** block must contain the following blocks:

Field FieldUpdater FieldUpdateStep

and may in some circumstances contain the following additional blocks:

- InitialUpdateStep
- FieldMultiUpdater
- PmlRegion

Instances of *`Field`* are not required to appear in occurances of *`FieldUpdater`*, however, any *`FieldUpdater`* must appear at least once in an `UpdateStep` or an `InitialUpdateStep`.

MultiField Parameters

`kind`

Type of **MultiField** algorithm; one of:

- **Null:** Works with VSimBase, VSimEM, VSimPD, VSimPA, and VSimMD licenses.

  no kind parameter; this is the default. The default behavior of not specifying a kind parameter for a **MultiField** is usually the correct choice for simulations.

- **`lightFrameEnvelopeMultiField`:** Works with VSimPA license.

  Implements the Laser Envelope Model. Fields can be discretized on Laser Envelope Model grids and interpolated to particle positions. When using the Envelope Model with particles, the species should be set to `kind = envBoris`. This `lightFrameEnvelopeMultiField` kind of **MultiField** creates special grids that have the same size and spacing as the main grid, but that move in the +x direction at the speed of light:

- **active grid:** The position of this grid is initialized to be offset from the main grid by $c\Delta t/2$ in the x direction. It is shifted by $c\Delta t$ in the $+x$ direction every timestep.
- **alternate grid:** This grid is initially offset by $-c\Delta t/2$, and is shifted by $c\Delta t$ in the $+x$ direction every timestep.
- **activeLightFrameFields** Specifies the fields which exist on the active light frame grid.
- **alternateLightFrameFields:** Specifies the fields which exist on the alternate light frame grid.

updateStepOrder (*optional*, *string vector*)
This specifies the order in which to execute the UpdateSteps. By default, if this attribute is not given, the UpdateSteps are executed in the order in which they are specified in the input file.

shiftUpdaters (*optional*, *integer; default = ‘‘false‘‘*)
If `true`, the updaters in the MultiField will be shifted along with the fields. This is only relevant for updaters that rely on GridBoundary information that might shift.

restoreTimeFromField (*optional*, *string*)
Name of the field from which **MultiField** will get the current time when restoring from a dump. `restoreTimeFromField` is required for cases in which **MultiField** cannot automatically determine the proper field from which to restore. **MultiField** can make this determination if there is a field updated by an updater which is specified in an UpdateStep with `toDtFrac = 1`.

3.7.2 Field Block

Field

Field object code block contained between the tags:

```
<Field *nameOfThisField*>

</Field>
```

Field Parameters

kind (*string*, *default = regular*)
Type of field algorithm; one of:

- `regular`

  Works with VSimBase, VSimEM, VSimPD, VSimPA, and VSimMD licenses.

  Sets its guard cells values by communication or boundary conditions.

- `interior`

  Works with VSimBase, VSimEM, VSimPD, VSimPA, and VSimMD licenses.

  Field whose guard cells are expected to be invalid; when the field is shifted in a moving window simulation, (newly) interior cells in a domain are filled from the (formerly) interior cells on other domains.

- `depField`

  Works with VSimBase, VSimEM, VSimPD, VSimPA, and VSimMD licenses.

Field that fills guard cells and interior cells (during messaging) by adding together the values for a cell on all domains that have that cell (either as interior or a guard cell). Particles are written into `depFields`.

- `funcField`

  Works with VSimBase, VSimEM, VSimPD, VSimPA, and VSimMD licenses.

  Field that fills guard cells, or newly interior cells in a moving window simulation, by calculating their values from a function.

Each of these kinds is explained in detail in the following sections. If no kind is given, the default (and usually correct choice for most simulations) will be used by Vorpal.

`numComponents` (*integer*, *default = 1*)
Number of field components. Other field component values include:

- `1`: scalar field
- `3`: vector field
- `4`: edge4v vector
- `9`: tensor field

`offset` (*string*)
Offset positions where the field values are located within a cell. offset values include:

- `none`

  Corner with lowest coordinates.

- `center`

  Center of cell.

- `edge`

  Component j is at the center of the lowest edge parallel to the jth direction.

- `face`

  Component j is at the center of the lowest face perpendicular to the jth direction.

- `edge4v`

  For rhoJ, 0-component is at node, 1 is at x-center-edge, 2 at y-center-edge, 3 at z-center-edge

`overlap` (*integer vector*)
depField default = `[1 2]`, default for all others = `[1 1]` Number of lower and upper guard cells. The guard cells are important if higher order particles are near the simulation boundaries or for communications when running in parallel.

`dumpPeriodicity` (*integer*, *default = Multifield dumpPeriod setting*)
How often to dump the **`Field`** data relative to that set by the global variable *`globalvariables.dumpPeriodicity`*. For example, if a simulation generates 10 dumps, then setting the Field's `dumpPeriodicity` to `3` will dump the Field data on the 3rd, 6th, and 9th dumps. Setting `dumpPeriodicity = 0` will suppress dumping. If `dumpPeriodicity` is omitted from the attribute set, **`Field`** will inherit its parent **`MultiField`**'s `dumpPeriodicity` setting. If **`Field`** and its parent **`MultiField`** have different values of `dumpPeriodicity`, **`Field`** will only dump when both its own `dumpPeriodicity` and its parent **`MultiField`**'s `dumpPeriodicity` coincide, that is, it will not dump at a time when there is no **`MultiField`** dump.

For more dumping options, see the **Dumping Fields, Particles, and GridBoundaries** section in the **VSim User Guide**.

interpolation (*string*)
No offset default = linearFromNodalFields, default for all others = esirk1stOrder. Defines the interpolation method used for particles. Values include:

- **linearFromNodalFields** Works with VSimBase, VSimEM, VSimPD, VSimPA, and VSimMD licenses.
- **polynomial** Works with VSimBase, VSimEM, VSimPD, VSimPA, and VSimMD licenses.
- **esirkHalfSine** Works with VSimPD and VSimPA licenses.
- **esirkGaussian** Works with VSimPD and VSimPA licenses.
- **esirk1stOrder** Works with VSimBase, VSimEM, VSimPD, VSimPA, and VSimMD licenses.
- **esirk2ndOrder,... esirk7thOrder** Works VSimPD and VSimPA licenses.

No offset default = linearFromNodalFields, default for all others = esirk1stOrder

The default for offset = none is linearFromNodalFields, otherwise the default is esirk1stOrder. When using higher order particles, `maxIntDepHalfWidth` must be set accordingly in the **Grid** block (see *Additional Attributes for Particle Simulations*).

dumpOnly (*integer*)
If set to true `1` in a Field block of *kind* = `depField` then depositors depositing into that field will only execute at dump time. This is useful for occurrences of `depField` that are dumped but not needed for the update, such as charge density for EM PIC.

For more dumping options, see the Dumping Fields, Particles, and GridBoundaries section in Output Data in the VSim User Guide.

Blocks Contained Within a Field Block

BoundaryCondition (*block*)
Boundary condition for the field.

InitialCondition (*block*)
Initial condition for the field.

Source (*block*)
A source for the field. This is the same as a **BoundaryCondition** for the field, however may be more useful for introducing a condition over a region of the domain rather than a plane or line as you would expect from a boundary condition.

Also See

Initial and Boundary Conditions

Initial and Boundary Conditions

Nested input block for fields in *MultiField* that is applied only at the beginning of the simulation to describe initial conditions. In typical use, an InitialCondition is set throughout a volume (although this convention is not a requirement and, as with a BoundaryCondition, an InitialCondition may be set at select surfaces).

BoundaryCondition: Nested input block for fields in *MultiField* that is applied at every time step to describe boundary conditions. In typical use, boundary conditions are set at select surfaces (although this convention is not a requirement and, as with an InitialCondition, a boundary condition may be set throughout a volume).

Periodic boundary conditions are defined in the *Decomp* input block.

InitialCondition and BoundaryCondition Parameters

`kind` (*string*)
: Type of boundary condition. Possible kind values include:

- constant
- copy
- varadd
- varset
- outGoingWave (for electromagnetic fields only)

`lowerBounds` (*integer vector*)
: Lower bounds of the volume where initial or boundary condition is applied.

`upperBounds` (*integer vector*)
: Upper bounds of the volume where initial or boundary condition is applied.

`amplitudes` (*float vector*)
: Values of the amplitude for each of the indices.

`phases` (*float vector*)
: Values of the phase for each of the indices.

`components` (*integer vector*)
: Components to be set by the condition.

`STFunc` (*block*)
: The space-time function to be used. *STFunc Block* is a kind-dependent parameter. The **`STFunc`** block must be named **`componentn`** where *n* is the component the function is setting. *`STFunc`* applies only to the following initial or boundary conditions:

- varadd
- varset
- outGoing

`sourceLowerBounds` (*integer vector*)
: Lower bounds of the region from which to copy; copy boundary condition only.

`sourceUpperBounds` (*integer vector*)
: Upper bounds of the region from which to copy; copy boundary condition only.

`maxApplyTime` (*real*)
: Time until which this boundary condition is applied; this applies to only BoundaryCondition.

BoundaryCondition outGoing Kind

Specifying `kind = outGoing` in the BoundaryCondition block sets an open boundary which allows electromagnetic waves with specific characteristics to leave the domain, instead of reflecting, as they would from a conductor.

You can also use the open BoundaryCondition to launch a wave. You might want to do this, for example, in the case when you have launched a wave from the left, as a result of which some wave is reflected. You can use an outGoing BoundaryCondition to launch the wave while allowing the reflected wave back through. See the *Example of Using kind = open to Handle Wave Reflection*.

outGoing Kind Parameters

phaseVelocity (*float*)
: Sets the phase velocity of the wave the user would like to leave the boundary. Giving phaseVelocity a value of the speed of light allows plane waves in vacuum to leave a boundary. For modes in a waveguide or other structure, the phase velocity may differ from the speed of light. A perfectly matched layer is required for allowing modes with differing phase velocities all to leave a boundary.

Example BoundaryCondition Block for launching an electromagnetic wave

```
#  Wave launcher for E_y at left
<BoundaryCondition  lower0Launcher>
  kind = varset
  lowerBounds = [0 -1 -1]
  upperBounds = [1 21 21] # upperBounds = [1 NY1 NZ1]
  minDim = 1
  components = [1]
  <STFunc  component1>
    k = [5026548.24574 0. 0.] # k = [KAY 0. 0.]
    vg = 299790000.0 # vg = LIGHTSPEED
    omega = 1.50690889859e+15
    amplitudes = [2.56837310961e+12]
    phases = [0.] # sine
    kind = planeWavePulse
    widths = [4e-06 2e-05 2e-05] # widths = [WXPUMP WYPUMP WZPUMP]
    origin = [-4e-06 0.0 0.0] # origin = [XSTARTPUMP 0.0 0.0]
  </STFunc>
</BoundaryCondition>
```

Examples of BoundaryCondition Blocks for conductor electromagnetic boundary conditions

```
# Set E_y to zero on x-lower boundary
<BoundaryCondition xLowerConductor>
  kind = constant
  # Value same at all cells
  lowerBounds = [0 -1 -1]
  # Lower cell limits for BC application
  upperBounds = [1  21 21]
  # Upper cell limits for BC application
  # There should be one amplitude for each index
  indices = [1]
  # E_y set by this
  amplitudes = [0.]
</BoundaryCondition>
```

```
# Set E_y and E_z to zero on x-upper boundary
<BoundaryCondition xUpperConductor>
  kind = constant
  lowerBounds = [40 -1 -1]
  upperBounds = [41  21 21]
  indices = [1 2]
  amplitudes = [0. 0.]
</BoundaryCondition>
```

Example of Using `kind = outGoingWave` boundary condition

```
<BoundaryCondition rightOpen>
  kind = outGoingWave
  phaseVelocity = $V_OVER_C*LIGHTSPEED$
  normalDir = 0
  velOverC = V_OVER_C
  lowerBounds = [NX -1 -1]      # Lower cell limits for application of BC
  upperBounds = [NX1 NY1 NZ1]  # Upper cell limits for BC application
  components = [1 2]              # field components boundary is applied to
  <STFunc function>
    kind = constantFunc
    amplitude = 0.
  </STFunc>
</BoundaryCondition>
```

Example of Using `kind = open` to Handle Wave Reflection

```
<BoundaryCondition leftOpenWaveLauncher>
  kind = outGoingWave
  phaseVelocity = $V_OVER_C*LIGHTSPEED$
  normalDir = 0
  lowerBounds = [0 -1 -1]
  upperBounds = [1 NY1 NZ1]
  components = [2]
  <STFunc function>
    kind = planeWavePulse
    amplitude = EWAVE
    phase = 1.57
    k = [KAY 0 0 ]
    omega = OMEGA
    vg = LIGHTSPEED
    widths = [5.e-6 1.e-5 1.e-5]
    origin = [-5.e-6 0.e-5 0.e-5]
  </STFunc>
</BoundaryCondition>
```

See also

- *MultiField*
- *Decomp*

3.7.3 FieldUpdater Block

FieldUpdater

Field updaters are defined between the tags:

```
<FieldUpdater *nameOfThisUpdater*>

</FieldUpdater>
```

FieldUpdater Parameters

`kind` (*required string*)

Type of updater; one of the CPU updaters:

- *curlUpdater*
- *curlUpdaterCoordProd*
- *cylEdgeToNodeVec*
- *deyMittraConstrainUpdater*
- *deyMittraUpdater*
- *divUpdater*
- *divUpdaterCoordProd*
- *dummyUpdater*
- *edgeToNodeVec*
- *epetraUpdater*
- *faceToNodeVec*
- *fieldBinOpUpdater*
- *geometryUpdater*
- *gradVecUpdater*
- *gradVecUpdaterCoordProd*
- *importFromFileUpdater*
- *lightFrameEnvelopeUpdater*
- *lightFrameEnvForceUpdater*
- *linearApplyUpdater*
- *linearSolveUpdater*
- *linIterUpdater*
- *linPlasDielcUpdater*
- *malUpdater*
- *neutralBoltzmannUpdater*
- *open*
- *permittivityUpdater*
- *phaseShiftVecUpdater*
- *poissonUpdater*
- *setEpsilonUpdater*
- *smooth1D*
- *STFuncUpdater*
- *unaryFieldOpUpdater*
- *userFuncUpdater*

  - *yeeAmpereDielVecUpdater*
  - *yeeAmpereUpdater*
  - *yeeFaradayUpdater*

`lowerBounds` (*integer vector*, *required*)
Lower bounds of updater region.

`upperBounds` (*integer vector*, *required*)
Upper bounds of updater region.

Advanced parameters available with certain updaters

Global region modification parameters

There are a number of parameters that modify the global region to be updated (as specified in `lowerBounds` and `upperBounds`) based on the component of the field being updated. These are available for the following updaters, which all update a single vector field:

- *curlUpdater*
- *curlUpdaterCoordProd*
- *yeeAmpereUpdater*
- *yeeConductorUpdater*
- *yeeFaradayUpdater*

The global region modification parameters are:

`expandToTopInComponentDir` (*optional integer*, *default = 0 (false)*)
When true: if the region upper bound in the direction of the component (mod 3) is the upper bound of the simulation, and that direction is not periodic, then the upper bound is increased by one so that the region includes the top of the simulation.

`contractFromBottomInComponentDir` (*optional integer*, *default = 0 (false)*)
When true: if the region lower bound in the direction of the component (mod 3) is the lower bound of the simulation, and that direction is not periodic, then the lower bound is increased by one so that the region excludes the bottom of the simulation.

`expandToTopInNonComponentDir` (*optional integer*, *default = 0 (false)*)
When true: if the region upper bound in a direction perpendicular to the component (mod 3) is the upper bound of the simulation, and that direction is not periodic, then the upper bound is increased by one so that the region includes the top of the simulation.

`contractFromBottomInNonComponentDir` (*optional integer*, *default = 0 (false)*)
When true: if the region lower bound in a direction perpendicular to the component (mod 3) is the lower bound of the simulation, and that direction is not periodic, then the lower bound is increased by one so that the region excludes the bottom of the simulation.

`expandAboveTopInComponentDir` (*optional integer*, *default = 0*)
If the region upper bound in the direction of the component (mod 3) is the upper bound of the simulation or higher, and that direction is not periodic, then the upper bound is increased by this value.

`expandBelowBottomInComponentDir` (*optional integer*, *default = 0*)
When true: if the region lower bound in the direction of the component (mod 3) is the lower bound of the simulation or below, and that direction is not periodic, then the lower bound is decreased by this value.

expandAboveTopInNonComponentDir (*optional integer*, *default = 0*)
When true: if the region upper bound in a direction perpendicular to the component (mod 3) is the upper bound of the simulation or higher, and that direction is not periodic, then the upper bound is increased by this value.

expandBelowBottomInNonComponentDir (*optional integer*, *default = 0*)
When true: if the region lower bound perpendicular to the direction of the component (mod 3) is the lower bound of the simulation or below, and that direction is not periodic, then the lower bound is decreased by this value.

Note: If both *expandToTopInComponentDir* and *expandAboveTopInComponentDir* are specified, and the conditions for expansion described above are met, the upper bound in the component direction is first increased by one, and then increased again by the value of *expandAboveTopInComponentDir*. If both *contractFromBottomInComponentDir* and *expandBelowBottomInComponentDir* are specified, and the conditions for contraction are met, the lower bound is first increased by one, and then decreased by the value of *expandBelowBottomInComponentDir*. Similar behavior occurs for the corresponding non-component direction parameters.

Local region modification parameters

There are two parameters available to expand the local update region beyond the local domain, into the guard cells. This can in certain cases be useful in simulations with periodic boundaries, to allow updaters to overwrite the values imposed by the periodic boundary conditions. It can also be used to make parallel algorithms more efficient, by updating the local guard cells directly rather than getting values messages from other domains. These are available for the following updaters:

- *curlUpdater*
- *curlUpdaterCoordProd*
- *open*
- *phaseShiftVecUpdater*
- *yeeAmpereUpdater*
- *yeeFaradayUpdater*

The local region modification parameters are:

cellsToUpdateBelowDomain (*optional integer vector, default = [0 0 0]*)
The number of cells (in each direction, x, y, and z) the updater updates below the local domain.

cellsToUpdateAboveDomain (*optional integer vector, default = [0 0 0]*)
The number of cells (in each direction, x, y, and z) the updater updates above the local domain.

StencilElement

A code block describing a stencil element in MultiField updaters that use a user-defined stencil. These updaters are:

epetraUpdater

linIterUpdater

StencilElement Parameters

minDim (*optional integer*, *default = 1*)
: If the dimension of the simulation is less than *minDim*, the element will not be used in the stencil.

rowFieldIndex (*required integer*)
: The index of the field to which this stencil element maps. Thus, when treating the stencil operation as a matrix, a value in this field corresponds to a matrix row.

columnFieldIndex (*required integer*)
: The index of the field from which this stencil element maps. Thus, when treating the stencil operation as a matrix, a value in this field corresponds to a matrix column.

cellOffset (*required integer vector*)
: The offset between the field component the stencil element maps to and the field component the element maps from. Thus, a *cellOffset* of $[m\ n\ p]$ means that a cell will be updated from a cell displaced by $m\Delta x + n\Delta y + p\Delta z$.

value (*required float*)
: The coefficient of the linear operation between the the mapped field values. Thus, when treating the stencil operation as a matrix, this is the value of the matrix entry.

FieldUpdater Kinds

CPU updaters:

curlUpdater

Works with VSimBase, VSimEM, VSimPD, VSimPA, and VSimMD licenses.

Multifield updater operation that solves equations of the form:

$$\frac{\partial \mathbf{F}}{\partial t} = A\left(\nabla \times \mathbf{G}\right) + B\mathbf{H},$$

via the operation:

$$F_i\left(t+\Delta t\right) = F_i\left(t\right) + \left(c_0 + c_1\Delta t\right)\left[A\left(\nabla \times \mathbf{G}\right)_{c_g+i} + BH_{c_h+i}\right],$$

$\mathbf{F}$ is the *writeFields*.

$\mathbf{G}$ is the first *readFields* .

$\mathbf{H}$ is an optional second *readFields* (if missing, the coefficient B is taken to be 0).

i is the updater component.

c_0 and c_1 are the *dtCoefficients* values (represented in the form $[c_0\ c_1]$)

A and B are the *readFieldFactors* values.

c_g and c_h are the *readFieldCompShifts* values (represented in the form $[c_g\ c_h]$ and usually both 0, see the note following *curlUpdater Parameters*).

curlUpdater Differencing Definitions

Forward curlUpdater differencing definition:

$$(\nabla \times G)_{i,j,k,x} = \frac{G_{i,j+1,k,z} - G_{i,j,k,z}}{\Delta y} - \frac{G_{i,j,k+1,y} - G_{i,j,k,y}}{\Delta z} \tag{3.1}$$

Backward curlUpdater differencing definition:

$$(\nabla \times G)_{i,j,k,x} = \frac{G_{i,j,k,z} - G_{i,j-1,k,z}}{\Delta y} - \frac{G_{i,j,k,y} - G_{i,j,k-1,y}}{\Delta z} \tag{3.2}$$

curlUpdater Parameters

The **curlUpdater** takes the `lowerBounds` and `upperBounds` parameters of *FieldUpdater*, as well as the *global region modification parameters* and *local region modification parameters*. In addition, **curlUpdater** takes the following parameters:

readFields (*required string vector*)
A vector of either one or two strings. The first string is the name of the field to take the curl of, and if provided, the second is the name of the field (multiplied by the specified factors) to add to the result.

writeFields (*required string vector*)
A vector containing a single element, which is the name of the field to update.

differencing (*required string*)
Either `forward` or `backward`, as described above.

useVecUpdater (*optional integer*, *default = 0 (false)*)
If true, the updater will update all three components of the vector field specified in *`writeFields`*, beginning with the specified *`component`*. The updated field must therefore have at least `component` $+ 3$ components.

component (*optional integer*, *default = 0*)
The field component to update, or if *`useVecUpdater`* is `true`, the first field component to update.

readFieldCompShifts (*optional integer vector, default = [0 0]*)
This vector must have the same number of elements as *`readFields`*. It specifies the amount by which to increment the component indices of the first field and the (optional) second field. It is equal to $[c_g \ c_h]$ in the description above. For example, if a magnetic field is represented by components 3–5 of the field `EandB`, then to calculate the curl of that magnetic field, one would specify `readFields = [EandB]` and `readFieldCompShifts = [3]`.

readFieldFactors (*optional float vector*)
If this is specified, there must be one element for each field specified in *`readFields`*. The terms in the update for each field are multiplied by the corresponding factors; they are the coefficients A and B in the description above. If not specified, the factors use values of 1 for each field.

dtCoefficients (*optional float vector, default = [1. 0.]*)
Two components $[c_0 \ c_1]$ as defined in the equation above. The result of the updater will be multiplied by $(c_0 + c_1 \Delta t)$, where Δt is the current time step.

gridBoundary (*optional string*)
If provided, only components on the interior of the specified GridBoundary will be updated. The method to define the interior is given in the `interiorness` parameters.

interiorness (*optional string*, *default = cellcenter*)
If the *`gridBoundary`* parameter is specified, this is the method the used to determine whether a component is *interior* to the boundary. The behavior depends on the `offset` specified in the updated *Field*. One of:

- **cellcenter:** If `offset = none`, or `offset = edge4v` and `component = 0`, then a cell is considered interior if its node is adjacent to at least one cell with center inside the boundary.

  If `offset = edge`, or `offset = edge4v` and `component` is not `0`, then a cell is considered interior if the edge specified by *`component`* is adjacent to at least one cell with center inside the boundary.

  If `offset = face`, then a cell is considered interior if the face specified by *`component`* is adjacent to at least one cell with center inside the boundary.

  If `offset = center`, then a cell is considered interior if its center is inside the boundary.

- **deymittra** If `offset = none`, or `offset = edge4v` and `component = 0`, then a cell is considered interior if all nodes adjacent to (i.e. displaced by a single edge from) its node are inside the boundary.

  If `offset = edge`, or `offset = edge4v` and `component` is not `0`, then a cell is considered interior if the edge specified by *`component`* has at least one adjacent node inside the boundary, and that edge is not ignored by the Dey-Mittra algorithm given the `dmFrac` parameter specified in the *`gridBoundary`*.

  If `offset = face`, then a cell is considered interior if all nodes adjacent to the face specified by *`component`* are inside the boundary.

  This *`interiorness`* option cannot be specified with `offset = center`.

- **dmnodal** This *`interiorness`* option is identical to `deymittra`.

lowerSkinDepth (*optional integer vector*)
Specifies the number of skin cells, in each direction, on the lower end of the local domain. The cells in the skin are updated before the fields specified as `messageFields` in the *UpdateStep or InitialUpdateStep block* are messaged. If not specified, the skin depth will be determined automatically.

upperSkinDepth (*optional integer vector*)
Specifies the number of skin cells, in each direction, on the upper end of the local domain. If not specified, the skin depth will be determined automatically.

Example Yee Ampere (Ey) Update Block

To solve the y component of Ampere's law with Maxwell's correction,

$$\nabla \times \mathbf{B} = \mu_0 \mathbf{J} + \mu_0 \varepsilon_0 \frac{\partial \mathbf{E}}{\partial t}, \tag{3.3}$$

via the **curlUpdater** operation

$$E_1(t+\Delta t) = E_1(t) + (0 + 1\Delta t)\left[c^2 \left(\nabla \times \mathbf{B}\right)_1 - \frac{1}{\epsilon_0} S_2\right], \tag{3.4}$$

where

$\mathbf{E}$ is the elecField, $\mathbf{B}$ is the magField, $\mathbf{J}$ is the current density, and $\mathbf{S}$ is the SumRhoJ field defined by

$$\mathbf{S} = (\rho, J_0, J_1, J_2), \tag{3.5}$$

use the following code:

```
<FieldUpdater ampere-y>
    kind = curlUpdater
    component = 1
    differencing = backward
    writeFields = [elecField]
    readFields = [magField SumRhoJ]
    readFieldFactors = [$c^2$ ~$(-1/\epsilon_0)$]
    dtCoefficients = [0. 1.]
    readFieldCompShifts = [0 1]
</FieldUpdater>
```

Adding S_2 is adding J_1, thus the component on the **SumRhoJ** field must be shifted so that J_1 is added to E_1.

curlUpdaterCoordProd

Works with VSimPD and VSimMD licenses.

This is a variation of the **curlUpdater** MultiField updater that should be used whenever non-uniform or non-Cartesian grids are used in the simulation. The FieldUpdater variety does not currently work with grid boundaries, however the FieldMultiUpdater variety does.

curlUpdaterCoordProd Parameters

The **curlUpdaterCoordProd** takes the `lowerBounds` and `upperBounds` parameters of *FieldUpdater*, as well as the *global region modification parameters* and *local region modification parameters*. In addition, **curlUpdaterCoordProd** takes the following parameters:

readFields (*required string vector*)
A vector of either one or two strings. The first string is the name of the field to take the curl of, and if provided, the second is the name of the field (multiplied by the specified factors) to add to the result.

writeFields (*required string vector*)
A vector containing a single element, which is the name of the field to update.

differencing (*required string*)
Either `forward` or `backward`, as described above.

useVecUpdater (*optional integer, default = 0 (false)*)
If true, the updater will update all three components of the vector field specified in *`writeFields`*, beginning with the specified *`component`*. The updated field must therefore have at least `component` + 3 components.

component (*optional integer, default = 0*)
The field component to update, or if *`useVecUpdater`* is `true`, the first field component to update.

readFieldCompShifts (*optional integer vector, default = [0 0]*)
This vector must have the same number of elements as *`readFields`*. It specifies the amount by which to increment the component indices of the first field and the (optional) second field. It is equal to $[c_g\ c_h]$ in the description above. For example, if a magnetic field is represented by components 3–5 of the field `EandB`, then to calculate the curl of that magnetic field, one would specify `readFields = [EandB]` and `readFieldCompShifts = [3]`.

readFieldFactors (*optional float vector*)
If this is specified, there must be one element for each field specified in *`readFields`*. The terms in the update for each field are multiplied by the corresponding factors; they are the coefficients A and B in the description above. If not specified, the factors use values of 1 for each field.

dtCoefficients (*optional float vector, default = [1. 0.]*)
Two components $[c_0\ c_1]$ as defined in the equation above. The result of the updater will be multiplied by $(c_0 + c_1\Delta t)$, where Δt is the current time step.

includeCylAxis (*optional integer, default = 0 (false)*)
Set this to true (1) if the cylindrical axis ($r = 0$) is included in this update. To obtain the correct behavior at the axis, one needs to specify two separate curlUpdaterCoordProd updaters; one with the axis and one without.

lowerSkinDepth (*optional integer vector*)
Specifies the number of skin cells, in each direction, on the lower end of the local domain. The cells in the skin are updated before the fields specified as `messageFields` in the *UpdateStep or InitialUpdateStep block* are messaged. If not specified, the skin depth will be determined automatically.

upperSkinDepth (*optional integer vector*)
Specifies the number of skin cells, in each direction, on the upper end of the local domain. If not specified, the skin depth will be determined automatically.

Example curlUpdaterCoordProd Block

```
<FieldUpdater  yeeFaraday_0>
    kind=curlUpdaterCoordProd
    useVecUpdater=1
    differencing=forward
    lowerBounds=[0 1 0]
    upperBounds=[NZ NR NPHI]
    dtCoefficients=[0.0 1.0]
    readFieldFactors=[-1.0]
    readFields=[elecField]
    writeFields=[magField]
</FieldUpdater>
<FieldUpdater  yeeFaraday_0_cyl>
    kind=curlUpdaterCoordProd
    useVecUpdater=1
    differencing=forward
    includeCylAxis=1
    lowerBounds=[0 0 0]
    upperBounds=[NZ 1 NPHI]
    dtCoefficients=[0.0 1.0]
    readFieldFactors=[-1.0]
    readFields=[elecField]
    writeFields=[magField]
 </FieldUpdater>
```

cylEdgeToNodeVec

Works with VSimMD and VSimPD licenses.

This updater interpolates all components of a vector field between grid nodes and edges in cylindrical coordinates. It should be used in conjucntion with the *linPlasDielcUpdater* in cylindrical coordinates, it is not necessary for standard edge to node updates.

cylEdgeToNodeVec Parameters

The **cylEdgeToNodeVec** takes the `lowerBounds` and `upperBounds` parameters of *FieldUpdater*, as well as the following parameters:

readFields (*required string vector*)
A single element, specifying the field to interpolate.

writeFields (*required string vector*)
A single element, specifying the field to update with the interpolated values.

nodeToEdge (*optional integer*, *default = 0 (false)*)
Whether to interpolate from nodes to edges instead of the default edges to nodes.

Example cylEdgeToNodeVec block

```
<FieldUpdater edgeToNodes>
  kind = cylEdgeToNodeVec
  lowerBounds = [0 0 0]
  upperBounds = [NZ1 NR1 NPHI1]
  readFields = [deltaelecField]
  writeFields = [deltaNodalE]
</FieldUpdater>
```

deyMittraConstrainUpdater

Works with VSimEM and VSimMD licenses.

Multifield updater that sets the electric field on the edges fully inside the conductor in a cut cell. This is done such that the interpolated value of the electric field at the center of the cut segment obeys the condition that the electric field parallel to the surface is `0`. For cuts with two unknown edges, it also uses the constraint that the derivative of the normal electric field at the center of the cut segment is `0`. By constraining the electric fields on the edges in this manner the interpolated fields will better match the correct behavior as they approach the conducting surface represented by the cut segment.

deyMittraConstrainUpdater is designed purely to provide better field interpolation in the cut cells and is not meant to do any dynamics. It should be used in conjugation with a *deyMittraUpdater*.

deyMittraConstrainUpdater Parameters

The deyMittraConstrainUpdater kind takes the `lowerBounds` and `upperBounds` parameters of *FieldUpdater*, as well as the following parameters:

readFields
(required string vector) A vector containing a single element, the electric field with an offset of *edge*.

writeFields
(required string vector): The same vector as in `readFields`.

gridBoundary
(required string): The boundary at which to modify the field.

Example deyMittraConstrainUpdater Block

```
<FieldUpdater deyMittraConstrain>
  kind = deyMittraConstrainUpdater
  lowerBounds = [0 0 0]
  upperBounds = [$XSIZE+1$ $YSIZE+1$ $ZSIZE+1$]
  readFields = [ElecMultiField]
  writeFields = [ElecMultiField]
  gridBoundary = cylinder
</FieldUpdater>
```

deyMittraUpdater

Works with VSimEM and VSimMD licenses.

MultiField updater that does the Yee Faraday update for Dey-Mittra (cut) cells. It updates those components of the magnetic field whose faces are cut by the specified `gridBoundary`, and not ignored by the Dey-Mittra algorithm given the `dmFrac` parameter of the boundary.

deyMittraUpdater Parameters

The deyMittraUpdater kind takes the `lowerBounds` and `upperBounds` parameters of *FieldUpdater*, as well as the following parameters:

readFields (*required string vector*)
A vector containing a single element, the electric field with an offset of *edge*.

writeFields (*required string vector*)
A vector containing a single element, the magnetic field to update with an offset of *face*.

gridBoundary (*required string*)
The boundary at which to update the field.

subtractYeeFaraday (*optional integer*, *default = 0 (false)*)
If true, the change in magnetic field given by the standard Yee update is subtracted from any components updated by this updater.

Example deyMittraUpdater Block

```
<FieldUpdater deyMittraFaraday>
  kind = deyMittraUpdater
  lowerBounds = [0 0 0]
  upperBounds = [XSIZE YSIZE ZSIZE]
  readFields = [ElecMultiField]
  writeFields = [MagMultiField]
  gridBoundary = cylinder
</FieldUpdater>
```

divUpdater

Works with VSimBase, VSimEM, VSimPD, VSimPA, and VSimMD licenses.

MultiField updater that takes the divergence of the of a field, and writes the result into another field.

divUpdater Parameters

The divUpdater kind takes the `lowerBounds` and `upperBounds` parameters of *FieldUpdater*, as well as the following parameters:

readFields (*required string vector*)
A vector containing a single element, the name of a vector field of which to take the divergence.

writeFields (*required string vector*)
A vector containing a single element, the name of a scalar field to update.

differencing (*optional string*, *default = backward*)
One of `forward` or `backward`, specifying the direction in which to take the finite difference. The default, `backward`, is generally used for taking the divergence of an edge field to update a node field; the `forward` value is used for the divergence of a face field to update a cell-centered field.

skipFirst (*optional integer*, *default = 0 (false)*)
Set this flag to `1` (true) to skip the first component of the vector field, i.e. to use components 1–3 in the divergence rather than 0–2. This should be used when taking the divergence of **J** in a **SumRhoJ** charge-current field. A **SumRhoJ** field has four components, the first of which is ρ; therefore the first component of this field must be skipped to get to the **J** components.

factor (*optional float*, *default = 1*)
A factor that multiplies the end result after taking the divergence.

gridBoundary (*optional string*)
If provided, only components on the interior of the specified GridBoundary will be updated. The method to define the interior is given in the `interiorness` parameters. If this parameter is provided, then the field specified in *`writeFields`* must have `offset = none` or `offset = center`.

interiorness (*optional string*, *default = cellcenter*)
If the `gridBoundary` parameter is specified, this is the method the used to determine whether a component is *interior* to the boundary. The behavior depends on the `offset` specified in the updated *Field*. One of:

- **cellcenter:** If `offset = none`, then a cell is considered interior if its node is adjacent to at least one cell with center inside the boundary.

  If `offset = center`, then a cell is considered interior if its center is inside the boundary.

- **deymittra:** If `offset = none`, then a cell is considered interior if all nodes adjacent to (i.e. displaced by a single edge from) its node are inside the boundary.

  This *`interiorness`* option cannot be specified with `offset = center`.

Example divUpdater Block

```
<FieldUpdater divergence>
  kind        = divUpdater
  lowerBounds = [ 0  0  0]
  upperBounds = [NX NY NZ]
  readFields  = [ElecMultiField]
  writeFields = [divE]
  skipFirst   = false
</FieldUpdater>
```

divUpdaterCoordProd

Works with VSimPD and VSimMD licenses.

MultiField updater that takes the divergence of the of a field, and writes the result into another field, on a non-uniform or non-Cartesian grid.

divUpdaterCoordProd Parameters

The divUpdaterCoordProd kind takes the `lowerBounds` and `upperBounds` parameters of *FieldUpdater*, as well as the following parameters:

`readFields` (*required string vector*)
A vector containing a single element, the name of a vector field of which to take the divergence.

`writeFields` (*required string vector*)
A vector containing a single element, the name of a scalar field to update.

`skipFirst` (*optional integer*, *default = 0 (false)*)
Set this flag to `1` (true) to skip the first component of the vector field, i.e. to use components 1–3 in the divergence rather than 0–2. This should be used when taking the divergence of **J** in a **`SumRhoJ`** charge-current field. A **`SumRhoJ`** field has four components, the first of which is ρ; therefore the first component of this field must be skipped to get to the **J** components.

`factor` (*optional float*, *default = 1*)
A factor that multiplies the end result after taking the divergence.

`includeCylAxis` (*optional integer*, *default = 0 (false)*)
Set this to true (`1`) if $r = 0$ is included in this update. To obtain the correct behavior at the axis, specify two separate **`divUpdaterCoordProd`** updaters; one with the axis and one without.

Example divUpdaterCoordProd Block

```
<FieldUpdater  Div_k_Grad_T>
    kind=divUpdaterCoordProd
    lowerBounds=[0 1 0]
    upperBounds=[NZ NR NPHI]
    readFields=[HeatFlux]
    writeFields=[dTemp]
<FieldUpdater>
<FieldUpdater  Div_k_Grad_T_axis>
    kind=divUpdaterCoordProd
    includeCylAxis=1
    lowerBounds=[0 0 0]
    upperBounds=[NZ 1 NPHI]
    readFields=[HeatFlux]
    writeFields=[dTemp]
</FieldUpdater>
```

dummyUpdater

Works with VSimBase, VSimEM, VSimPD, VSimPA, and VSimMD licenses.

MultiField updater that does not update values. Use of dummyUpdater may be helpful if you require specific timing and messaging of fields. In the update step, you can specify a `toDtFrac` such that during that update step, nothing is done to the field except for modifying its time.

dummyUpdater Parameters

The dummyUpdater kind requires the `lowerBounds` and `upperBounds` parameters of *FieldUpdater*, though they are ignored. It also takes the following parameter:

writeFields (*optional string vector*, *default = []*)
A vector containing the names of any number of fields. The updater will modify the times of these fields.

Example dummyUpdater Block

```
<FieldUpdater dummy>
  kind = dummyUpdater
  lowerBounds = [ 0   0   0]
  upperBounds = [NX  NY  NZ]
  writeFields = [activeEnvFld]
</FieldUpdater>
```

edgeToNodeVec

Works with VSimBase, VSimEM, VSimPD, VSimPA, and VSimMD licenses.

MultiField updater that interpolates field components between grid edges and nodes.

nodeToEdge Parameters

The edgeToNodeVec updater takes the `lowerBounds` and `upperBounds` parameters of *FieldUpdater*, as well as the following parameters:

readFields (*required string vector*)
A vector containing a single element, the name of the field to interpolate.

writeFields (*required string vector*)
A vector containing a single element, the name of field to update with the interpolated values.

nodeToEdge (*optional integer*, *default = 0 (false)*)
If `true`, the updater interpolates from nodes to edges. By default, the updater interpolates from edges to nodes.

Example edgeToNodeVec Block

```
<FieldUpdater nodalEupdate>
  kind                      = edgeToNodeVec
  lowerBounds               = [   0    0    0]
  upperBounds               = [NX1 NY1 NZ1]
  readFields                = [elecField]
  writeFields               = [nodalE]
</FieldUpdater>
```

epetraUpdater

Works with VSimBase, VSimEM, VSimPD, VSimPA, and VSimMD licenses.

Multifield updater that works by defining a matrix relationship, $Ax = b$, between two vectors, x and b, and the matrix A, e.g.,

$$\begin{bmatrix} A_{00} & A_{01} \\ A_{10} & A_{11} \end{bmatrix} \begin{bmatrix} F_1 \\ G_1 \end{bmatrix} = \begin{bmatrix} F_2 \\ G_2 \end{bmatrix}.$$

epetraUpdater Parameters

The epetraUpdater updater takes the `lowerBounds` and `upperBounds` parameters of *FieldUpdater*, as well as the following parameters:

`readFields` (*required string vector*)
A vector containing the names of fields to read. If multiple components of a field are read, then the field name must be repeated once for each component.

`readComponents` (*required integer vector*)
For each *`readFields`*, a component; the jth component of this vector is used for the jth *`readFields`* specified.

`writeFields` (*required string vector*)
A vector containing the names of fields to update. If multiple components of a field are written, then the field name must be repeated once for each component.

`writeComponents` (*required integer vector*)
For each field in *`writeFields`*, a component; the jth component of this vector is used for the jth *`writeFields`* specified.

`problemType` (*optional string*, *default = multiply*)
One of either `multiply` or `solve`. The update is either that one starts with x and gets b (multiply) or one starts with b and gets x (solve), with x and b as described above. If multiply, the readFields should describe x and if solve, the readFields should describe b.

`readFactors` (*optional float vector*)
Vector describing the factors by which to multiply the *`readFields`* prior to any updating. The vector can be any length; if the vector has fewer elements than the number of *`readFields`*, or is not specified, the values of the multiplicative factors default to `1`.

`writeFactors` (*optional float vector*)
Vector describing the factors by which to multiply the *`writeFields`* prior to writing. The vector can be any length; if the vector has fewer elements than the number of *`writeFields`*, or is not specified, the values of the multiplicative factors default to `1`.

`maxIters` (*optional integer*, *default = 100*)
maximum number of iterations to take for convergence to the desired residual.

`solver` (*required string*)
One of:

- `cg`
- `gmres`
- `cgs`
- `tfqmr`
- `bicgstab`

These are the iterative solvers (and one direct sovler) available in Aztec that can be used for solving a linear system of equations. Please refer to the Trilinos documentation for further details.

`scaling` (*optional string*, *default = none*)
One of:

- `none`
- `Jacobi`
- `row_sum`
- `sym_diag`
- `sym_row_sum`

Please refer to the Trilinos documentation for further details on these scalers.

`precond` (*optional string*, *default = dom_decomp_ilu*)
One of:

- `ml`
- `dom_decomp_ilu`
- `dom_decomp_ilut`
- `neumann`
- `ls`
- `jacobi`

Please refer to the Trilinos documentation for further details on these preconditioners.

`output` (*optional string*, *default = none*)
Desired level of output messages; one of:

- `all`
- `none`
- `warnings`
- `last`
- `brieflast`

`desiredResid` (*optional float*, *default = 1.e-9*)
The desired residual.

`diagFac` (*real*, *default = 1*)
This should be the value equivalent to the diagonal value on the matrix. diagFac is used to scale the equation.

`mlThreshold` (*optional float*, *default = 0*)
Use only when pre-conditioner is `ml`: weight at which the multi-level preconditioner considers two nodes to be connected. Weight is important when the problem involves cells with large aspect ratios.

`boundaryAtBottomInComponentDir` (*optional integer vector*, *default = []*)
If true, the boundary is at the bottom in the component direction. E.g., do not put in a stencil for Vx along the IX=0 plane, as that will be filled in by boundary conditions.

`boundaryAtBottomInNonComponentDir` (*optional integer vector*, *default = []*)
If true, the boundary is at the bottom in the non-component directions. E.g., do not put in a stencil for Vy and Vz along the IY=IZ=0 planes, as they will be filled in by boundary conditions.

boundaryAtTopInComponentDir (*optional integer vector*, *default = []*)
: If true, the boundary is at the top in the component direction. E.g., do not put in a stencil for Vx along the IX=0 plane, as that will be filled in by boundary conditions.

boundaryAtTopInNonComponentDir (*optional integer vector*, *default = []*)
: If true, the boundary is at the top in the non-component directions. E.g., do not put in a stencil for Vy and Vz along the IY=IZ=0 planes, as they will be filled in by boundary conditions.

MatrixFiller (*required parameter block*)
: To define the matrix A, you must use a matrixFiller block. At least one matrixFiller block is required for the epetraUpdater. matrixFiller parameters include:

  `kind` (required string) – one of:

  - `interior`
  - `cutcell`

  StencilElement (required parameter block): The MatrixFiller block also must contain at least one **StencilElement** block. **StencilElement** defines the non-zero values in a row of the matrix, and is described in *StencilElement*.

Example epetraUpdater Block

```
<FieldUpdater eyUpdate>

  kind = epetraUpdater
  problemType = multiply
  lowerBounds = [0 0 0]
  upperBounds = [NX NY NZ]
  readFields = [edgeElec faceMag faceMag]
  readComponents = [1 2 0] # 0 = Ey, 1 = Bz, 2 = Bx
  writeFields = [edgeElec]
  writeComponents = [1]
  boundaryAtBottomInComponentDir = [0]
  boundaryAtBottomInNonComponentDir = [1]
  boundaryAtTopInComponentDir = [1]
  boundaryAtTopInNonComponentDir = [1]

  <MatrixFiller eyupmat>
    kind=interior
    <StencilElement eyey>
      value = 1.
      minDim = 0
      cellOffset = [0 0 0]
      rowFieldIndex = 0
      columnFieldIndex = 0
    </StencilElement>
    <StencilElement eybxup>
    ...
    </StencilElement>
    ...
  </MatrixFiller>

</FieldUpdater>
```

faceToNodeVec

Works with VSimBase, VSimEM, VSimPD, VSimPA, and VSimMD licenses.

MultiField updater that interpolates field components between grid faces and nodes.

faceToNodeVec Parameters

The `faceToNodeVec` updater takes the `lowerBounds` and `upperBounds` parameters of *FieldUpdater*, as well as the following parameters:

`readFields` (*required string vector*)
A vector containing a single element, the name of the field to interpolate.

`writeFields` (*required string vector*)
A vector containing a single element, the name of field to update with the interpolated values.

`nodeToFace` (*optional integer*, *default = 0 (false)*)
If `true`, the updater interpolates from nodes to faces. By default, the updater interpolates from faces to nodes.

Example faceToNodeVec Block

```
<FieldUpdater nodalBupdate>
  kind                      = faceToNodeVec
  lowerBounds               = [  0    0    0]
  upperBounds               = [NX1 NY1 NZ1]
  readFields                = [magField]
  writeFields               = [nodalB]
</FieldUpdater>
```

fieldBinOpUpdater

Works with VSimBase, VSimEM, VSimPD, VSimPA, and VSimMD licenses.

MultiField updater that applies a mathematical operation on two fields (F and G) specified through the *`readFields`* parameter, and writes the result to the *`writeFields`* parameter.

fieldBinOpUpdater Parameters

The `fieldBinOpUpdater` takes the `lowerBounds` and `upperBounds` parameters of *FieldUpdater*, as well as the following parameters:

`readFields` (*required string vector*)
A vector of the names of the two fields on which to operate.

`writeFields` (*required string vector*)
A vector containing a single element, the name of field to update.

`binOp` (*required string*)
Operation to apply to the field; one of `add`, `subtract`, `multiply` or `divide`. Operations are:

```
add:        (aCoeff*Fj)+(bCoeff*Gj)
subtract:    (aCoeff*Fj)-(bCoeff*Gj)
multiply:    (aCoeff+Fj)*(bCoeff+Gj)
divide:       (aCoeff+Fj)/(bCoeff+Gj)
```

readComponents (*optional integer vector*)
The components to use in the operand fields.

writeComponent (*optional integer*)
The component to update.

Note: The *readComponents* and *writeComponent* parameters work together, and if one is specified, the other must be as well. If neither are specified, then all the components are updated. In that case, both of the *readFields* and the *writeFields* must all have the same number of components.

aCoeff (*optional float*)
Coefficient for first field (F, as described above). Default values:

```
add:        1.0
subtract:    1.0
multiply:    0.0
divide: 0.0.
```

bCoeff (*optional float*)
Coefficient for second field (G, as described above). Default values:

```
add:        1.0
subtract:    1.0
multiply:    0.0
divide: 0.0.
```

Example fieldBinOpUpdater Block

```
<FieldUpdater addUpdate> # aA + bB
  kind = fieldBinOpUpdater
  lowerBounds = [0 0 0]
  upperBounds = [NX1 NY1 NZ1]
  binOp = add
  aCoeff = 1.0
  bCoeff = 1.0
  readFields = [F1 F2]
  writeFields = [addField]
</FieldUpdater>
```

geometryUpdater

works with VSimEM and VSimMD licenses.

MultiField updater that evaluates a geometric quantity (pertaining to the individual cell) and sets (or adds or multiplies) the field component to that geometric quantity.

geometryUpdater Parameters

The `geometryUpdater` takes the `lowerBounds` and `upperBounds` parameters of *FieldUpdater*, as well as the following parameters:

operation (*required string*)
: One of:

```
set: Fj=g
```

```
add: Fj+=g
```

```
multiply: Fj*=g
```

geometricQuantity (*required string*)
: geometric quantity `g` (see operation) to be evaluated; one of:

volumeFraction: Fraction of the volume in the cell that is inside the specified **gridBoundary**.

surfaceOutwardNormal: Outward normal to the surface of the gridBoundary in a cell.

surfaceArea The surface area of the *queryGridBoundary* within the cell.

faceAreaFraction The fraction of the cell face (normal to *queryComponent*) that is inside the *queryGridBoundary*

edgeLineFraction The fraction of the cell edge (parallel to the *queryComponent* direction) that is inside the *queryGridBoundary*.

octantVolumeFraction The fraction of the cell octant that is inside the *queryGridBoundary*.

queryGridBoundary (*required string*)
: Name of the gridBoundary to be used to calculate the *geometricQuantity* (as opposed to the gridBoundary all updaters have, which determines which cells are updated by that updater).

queryComponent (*optional integer*, *default = 0*)
: Component of the data to be read for the `surfaceOutwardNormal`, `edgeLineFraction`, and `faceAreaFraction` quantities.

writeFields (*required string vector*)
: A vector containing a single element, the name of field to update with the geometric data.

writeComponents (*required integer vector*)
: The components of F to update. The geometric quantity will be evaluated for each component.

Example geometryUpdater Block

```
<FieldUpdater calcVolPlane>
  kind = geometryUpdater
  lowerBounds = [0 0 0]
  upperBounds = [NX NY NZ]
  operation = set
  geometricQuantity = volumeFraction
# update cell volume using queryGridBndry
  queryGridBoundary = plane
  writeFields = [VolFracPlane]
```

(continues on next page)

(continued from previous page)

```
  writeComponents = [0]
</FieldUpdater>
```

gradVecUpdater

Works with VSimBase, VSimEM, VSimPD, VSimPA, and VSimMD licenses.

Multifield updater that computes the gradient of a scalar field given by the *`readFields`* parameter and writes the resulting vector field to *`writeFields`*.

gradVecUpdater Parameters

The `gradVecUpdater` takes the `lowerBounds` and `upperBounds` parameters of *FieldUpdater*, as well as the following parameters:

`readFields` (*required string vector*)
A single element, the name of the scalar field for which to compute the gradient.

`writeFields` (*required string vector*)
A single element, the vector field to update with the computed gradient.

`factor` (*optional float*, *default = 1*)
Factor by which to multiply the field.

`differencing` (*optional string*, *default = forward*)
The direction in which to take the finite differences, one of:

- **`forward`:** Performs a gradient of a nodal field to compute the gradient on grid edges.
- **`backward`:** Performs a gradient of a cell-centered field to compute the gradient on grid faces.

Example gradVecUpdater Block

```
<FieldUpdater grad_T>
  kind        = gradVecUpdater
  lowerBounds = [ 0   0   0]
  upperBounds = [$NX+1$ $NY+1$ $NZ+1$]
  readFields  = [Temperature]
  writeFields = [HeatFlux]
</FieldUpdater>
```

gradVecUpdaterCoordProd

Works with VSimBase, VSimEM, VSimPD, VSimPA, and VSimMD licenses.

Multifield updater that computes the gradient of a scalar field given by the *`readFields`* parameter and writes the resulting vector field to *`writeFields`*, on a non-uniform or non-Cartesian grid.

gradVecUpdaterCoordProd Parameters

The `gradVecUpdaterCoordProd` takes the `lowerBounds` and `upperBounds` parameters of *FieldUpdater*, as well as the following parameters:

`readFields` (*required*, *string*, *vector*)
A single element, the name of the scalar field for which to compute the gradient.

`writeFields` (*required*, *string*, *vector*)
A single element, the vector field to update with the computed gradient.

`factor` (*optional float*, *default = 1*)
Factor by which to multiply the field.

`includeCylAxis` (*optional integer*, *default = 0 (false)*)
Set this to true (1) if $r = 0$ is included in the simulation. To obtain the correct behavior at the axis, specify two separate `gradVecUpdaterCoordProd` updaters; one with the axis and one without.

Example gradVecUpdaterCoordProd Block

```
<FieldUpdater gradPhi>
    kind=gradVecUpdaterCoordProd
    lowerBounds=[0 1]
    upperBounds=[NZ NR]
    readFields=[phi]
    writeFields=[edgeE]
    factor = 1.0
</FieldUpdater>
<FieldUpdater gradPhi_axis>
    kind=gradVecUpdaterCoordProd
    includeCylAxis=1
    lowerBounds=[0 0]
    upperBounds=[NZ 1]
    readFields=[phi]
    writeFields=[edgeE]
    factor = 1.0
 </FieldUpdater>
```

importFromFileUpdater

Works with VSimBase, VSimEM, VSimPD, VSimPA, and VSimMD licenses.

Updates a field from a file (e.g,. an hdf5 file). This updater sets the values of a field from a dataset in an appropriate file (e.g., a Vorpal field dump file).

importFromFileUpdater Parameters

The `importFromFileUpdater` takes the `lowerBounds` and `upperBounds` parameters of *FieldUpdater*, as well as the following parameters:

`writeFields` (*required string vector*)
A single element, the field to update with the imported data.

`fileName` (*required string*)
Name of the HDF5 file containing the data to read.

`dataset` (*required string*)
: Name of the HDF5 dataset to read from the file.

`component` (*optional integer*, *default = 0*)
: The component to read, if reading a single component. Either this parameter or *`writeComponents`* must be given, but not both.

`writeComponents` (*optional integer vector*)
: The components of the writeField to be set from the file. If given, the vector must have at least one element, and elements must be in sequence, e.g., [0 1 2] or [2 3], not [0 2], not [3 2]. If *`writeComponents`* is not given, the updater will update only the component corresponding to the *`component`* parameter. Either *`component`* or *`writeComponents`* must be given, but not both.

`datasetLowerBounds` (*optional integer vector*)
: See *`datasetUpperBounds`*.

`datasetUpperBounds` (*optional integer vector*)
: With *`datasetLowerBounds`*, the bounds describing the subset of the file's dataset (array) to be copied to the field; the region of cells includes the lower bound, but excludes the upper (as usual for lowerBounds and upperBounds in Vorpal). These bounds are specified by grid index, and must be of the same size as the upper/lowerBounds provided. So for example to import the first 10 cells of a 3 component field

lowerBounds = [0 0 0] upperBounds = [10 10 10] dataSetLowerBounds = [0 0 0 0] datasetUpperBounds = [10 10 10 3]

If you wanted to import the first 10 cells of a 3 component field into cells 10-20 of the destination field it would look like lowerBounds = [10 10 10] upperBounds = [20 20 20] dataSetLowerBounds = [0 0 0 0] datasetUpperBounds = [10 10 10 3]

The length in each dimension must correspond to the length in a spatial dimension (of the volume described by lower/upperBounds), in order. For example, if the updater updates a box in space of size $10 \times 12 \times 20$, then then the subset of the file's dataset may be $10 \times 1 \times 12 \times 20$ or $1 \times 10 \times 1 \times 12 \times 1 \times 12 \times 1$, but it may not be $12 \times 10 \times 20$ or $12 \times 12 \times 20$.

(In a serial simulation, the above correspondence is relaxed if the *`allowDatasetReshapeInSerial`* option is `true`.)

The compatibility of bounds is slightly altered if copyUniformInDir is set; in this case, the length of the dataset dimension corresponding to the direction specified by copyUniformInDir must be one, rather than the same as the length of the updater's bounds in that direction.

Default: include entire simulation domain + 1 in each dimension, and all components of the writeField (because that corresponds to Vorpal's field dump-files)

`copyUniformInDir` (*optional integer*)
: Allows a (lower-dimensional) file dataset to be copied to a field slice, after which those values are copied into adjacent field slices. For example, if a file contains a 2D dataset representing a function $f(x, y)$, setting `copyUniformDir = 2` would allow setting a 3D field $F(x, y, z) = f(x, y)$ for every z. When *`copyUniformInDir`* is specified, the dataset dimensions must correspond to the updater's bounds after the updater's bounds have been modified to have length 1 in the direction of copyUniformDir.

For example, suppose we have a 1D dataset with length 4 and values [10,20,30,40], and a 2+1-dimensional field with dimensions $4 \times 3 \times 1$ (a 4×3 scalar field in 2D). If the dataset bounds are `dataSetLowerBounds=[0]` and `datasetUpperBounds=[4]`, and the field bounds are `lowerBounds=[0,1]` and `upperBounds=[4,2]` (and `writeComponents=[0]`, the only choice for this scalar field), then the field will be set to:

```
0  0  0  0
10 20 30 40
0  0  0  0
```

(where the first dimension is horizontal, and the second vertical) but if field bounds are `[0,1]` and `[4,3]` and `copyUniformDir = 1`, then the field will be set to:

```
0  0  0  0
10 20 30 40
10 20 30 40
```

allowDatasetReshapeInSerial (*optional integer*, *default = 0 (false)*)
This option, available only for serial simulations, allows the datasetBounds to be reshaped to match the updater's bounds; for example, a 1D dataset with 12 elements might be used to update a 4×3 sub-array of the writeField. Use of this option is discouraged.

Example importFromFileUpdater Block

```
<FieldUpdater readB>
  kind = importFromFileUpdater
  fileName = sourceSim_B_5.h5
  dataset = "B"
  writeFields = [B]
  writeComponents = [0 1 2]
  lowerBounds = [0 0 0]
  upperBounds = [$NX+1$ $NY+1$ $NZ+1$]
</FieldUpdater>
```

lightFrameEnvelopeUpdater

Works with VSimPA license.

Multifield updater that updates the envelope fields in the laser envelope model. To use it effectively, you must use it with the *lightFrameEnvelopeMultiField* `kind` of MultiField. When using the envelope model with particles, the species should be of kind *envBoris*.

lightFrameEnvelope Updater Parameters

The `lightFrameEnvelopeUpdater` takes the `lowerBounds` and `upperBounds` parameters of *FieldUpdater*, as well as the following parameters:

readFields (*required string vector*)
Three field names: the active envelope field, the alternate field, and the susceptibility field. The envelope fields are complex scalars, with the real part in component 0 and the imaginary part in component 1.

writeFields (*required string vector*)
Two field names: the active and alternate envelope fields.

omega (*required float*)
Angular frequency of the laser, in Hz.

Solver (*required parameter block*)
The `lightFrameEnvelopeUpdater` requires a block of type `Solver` of any name. This block provides parameters for the linear solver used in the update.

In general, the possible values of solver block parameters correspond to the values of parameters of the AztecOO library used by Vorpal. String parameters are case insensitive and need not have the AZ_ prefix. Not all AztecOO parameters can be set from the input file, and the parameters in the example file have been found to work well. However, solver parameters that you may want to adjust include:

kind (optional string): Specifies the iterative solver available in Aztec to use for the linear system of equations of the envelope model. Please refer to the Trilinos documentation for further details. One of:

- `cg`
- `gmres`
- `cgs`
- `tfgmr`
- `bicgstab`

precond (optional string): Please refer to the Trilinos documentation for details regarding Trilinos preconditioners. The precond parameter specifies a Trilinos preconditioner; one of:

- `ml`
- `dom_decomp_ilu`
- `dom_decomp_ilut`
- `neumann`
- `ls`
- `jacobi`

output (optional string): specifies information to be printed. Possible values to specify level of output include:

- `all`
- `none`: suppress residual data
- `warnings`
- `last`
- `summary`

tolerance (optional float): Tolerance for solver convergence.

Example lightFrameEnvelopeUpdater Block

```
<FieldUpdater envUpdater>
  kind = lightFrameEnvelopeUpdater
  lowerBounds = [ 0   0   0]
  upperBounds = [NX  NY  NZ]
  readFields = [activeEnvFld altEnvFld chi]
  writeFields = [activeEnvFld altEnvFld]

  omega = OMEGA

  <Solver mySolver>
    kind = gmres
    precond = dom_decomp
    output = all
    tolerance = 1.e-08
  </Solver>
</FieldUpdater>
```

lightFrameEnvForceUpdater

Works with VSimPA license.

MultiField updater that computes the ponderomotive force from an envelope field in the Laser Envelope Model; see *lightFrameEnvelopeMultiField* for details.

lightFrameEnvForceUpdater Parameters

The `lightFrameEnvForceUpdater` takes the `lowerBounds` and `upperBounds` parameters of *FieldUpdater*, as well as the following parameters:

readFields (*required string vector*)
A single element, the name of the envelope field.

writeFields (*required string vector*)
A single element, the name of the resulting force field.

Example lightFrameEnvForceUpdater Block

```
<FieldUpdater forceUpdater>
  kind = lightFrameEnvForceUpdater
  lowerBounds = [ 0    0    0]
  upperBounds = [NX   NY   NZ]
  readFields = [activeEnvFld]
  writeFields = [forceFld]
</FieldUpdater>
```

linearApplyUpdater

Works with VSimBase, VSimEM, VSimPD, VSimPA, and VSimMD licenses.

Multifield updater that works by defining a matrix relationship, $Ax = b$, between two vectors, x and b, and the matrix A, e.g.,

$$\begin{bmatrix} A_{00} & A_{01} \\ A_{10} & A_{11} \end{bmatrix} \begin{bmatrix} F_1 \\ G_1 \end{bmatrix} = \begin{bmatrix} F_2 \\ G_2 \end{bmatrix}.$$

linearApplyUpdater Parameters

The `linearApplyUpdater` takes the `lowerBounds` and `upperBounds` parameters of *FieldUpdater*, as well as the following parameters:

minDim (*optional integer*, *default = 1*)
If the dimension of the simulation is less than `minDim`, this updater will not be applied.

readFields (*required string vector*)
A vector containing the names of fields to read. If multiple components of a field are read, then the field name must be repeated once for each component.

readComponents (*required integer vector*)
For each `readField`, a component; references to the jth `readField` will use the component specified in the jth element of this vector.

writeFields (*required string vector*)
: A vector containing the names of fields to update. If multiple components of a field are written, then the field name must be repeated once for each component.

writeComponents (*required integer vector*)
: For each `writeField`, a component; references to the jth `writeField` will use the component specified in the jth element of this vector.

writeEquationToFile (*optional integer, default = 0 (false)*)
: The matrix A, left-hand side vector x, and (after solution) unknown vector b, will be written out in MatrixMarket format when set to `true`.

matrixfiller (*required parameter block*)
: To define the matrix `A`, you must use a MatrixFiller block. At least one MatrixFiller block is required for the linearApplyUpdater.

vectorwriter (*required parameter block*)
: To define the left-hand side vector `x`, you must use a VectorWriter block. At least one VectorWriter block is required for the linearApplyUpdater.

vectorreader (*required parameter block*)
: To access the unknown vector `b`, you must use a VectorReader block. At least one VectorReader block is required for the linearApplyUpdater.

Example linearSolveUpdater Block

```
<FieldUpdater eyUpdate>

  kind = linearApplyUpdater
  lowerBounds = [0 0 0]
  upperBounds = [NX NY NZ]
  readFields = [edgeElec faceMag faceMag]
  readComponents = [1 2 0] # 0 = Ey, 1 = Bz, 2 = Bx
  writeFields = [edgeElec]
  writeComponents = [1]

  <MatrixFiller eyupmat>
    kind = interior
    <StencilElement eyey>
      value = 1.
      minDim = 0
      cellOffset = [0 0 0]
      rowFieldIndex = 0
      columnFieldIndex = 0
    </StencilElement>
    <StencilElement eybxup>
    ...
    </StencilElement>
    ...
  </MatrixFiller>

  <VectorWriter eyuplhs>
    kind = fieldVectorWriter
    minDim = 0
    readField = faceMag
    readComponent = 2
    lowerBounds = [0 0 0]
```

(continues on next page)

(continued from previous page)

```
    upperBounds = [NX NY NZ]
    component = 1
  </VectorWriter>

  <VectorReader eyupunk>
    kind = fieldVectorReader
    minDim = 0
    writeField = edgeElec
    writeComponent = 1
    lowerBounds = [0 0 0]
    upperBounds = [NX NY NZ]
  </VectorReader>

</FieldUpdater>
```

linearSolveUpdater

Works with VSimBase, VSimEM, VSimPD, VSimPA, and VSimMD licenses.

Multifield updater that works by defining a matrix relationship, $Ax = b$, between two vectors, x and b, and the matrix A, e.g.,

$$\begin{bmatrix} A_{00} & A_{01} \\ A_{10} & A_{11} \end{bmatrix} \begin{bmatrix} F_1 \\ G_1 \end{bmatrix} = \begin{bmatrix} F_2 \\ G_2 \end{bmatrix}.$$

linearSolveUpdater Parameters

The `linearSolveUpdater` takes the `lowerBounds` and `upperBounds` parameters of *FieldUpdater*, as well as the following parameters:

readFields (*required string vector*)
A vector containing the names of fields to read. If multiple components of a field are read, then the field name must be repeated once for each component.

readComponents (*required integer vector*)
For each `readField`, a component; references to the jth `readField` will use the component specified in the jth element of this vector.

writeFields (*required string vector*)
A vector containing the names of fields to update. If multiple components of a field are written, then the field name must be repeated once for each component.

writeComponents (*required integer vector*)
For each `writeField`, a component; references to the jth `writeField` will use the component specified in the jth element of this vector.

writeEquationToFile (*optional integer, default = 0 (false)*)
The matrix A, right-hand side vector b, and (after solution) unknown vector x, will be written out in MatrixMarket format when set to `true`.

MatrixFiller (required parameter block):

To define the matrix `A`, you must use a matrixFiller block. At least one matrixFiller block is required for the linearSolveUpdater.

VectorWriter (required parameter block):

To define the right-hand side vector b, you must use a VectorWriter block. At least one VectorWriter block is required for the linearSolveUpdater.

VectorReader (required parameter block):

To access the unknown vector x, you must use a VectorReader block. At least one VectorReader block is required for the linearSolveUpdater.

LinearSolver (required parameter block):

To solve the equation, you must use a LinearSolver block.

Example linearSolveUpdater Block

```
<FieldUpdater eyUpdate>

  kind = linearSolveUpdater
  lowerBounds = [0 0 0]
  upperBounds = [NX NY NZ]
  readFields = [edgeElec faceMag faceMag]
  readComponents = [1 2 0] # 0 = Ey, 1 = Bz, 2 = Bx
  writeFields = [edgeElec]
  writeComponents = [1]

  <MatrixFiller eyupmat>
    kind = interior
    <StencilElement eyey>
      value = 1.
      minDim = 0
      cellOffset = [0 0 0]
      rowFieldIndex = 0
      columnFieldIndex = 0
    </StencilElement>
    <StencilElement eybxup>
    ...
    </StencilElement>
    ...
  </MatrixFiller>

  <VectorWriter eyuprhs>
    kind = fieldVectorWriter
    minDim = 0
    readField = faceMag
    readComponent = 2
    lowerBounds = [0 0 0]
    upperBounds = [NX NY NZ]
    component = 1
  </VectorWriter>

  <VectorReader eyupunk>
    kind = fieldVectorReader
    minDim = 0
    writeField = edgeElec
    writeComponent = 1
    lowerBounds = [0 0 0]
    upperBounds = [NX NY NZ]
  </VectorReader>
```

(continues on next page)

(continued from previous page)

```
  <LinearSolver mySolver>
    kind = iterativeSolver
    <BaseSolver>
      kind = gmres
    </BaseSolver>
    <Preconditioner>
      kind = multigrid
      mgDefaults = SA
    </Preconditioner>
    tolerance = 1.e-10
    maxIterations = 1000
    output = 1
  </LinearSolver>

</FieldUpdater>
```

linIterUpdater

Works with VSimBase, VSimEM, VSimPD, VSimPA, and VSimMD licenses.

General translation-invariant linear Multifield updater. Depending on the specified parameter operation, linIterUpdater can perform the following updates for multiple fields: $\mathbf{E^1}$, ..., $\mathbf{E^N}$ simultaneously, where:

$$E^i_{b_i}(x,y,z,t+\Delta t) = \sum_j A_{ij} F^j_{b'_j}(x+m_j\Delta x, y+n_j\Delta y, z+p_j\Delta z, t)$$

$$E^i_{b_i}(x,y,z,t+\Delta t) = E^i_{b_i}(x,y,z,t) + \sum_j A_{ij} F^j_{b'_j}(x+m_j\Delta x, y+n_j\Delta y, z+p_j\Delta z, t)$$

$$E^i_{b_i}(x,y,z,t+\Delta t) = E^i_{b_i}(x,y,z,t) \sum_j A_{ij} F^j_{b'_j}(x+m_j\Delta x, y+n_j\Delta y, z+p_j\Delta z, t)$$

$$E^i_{b_i}(x,y,z,t+\Delta t) = \frac{E^i_{b_i}(x,y,z,t)}{\sum_j A_{ij} F^j_{b'_j}(x+m_j\Delta x, y+n_j\Delta y, z+p_j\Delta z, t)}$$

In each case, b_i is the user-chosen component of the field $\mathbf{E^i}$, b'_j is the user-chosen component of the field $\mathbf{F^j}$, m_j, n_j, and p_j correspond to the cellOffset `StencilElement` parameter (see *StencilElement*), and Δx, Δy, and Δz are the dimensions of the cell.

linIterUpdater updates cell-by-cell, based on an internal iterator approach (hence the name *iter*). linIterUpdater evaluates the right-hand-side of the equation (as shown above) for a cell, applies the updates to the left-hand-side of the equation for a cell, and then proceeds to the next cell. This is important to remember if a field is both a `readFields` and a `writeFields`.

The matrix $\mathbb{A}$ is described in the input file in such a way as to be compatible with other matrix solvers in Vorpal. There are a few differences, however, because with the linIterUpdater, the full matrix is never created, so the linIterUpdater can perform some extra operations to modify matrix elements at each time-step with negligible computation. Vorpal performs multiplication and division cell-by-cell, not using matrix multiplication.

You can choose to modify the matrix coefficients at each time-step. In many updates, for instance, you may prefer to multiply the coefficients by the time-step at each time-step, allowing for (usually only very slightly) varying time-steps. You are likely to find that this operation is too costly to implement in matrix updaters that construct the entire matrix. (Of course, such updaters can use matrices that are not translationally invariant, while the linIterUpdater cannot.)

linIterUpdater cannot perform matrix solves, so `rowFieldIndex` always refers to `writeFieldIndex`, but you must use the `rowFieldIndex` attribute. `.columnFieldIndex` refers to `.readFieldIndex`, meaning the index of fields in the writeFields and readFields lists. For example, if `readFields = [elecField elecField elecField SumRhoJ SumRhoJ SumRhoJ]` and `readComponents = [0 1 2 0 1 2]` then `columnFieldIndex = 3` refers to `SumRhoJ_0`.

linIterUpdater Parameters

The `linIterUpdater` takes the `lowerBounds` and `upperBounds` parameters of *FieldUpdater*, as well as the following parameters:

operation (*required string*)
One of:

- `set`: See Equation 1.
- `add`: See Equation 2.
- `multiply`: See Equation 3.
- `divide`: See Equation 4.

readFields (*required string vector*)
A vector containing the names of fields to read. If multiple components of a field are read, then the field name must be repeated once for each component.

readComponents (*required integer vector*)
For each *readFields*, a component; the jth component of this vector is used for the jth *readFields* specified.

writeFields (*required string vector*)
A vector containing the names of fields to update. If multiple components of a field are written, then the field name must be repeated once for each component.

writeComponents (*required integer vector*)
For each *writeFields*, a component; the jth component of this vector is used for the jth *writeFields* specified.

dtCoefficients (*optional float vector, default = [1.0 0.0]*)
Two components $[c_0\ c_1]$. The matrix coefficients will be multiplied by $(c_0 + c_1\Delta t)$ where Δt is the current time step. If c_1 is not specified it is assumed to be zero.

FieldMatrix (*required parameter block*)
Describes matrix $\mathbb{A}$. It must contain only *StencilElement* code blocks, each of which is used to describe an element of matrix $\mathbb{A}$.

Example linIterUpdater Block

```
<FieldUpdater ampereLinIterVec>
  kind = linIterUpdater
  operation = add
  lowerBounds = [0 0 0]
  upperBounds = [NX NY NZ]
  dtCoefficients = [0. 1.]
  readFields = [Binit Binit Binit SumRhoJ SumRhoJ SumRhoJ]
  readComponents = [0 1 2 1 2 3]
  writeFields = [EcalcLinIterVec EcalcLinIterVec EcalcLinIterVec]
  writeComponents = [0 1 2]
  <FieldMatrix dEdt>
```

(continues on next page)

(continued from previous page)

```
  # dEx / dt
    <StencilElement ExBz0>
      minDim = 2
      value = $LIGHTSPEED**2/DY$
        cellOffset = [0 0 0]
        columnFieldIndex = 2
        rowFieldIndex = 0
    </StencilElement>
    ...
  </FieldMatrix>
</FieldUpdater>
```

linPlasDielcUpdater

Works with VSimEM and VSimMD licenses.

Multifield updater that is used to describe linear plasma dielectric model for cold plasma. To use this updater, `readFields` should always have a background magnetic field defined in the `MultiField` block. Also, if damping is used the damping fields need to be added to `readFields`. In the update step, the nodal electric field must be specified in the `messageFields` parameter.

linPlasDielcUpdater Parameters

The `linPlasDielcUpdater` takes the `lowerBounds` and `upperBounds` parameters of *FieldUpdater*, as well as the following parameters:

nspecies (*required integer*)
An integer value that defines the number of species (e.g., `2` might refer to Krypton ions and electrons).

massNumbers (*required float vector*)
A vector of floating point values that defines the mass of the species. See the example block for common mass numbers.

chargeNumbers (*required float vector*)
A vector of floating point values that defines the charge of the species.

includeDamping (*optional integer, default = 0 (false)*)
Defines whether damping on the simulation will be used.

readFields (*required string vector*)
The names of the fields to use in this update: The background magnetic field, of offset *none*, a particle density field for each species, also of offset *none*, and if damping is included, a damping coefficient for each species.

writeFields (*required string vector*)
The fields to update: An electric field, of offset *none*, and a three-component current field for each species, also of offset *none*.

Example linPlasDielcUpdater Block

```
<FieldUpdater  plasmaDielectric>
    kind = linPlasDielcUpdater
    lowerBounds = [0 0 0]
    upperBounds = [NX1 NY1 NZ1]
```

(continues on next page)

(continued from previous page)

```
    nspecies = 2
    # common mass numbers: e=0.5486e-3, He=4.00, N=14.00, Ne=20.18, Ar=39.95, Kr=83.80
    massNumbers = [83.80  0.5486e-3]
    chargeNumbers = [1.0  -1.0]
    includeDamping = 0
    readFields = [B0  density0ion1  density0electron]
    writeFields = [nodalE  linearJion1  linearJelectron]
  </FieldUpdater>
```

malUpdater

Works with VSimEM, VSimPA, and VSimMD licenses.

The malUpdater is a MultiField updater that uses isotropic electric and magnetic damping profiles to absorb an incident wave in a slab. These Matched Absoring Layers (MALs) are more stable than Perfectly Matched Layers (PMLs), which use the same electric and magnetic damping profiles, but are anisotropic. MALs are typically applied to a specific boundary of the grid.

malUpdater Parameters

The malUpdater takes the `lowerBounds` and `upperBounds` parameters of *FieldUpdater*, as well as the following parameters:

upperOrLower (*required string vector*)
Whether slab is at upper or lower bounds of specified direction: upper or lower

numOrDenom (*required string vector*)
How to apply value of frac. If "denom" the amplitude is 1/(1 + frac). If "numer" the amplitude is 1-frac.

writeField (*required string vector*)
A vector containing a single element, the name of the field to update.

dir (*required integer*)
Direction along which damping profile varies and the slab normal: 0, 1, or 2

frac (*required float*)
The peak damping amplitude: 0.5 is suggested, typical range is 0.125 to 2.0.

power (*required float*)
The damping profile goes as x**power: 3.0 is suggested, typical range is 1.0 to 4.0.

Example malUpdater Block

```
<FieldUpdater malexlow>
  kind = malUpdater
  lowerBounds = [0  0  0]
  upperBounds = [MAL_THICKNESS  NY  NZ]
  upperOrLower = lower
  numOrDenom = denom
  writeField = E
  dir = 0
  frac = 0.5
  power = 3.0
</FieldUpdater>
```

multiDielectricUpdater

Works with VSimEM license.

This FieldUpdater sets the inverse permittivity to multiple gridBoundary shapes.

multiDielectricUpdater Parameters

The `multiDielectricUpdater` takes the `lowerBounds` and `upperBounds` parameters of *FieldUpdater*, as well as the following parameters:

permittivityField (*required string*)
: The name of the vector field to write the inverse permittivity to.

backgroundPermittivity (*required float*)
: The value of the permittivity for all other space that is not given in a DielectricShape block.

DielectricShape (*optional block*)
: Any number of DielectricShape blocks may be included. These blocks allow for setting the permittivity inside a pre-defined GridBoundary. If GridBoundaries specified in multiple DielectricShape blocks overlap, the permittivity in the last such block is used; thus later blocks overwrite earlier ones. The parameters of a DielectricShape block are described in *DielectricShape Block*.

multiDielectricUpdater Example

```
<FieldUpdater setInvEps>
  kind = multiDielectricUpdater
  lowerBounds = [0  0  0]
  upperBounds = [46  51  51]
  permittivityField = invEps

  <DielectricShape cylinder0Unionsphere0Shape>
    boundary = cylinder0Unionsphere0
    permittivity = 9.9
  </DielectricShape>

  <DielectricShape cube0Minuscylinder00Shape>
    boundary = cube0Minuscylinder00
    permittivity = 9.0
  </DielectricShape>

  backgroundPermittivity = 1.0
</FieldUpdater>
```

neutralBoltzmannUpdater

Works with VSimPD license.

MultiField updater that computes the electrostatic potential satisfying the Boltzmann relation. It reads a charge density field ρ and updates a potential field.

$$\phi = -T_0 \, ln\left(\frac{\rho}{n_0 q_0}\right) - \phi_0$$

neutralBoltzmannUpdater Parameters

The `neutralBoltzmannUpdater` takes the `lowerBounds` and `upperBounds` parameters of *FieldUpdater*, as well as the following parameters:

readFields (*required string vector*)
: A single element, the name of the field containing the charge density in component 0.

writeFields (*required string vector*)
: A single element, the name of field to update. Component 0 of this field will be updated with the electrostatic potential.

n0 (*required float*)
: Background density in #/m^3.

T0 (*required float*)
: Temperature (in eV).

q0 (*required float*)
: Charge of electron.

phi0 (*required float*)
: Constant offset to electrostatic potential.

mion (*required float*)
: (not used).

mindensity (*required float*)
: Lowest density allowed.

Example neutralBoltzmannUpdater Block

```
<FieldUpdater esSolve>
  kind = neutralBoltzmannUpdater

  n0 = ELECDENS
  T0 = ELECTEMPERATURE
  q0 = ELECCHARGE
  phi0 = 4.6
  mindensity = NP2C

  lowerBounds = [0 0 0]
  upperBounds = [NX NY NZ]
  readFields  = [rhoJ]
  writeFields = [phi]
</FieldUpdater>
```

open

Works with VSimEM, VSimPD, and VSimMD licenses.

Allows for outgoing waves on a Yee grid. This is only allowable as a boundary condition on the domain boundary.

open Parameters

The `open` MultiField updater takes the `lowerBounds` and `upperBounds` parameters of *FieldUpdater*, as well as the *local region modification parameters*. In addition, `open` takes the following parameters:

`readFields` (*required string vector*)
: A vector containing a single element, the name of the magnetic field to use in the open boundary comdition.

`writeFields` (*required string vector*)
: A vector containing a single element, the name of the electric field to update with appropriate values for an open boundary.

`component` (*required integer*)
: The component of the electric field to update.

`velOverC` (*required float*)
: Ratio of the wave velocity to the speed of light. velOverC can be negative to specify direction of propagation.

`normalDir` (*optional integer*)
: Propagation axis of the outgoing wave. If *`normalDir`* is not specified, Vorpal assumes that the axis will be the first simulated direction with an upper bound 1 greater than the lower bound.

`STFunc` (*required parameter block*)
: A parameter block of type *`STFunc`* (with any name) must be specified. This describes the anticipated functional form of the outgoing wave.

Example open Block

```
<FieldMultiUpdater leftOpenBC>
  kind=open
  lowerBounds = [0 -1 -1]
  upperBounds = [1   $YSIZE+1$   $ZSIZE+1$]
  readFields = [MagMultiField]
  writeFields = [ElecMultiField]
  components = [1 2]
  velOverC = -1.0
  <STFunc function>
    kind = expression
    expression = AMP * sin(OMEGA * t)
  </STFunc>
</FieldMultiUpdater>
```

permittivityUpdater

Works with a VSimEM license.

This is a MultiField updater that computes the effective tensor permittivity on the computational grid in the presence of a material interface. One specifies a region and the relative permittivities inside and outside the region, and the updater will compute the inverse physical permittivity tensor everywhere within `lowerBounds` and `upperBounds`, including the effective tensor at the interface.

permittivityUpdater Parameters

The `permittivityUpdater` takes the `lowerBounds` and `upperBounds` parameters of *FieldUpdater*, as well as the following parameters:

region (*required string*)
: The name of the `STRgn` block describing the dielectric geometry.

invPermittivityField (*required string*)
: The name of the inverse permittivity tensor field to update. It must have 6 components.

The permittivities can be specified as either scalars or tensors. To use scalars, use the following parameters:

insidePermittivity (*required float*)
: The relative dielectric permittivity inside the volume given in the *region* parameter.

outsidePermittivity (*required float*)
: The relative dielectric permittivity outside the volume given in the *region* parameter.

To use tensor permittivities, use the following parameters:

insidePermittivityDiag (*required float vector*)
: The diagonal components of the relative permittivity inside the volume given in the *region* parameter. This vector must have 3 elements, which correspond to $[\epsilon_{xx}, \epsilon_{yy}, \epsilon_{zz}]$.

insidePermittivityOffDiag (*optional float vector, default = [0. 0. 0.]*)
: The off-diagonal components of the relative permittivity inside the volume given in the *region* parameter. This vector must have 3 elements, which correspond to $[\epsilon_{yz}, \epsilon_{zx}, \epsilon_{xy}]$. This updater enforces a symmetric permittivity tensor, i.e. $\epsilon_{ji} = \epsilon_{ij}$.

outsidePermittivityDiag (*required float vector*)
: The diagonal components of the relative permittivity outside the volume given in the *region* parameter. This vector must have 3 elements, which correspond to $[\epsilon_{xx}, \epsilon_{yy}, \epsilon_{zz}]$.

outsidePermittivityOffDiag (*optional float vector, default = [0. 0. 0.]*)
: The off-diagonal components of the relative permittivity outside the volume given in the *region* parameter. This vector must have 3 elements, which correspond to $[\epsilon_{yz}, \epsilon_{zx}, \epsilon_{xy}]$. This updater enforces a symmetric permittivity tensor, i.e. $\epsilon_{ji} = \epsilon_{ij}$.

Example permittivityUpdater Block

```
<FieldUpdater setEpsilon>
  kind = permittivityUpdater
  lowerBounds = [  0    0    0]
  upperBounds = [NX1  NY1  NZ1]
  region = sphere
  invPermittivityField = invEpsilon
  insidePermittivityDiag = [REL_EPS_XX  REL_EPS_YY  REL_EPS_ZZ]
  insidePermittivityOffDiag = [REL_EPS_YZ  REL_EPS_XZ  REL_EPS_XY]
  outsidePermittivityDiag = [1.0  1.0  1.0]
</FieldUpdater>
```

phaseShiftVecUpdater

Works with VSimBase, VSimEM, VSimPD, VSimPA, and VSimMD licenses.

A `phaseShiftVecUpdater` is a MultiField updater that treats the two specified fields as the real and imaginary components of a single complex field, and applies a phase shift. Thus, if **F** and **G** are the fields and ϕ is the phase shift, the operation is

$$\begin{bmatrix}\mathbf{F}\\ \mathbf{G}\end{bmatrix} \mapsto \begin{bmatrix}\mathbf{F}\cos\phi - \mathbf{G}\sin\phi\\ \mathbf{F}\sin\phi + \mathbf{G}\cos\phi\end{bmatrix}.$$

Therefore, if $\mathbf{E} = \mathbf{F} + i\mathbf{G}$, then the operation is $\mathbf{E} \mapsto e^{i\phi}\mathbf{E}$.

This updater is useful for applying a phase shift across a periodic boundary to enforce a specific Bloch wavevector in a system with a periodic geometry.

phaseShiftVecUpdater Parameters

The `phaseShiftVecUpdater` takes the `lowerBounds` and `upperBounds` parameters of *FieldUpdater*, as well as the *local region modification parameters*. In addition, `phaseShiftVecUpdater` takes the following parameters:

minDim (*optional integer*, *default = 1*)
If the dimension of the simulation is less than *minDim*, this updater will not be applied.

writeFields (*required string vector*)
The two fields to update. The first is the real part of the complex field (corresponding to $\mathbf{F}$ above) and the second is the imaginary part ($\mathbf{G}$).

phase (*required float*)
The phase to use in the phase shift transformation, corresponding to ϕ in the description above.

Example phaseShiftVecUpdater Block

```
<FieldUpdater leftPhaseShift_E>
  kind = phaseShiftVecUpdater
  lowerBounds = [-1   0   0]
  upperBounds = [ 0  NY  NZ]
  cellsToUpdateBelowDomain = [1 1 1]
  cellsToUpdateAboveDomain = [1 1 1]
  writeFields = [elecField elecFieldI]
  phase = -X_PHASE
</FieldUpdater>
```

poissonUpdater

Works with VSimBase, VSimEM, VSimPD, VSimPA, and VSimMD licenses.

Multifield updater that solves Poisson's equation for an electric potential Φ given a charge density ρ. This implements the equation

$$\nabla^2\Phi = \frac{\rho}{f};$$

here f is an arbitrary factor that defaults to $-\epsilon_0$.

poissonUpdater Parameters

The `poissonUpdater` takes the `lowerBounds` and `upperBounds` parameters of *FieldUpdater*, as well as the following parameters:

readFields (*required string vector*)
A single element, the name of the charge density field.

writeFields (*required string vector*)
A single element, the name of the potential field to compute.

solver (*optional string*)
Please refer to the Trilinos documentation for details regarding Trilinos iterative solves available in Aztec. The solver parameter specifies an iterative solve that can be used for solving a linear system of equations; one of:

- `cg`
- `gmres`
- `cgs`
- `tfqmr`
- `bicgstab`

output (*optional string*)
Desired level of output messages; one of:

- `all`
- `none`
- `warnings`
- `last`
- `brieflast`

scaling (*optional string*)
Please refer to the Trilinos documentation for details regarding Trilinos scalers. The scaling parameter specifies a Trilinos scaler value; one of:

- `none`
- `Jacobi`
- `row_sum`
- `sym_diag`
- `sym_row_sum`

maxIters (*optional integer*, *default = 100*)
Maximum number of iterations to take for convergence to the desired residual.

factor (*optional float*, *default = -epsilon_0*)
Factor by which to divide ρ.

desiredResid (*optional float*, *default = 1.e-9*)
Specifies residual.

Example poissonUpdater Block

```
<FieldUpdater esSolve>
  kind = poissonUpdater
  lowerBounds = [0 0 0]
  upperBounds = [NX1 NY1 NZ1]
  readFields = [rho]
  writeFields = [phi]
  output = none
</FieldUpdater>
```

potentialUpdater

Works with VSimEM, VSimPD, and VSimMD licenses.

An updater that calculates the potential due to a charge distribution field.

potentialUpdater Parameters

The `potentialUpdater` MultiField updater takes the `lowerBounds` and `upperBounds` parameters of *FieldUpdater*. In addition, **`potentialUpdater`** takes the following parameters:

`sourceLowerBounds` (*required int vector*)
The lower bounds of the region containing sources.

`sourceUpperBounds` (*required int vector*)
The upper bounds of the region containing sources.

`direction` (*required integer*)
The direction associated with the update region.

`factor` (*required float*)
Multiplicative factor for scaling of potential.

`readField` (*required string vector*)
A vector containing a single element, the name of the field to use in the potentialUpdater.

`writeField` (*required string vector*)
A vector containing a single element, the name of the field to update with appropriate values for a potentialUpdater.

`readComponent` (*optional integer*)
The component of the read field to be used.

`writeComponent` (*optional integer*)
The component of the write field to write to.

`gammaVec` (*optional float vector*)
Relativistic gamma used for scaling of lengths.

`minDim` (*optional integer*)
The minimum dimensionality for which to apply the updater.

Example potentialUpdater Block

```
<FieldUpdater psiZHiBoundaryCalculation>
  kind = potentialUpdater
  lowerBounds = [0 0 176]
  upperBounds = [129 177 177]
  minDim = 3
  readField = kappa
  readComponent = 0
  writeField = psi
  writeComponent = 0
  direction = 2
  sourceLowerBounds = [0 0 0]
  sourceUpperBounds = [129 177 177]
  restoreTimeFromField = phi
```

(continues on next page)

(continued from previous page)

```
  factor = 112940906675.81473
  gammaVec = [490.2367906066536 1.0 1.0]
</FieldUpdater>
```

setEpsilonUpdater

Works with VSimEM and VSimMD licenses.

Fills the symmetric inverseEpsilon field (6 components, xx, yy, zz, xy=yx, yz=zy, zx=xz) from the given dielectric functions.

When a cell is partially filled (as determined by subsampling the functions over the cell), the dielectric is averaged appropriately over the cell. The number of samples per cell should increase as the simulation resolution increases, to maintain accuracy.

The setEpsilonUpdater requires one `writeFields` (the `invEpsilon` field with six (6) components) and no (0) `readFields`.

To define the dielectric constant, use **`STFunc`** blocks with the name `dielectricConstNM` where *N* and *M* represent the components [X Y Z]. A total of 6 blocks (components) are needed. Vorpal will first look for XY, YZ, and ZX, and use these values if they are present. However, if, XY is missing, Vorpal will use YX instead. If both XY and YX are present, Vorpal will use XY and ignore YX.

To define the surface normals (the vector normal to the dielectric interface at any location (x,y,z) that is within a cell length of the interface), use an STFunc block with a name `surfNormalN` where *N* represents the component [X Y Z].

setEpsilonUpdater Parameters

The `setEpsilonUpdater` takes the `lowerBounds` and `upperBounds` parameters of *FieldUpdater*, as well as the following parameters:

`writeFields` (*required string vector*)
A single element, the name of the 6-component inverse permittivity tensor field to update.

`samplesPerCell` (*optional integer vector, default = [1 1 1]*)
Number of locations (in each direction, x, y, and z) at which the dielectric is sampled in each cell. Although the default is `[1 1 1]`, usually `[7 7 7]` or higher would be a better choice.

`dielectricConstXX` (*required parameter block*)
This parameter block, which must be of type **`STFunc`**, specifies a function describing the xx component of the relative permittivity.

`dielectricConstYY` (*required parameter block*)
Similar to `dielectricConstXX`, but describing the yy component.

`dielectricConstZZ` (*required parameter block*)
Similar to `dielectricConstXX`, but describing the zz component.

`dielectricConstXY` (*required parameter block*)

and

`dielectricConstYX` (*required parameter block*)
This parameter block, which must be of type **`STFunc`**, specifies a function describing the xy component of the relative permittivity. At least one of these parameter blocks must be specified; if both are, `dielectricConstXY` is used and `dielectricConstYX` is ignored.

dielectricConstYZ (*required parameter block*)

and

dielectricConstZY (*required parameter block*)
Similar to `dielectricConstXY` and `dielectricConstYX`, but describing the yz component. If both are specified, `dielectricConstYZ` is used and `dielectricConstZY` is ignored.

dielectricConstZX (*required parameter block*)

and

dielectricConstXZ (*required parameter block*)
Similar to `dielectricConstXY` and `dielectricConstYX`, but describing the zx component. If both are specified, `dielectricConstZX` is used and `dielectricConstXZ` is ignored.

surfNormalX (*required parameter block*)
This parameter block, which must be of type **STFunc**, specifies a function describing the x component of the unit vector normal to the dielectric interface at the point on the interface closest to the position of the function argument. This value must be valid for any position argument within one grid cell of the interface.

surfNormalY (*required parameter block*)
Similar to `surfNormalX`, but describing the y component of the surface normal unit vector.

surfNormalZ (*required parameter block*)
Similar to `surfNormalX`, but describing the z component of the surface normal unit vector.

Example dielectricConst STFunc Block

```
<STFunc dielectricConstXZ>
    kind = expression
    expression = 1.+H(R^2-(x-X0)^2-(y-Y0)^2-(z-Z0)^2)*(REL_EPS_ZZ-1)
</STFunc>
```

Example surfNormal STFunc Block

```
<STFunc surfNormalX>
    kind = expression
    expression = x-X_SPHERE
</STFunc>
```

smooth1D

Works with VSimBase, VSimEM, VSimPD, VSimPA, and VSimMD licenses.

Updater applies a multiplicative 1D digital filter of the form $\begin{bmatrix} a & b & a \end{bmatrix}$ across a 1D grid for the specified component of a specified field. **smooth1D** is typically used for a single pass smoothing of the current. Only one field component can be smoothed. This updater can be applied to a 1D, 2D or 3D grid for which smoothDir will give the specific axis on which to apply smooth1D in a 1D fashion.

smooth1D Parameters

The `smooth1D` updater takes the `lowerBounds` and `upperBounds` parameters of *FieldUpdater*, as well as the following parameters:

readFields (*required string vector*)
A single element, the name of the field to smooth.

writeFields (*required string vector*)
A single element, the name of the field to update with the smoothed values.

aFac (*optional float*, *default = 0.25*)
The off-center stencil element value.

bFac (*optional float*, *default = 0.5*)
The center stencil element value.

smoothDir (*optional integer*, *default = 0*)
The direction over which to smooth.

smoothComp (*optional integer*, *default = 0*)
The field component to smooth.

Example smooth1D Block

```
<FieldUpdater smthPhiTau>
    kind = smooth1D
    lowerBounds = [1 0 0]
    upperBounds = [NX NY NZ]
    readFields = [phiTauFld]
    writeFields = [phiTauFldSmth]
    aFac = 0.25
    bFac = 0.5
    smoothDir = 0
    smoothComp = 0
</FieldUpdater>
```

STFuncUpdater

Works with VSimBase, VSimEM, VSimPD, VSimPA, and VSimMD licenses.

MultiField updater that performs one of the following operations on a field **F** and an **STFunc** f:

$$F_j(x, y, z, t + \Delta t) = f(x, y, z, t) \tag{3.6}$$

$$F_j(x, y, z, t + \Delta t) = F_j(x, y, z, t) + f(x, y, z, t) \tag{3.7}$$

$$F_j(x, y, z, t + \Delta t) = F_j(x, y, z, t) f(x, y, z, t) \tag{3.8}$$

For updating a single component of **F**, the updater is straightforward; the operation is specified by the operation parameter.

However, for updating multiple components of **F**, if different components of **F** are located at different places (e.g., for a Yee electric field, E_x and E_y are located at different places within a mesh cell), STFunc is evaluated only once for all components; STFunc is evaluated at the position of the component given by the component parameter of the updater and the field offset of field **F**. In this case, the components of **F** that are updated are those given by the parameter *writeComponents*.

If you want to have f evaluated at the location of each component then you must use a separate **STFuncUpdater** definition for each component (or otherwise update all components using **FieldMultiUpdater**).

STFuncUpdater Parameters

The `STFuncUpdater` takes the `lowerBounds` and `upperBounds` parameters of *FieldUpdater*, as well as the following parameters:

`minDim` (*optional integer*, *default = 1*)
If the dimension of the simulation is less than *`minDim`*, this updater will not be applied.

`operation` (*required string*)
One of:

- Set: See (3.6).
- Add: See (3.7).
- Multiply: See (3.8).

`writeFields` (*required string vector*)
A single element, the name of the field to update.

`component` (*optional integer*, *default = 0*)
Component j of $\mathbf{F}$ to be updated. However, if *`writeComponents`* is not `[0]`, the component, along with the field offset of $\mathbf{F}$, determines the location in each cell where the function $\mathbf{f}$ will be evaluated.

`writeComponents` (*optional integer vector, default = [0]*)
List of components of $\mathbf{F}$ that will be updated.

Note: The **`STFunc`** f will be evaluated only once per cell for all components, and the location within each cell at which f is evaluated is determined by the *`component`* parameter.

`dtCoefficients` (*optional float vector, default = [1.0 0.0]*)
Two components $[c_0\ c_1]$: the function will be multiplied by $(c_0 + c_1\Delta t)$, where Δt is the current time step. If c_1 is not specified it is assumed to be zero.

`STFunc` (*required parameter block*)
A parameter block of type **`STFunc`** (with any name) must be specified. This describes f.

Example STFuncUpdater block

```
<FieldUpdater initialPulse>
  kind = STFuncUpdater
  lowerBounds = [NX_BEGIN NY_BEGIN NZ_BEGIN]
  upperBounds = [NX_END NY_END NZ_END]
  operation = set
  writeFields = [elecField]
  component = 2
  <STFunc unimportantName> # a gaussian
    kind = expression
    expression = E_AMP * exp(-0.5 * ( (x-X_INIT)^2 + (y-Y_INIT)^2 \
        + (z-Z_INIT)^2 ) / WIDTH_INIT^2)
  </STFunc>
</FieldUpdater>
```

unaryFieldOpUpdater

Works with VSimBase, VSimEM, VSimPD, VSimPA, and VSimMD licenses.

Multifield updater that performs one of the following operations on a field $\mathbf{F}$, another field $\mathbf{G}$, and a **STFunc** f:

$$F_j(x, y, z, t + \Delta t) = f(x, y, z, t)G_j(x, y, z, t) \tag{3.9}$$

$$F_j(x, y, z, t + \Delta t) = F_j(x, y, z, t) + f(x, y, z, t)G_j(x, y, z, t) \tag{3.10}$$

$$F_j(x, y, z, t + \Delta t) = F_j(x, y, z, t)f(x, y, z, t)G_j(x, y, z, t) \tag{3.11}$$

$$F_j(x, y, z, t + \Delta t) = \frac{F_j(x, y, z, t)}{f(x, y, z, t)G_j(x, y, z, t)} \tag{3.12}$$

For updating a single component of $\mathbf{F}$, the updater is straightforward; the operation is specified by the operation parameter.

However, for updating multiple components of $\mathbf{F}$, if different components of $\mathbf{F}$ are located at different places (e.g., for a Yee electric field, E_x and E_y are located at different places within a mesh cell), STFunc is evaluated only once for all components; STFunc is evaluated at the position of the component given by the component parameter of the updater and the field offset of field $\mathbf{F}$. In this case, the components of $\mathbf{F}$ that are updated are those given by the parameter *writeComponents*.

If you want to have f evaluated at the location of each component, you must use a separate **STFuncUpdater** definition for each component (or otherwise update all components using **FieldMultiUpdater**).

unaryFieldOpUpdater Parameters

The `unaryFieldOpUpdater` takes the `lowerBounds` and `upperBounds` parameters of *FieldUpdater*, as well as the following parameters:

operation (*required string*)
One of:

- Set: See (3.9).
- Add: See (3.10).
- Multiply: See (3.11).
- Divide: See (3.12).

readFields (*required string vector*)
A single element, the name of the field to use in the operation ($\mathbf{G}$, above).

writeFields (*required string vector*)
A single element, the name of the field to update ($\mathbf{F}$, above).

component (*optional integer*, *default = 0*)
Component j of $\mathbf{F}$ to be updated. However, if *writeComponents* is not `[0]`, the component, along with the field offset of $\mathbf{F}$, determines the location in each cell where the function **f** will be evaluated.

readComponents (*optional integer vector*)
List of components of $\mathbf{G}$ used to update $\mathbf{F}$, e.g., if *writeComponents* = `[0 1 3]` and *readComponents* = `[2 1 3]`, then $F_0 = fG_2$, $F_1 = fG_1$, and $F_3 = fG_3$, where f is evaluated at the location of component. If this is not specified, it defaults to the value of *writeComponents*.

writeComponents (*optional integer vector, default = [0]*)
List of components j of $\mathbf{F}$ that will be updated.

Note: The **STFunc** f will be evaluated only once per cell for all components, and the location within each cell at which f is evaluated is determined by the *component* parameter.

`bumpReadIter` (*integer vector, default = [0 0 0]*)
: Number of cells in each simulated dimension to offset the location where G is written from the location where F is read.

`dtCoefficients` (*optional float vector, default = [1.0 0.0]*)
: Two components $[c_0\ c_1]$: the function f will be multiplied by $(c_0 + c_1\Delta t)$, where Δt is the current time step. If c_1 is not specified it is assumed to be zero.

`gridBoundary` (*optional string*)
: If provided, only components on the interior of the specified GridBoundary will be updated. The method to define the interior is given in the `interiorness` parameters. If this parameter is provided, then the field specified in *`writeFields`* must have `offset = none` or `offset = center`.

`interiorness` (*optional string*, *default = cellcenter*)
: If the *`gridBoundary`* parameter is specified, this is the method the used to determine whether a component is *interior* to the boundary. The behavior depends on the `.offset` specified in the updated *Field*.

One of:

`cellcenter`: If `offset = none`, then a cell is considered interior if its node is adjacent to at least one cell with center inside the boundary.

If `offset = center`, then a cell is considered interior if its center is inside the boundary.

`deymittra`: If `offset = none`, then a cell is considered interior if all nodes adjacent to (i.e. displaced by a single edge from) its node are inside the boundary.

This *`interiorness`* option cannot be specified with `offset = center`.

`STFunc` (required parameter block)

A parameter block of type **`STFunc`** (with any name) must be specified. This describes f.

userFuncUpdater

Works with VSimBase, VSimEM, VSimPD, VSimPA, and VSimMD licenses.

Sets field values to the result of a user-given function.

The FieldUpdater of `kind = userFuncUpdater` is a very flexible updater that sets a field to the result of a user-given <Expression> (see *expression*). The expression takes as arguments field values and spatial positions, as well as time and time step.

The userFuncUpdater can be used to do many different things; to understand how it works in some simple cases, it may be easier to skip to the examples at the end of this section.

The userFuncUpdater places *iterators* at the first cell of each read/writeField. The iterators are then moved from cell to cell; each iterator keeps track of its position and the field value at that position. As the iterators are collectively moved from cell to cell, an update is performed at each cell. An iterator knows its cell index, as well as the exact spatial location of the field value corresponding to that cell index (taking into account the field offset; e.g., the Yee electric field is usually located at the center-edges of a cell).

A userFuncUpdater has one iterator for every writeField, and one for every readField. The userFuncUpdater sets the field-values through the iterators on its writeFields according to a function of, among other things, the iterator positions, and the values of the readFields at their iterators. It then moves all the iterators by one cell and does the same thing, etc.

A userFuncUpdater can also has one or more readScalars. These scalars are read only. They can be used as variables in the function expression. userFuncUpdater can not update any scalars. To updater a scalar, one should use userFuncScalarUpdater.

A userFuncUpdater specifies a function (an Expression) that takes as arguments the values of the read fields, the positions of the read- and write-Field iterators, as well as the time and time step; and the function returns a vector with one value for each writeField (at each cell, each writeField is set to the corresponding value).

userFuncUpdater Parameters

The `userFuncUpdater` takes the `lowerBounds` and `upperBounds` parameters of *FieldUpdater*, as well as the following parameters:

`readFields` (*optional string vector*, *default = []*)
: The names of the fields to read. A field name can be repeated if multiple iterators are used from a single field.

`readScalars` (*optional string vector*, *default = []*)
: The names of the scalars to read. A scalar name can be directly used as a variable in a userFunc expression.

`writeFields` (*required string vector*)
: The names of the fields to write. A field name can be repeated if multiple iterators are used from a single field.

`readComponents` (*optional integer vector*)
: The component used for each readField. If any *`readFields`* are specified, this must be as well, and have the same number of elements as *`readFields`*.

`writeComponents` (*required integer vector*)
: The component used for each writeField. This must have the same length as *`writeFields`*.

`readItersShiftInX` (*optional integer vector*)
: This has the same length as *`readFields`*; by default it is all zeros. With this specified, each *`readFields`* iterator is shifted by the corresponding number of cells in the x direction. For example, `readItersShiftInX = [1 0]` will shift the iterators on the first *`readFields`* by one cell in the $+x$ direction, and won't shift the iterators on the second *`readFields`* in x.

`readItersShiftInY` (*optional integer vector*)
: Similar to *`readItersShiftInX`*, but shifts iterators in y. If the y direction is not simulated, this is ignored.

`readItersShiftInZ` (*optional integer vector*)
: Similar to *`readItersShiftInX`*, but shifts iterators in z. If the z direction is not simulated, this is ignored.

`readFieldVarNames` (*optional string vector*)
: The variable names given to the value of each field–this is how the field values will be referenced in the Expression. If any *`readFields`* are specified, this must be as well, and have the same number of elements as *`readFields`*. Each name must be unique; some names are forbidden, such as `t` and `dt` and `n`.

`readFieldPosVariables` (*optional string vector*)
: The names given to the variables representing the position of the corresponding readField value; each such variable is an N-dimensional vector, where N is the dimension of the simulation. However, `notUsed` means that the corresponding field position is not used in the Expression; ignoring a position in this way can speed up computation. If `pos` is the variable name given to a certain readField, then the variable `pos` (that appears in the Expression) will be an N-dimensional vector. For example, in an expression, the x position would be accessed by `select(pos, 0)` and the y position (in 2D or 3D simulation) by `select(pos, 1)`. If given, this parameter must be a vector with the same length as *`readFields`*. If not given, all elements are set to `notUsed`.

`writeFieldPosVariables` (*optional string vector*)
: Analogous to *`readFieldPosVariables`*, but for *`writeFields`*.

`updateFunction` (*required parameter block*)
: A parameter block of type **`Expression`** and name *`updateFunction`* is required to define the function used to update the *`writeFields`*. This block takes the same parameters as an *expression UserFunc*, except for `Input` blocks and the `inputOrder` parameter. The input variables for the `expression` are defined by the

above parameters, so they are not to be specified in the *updateFunction* block. In addition, the expression can take the following input valiables:

t: The simulation time (actually, the time to which the updater been told to update its writeFields; c.f. toDtFrac in a MultiField UpdateStep).

dt: The most recent time step (Δt).

n: The current simulation step (an integer).

(These names are reserved—they cannot be used as variable names in `readFieldVarNames`, `readFieldPosVariables`, or `writeFieldPosVariables`.)

The Expression must return a vector with the same length as the number of writeFields—each writeField value will be set to the corresponding component of the return vector.

userFuncUpdater Examples

Here is a userFuncUpdater to set a scalar field F to a function $f(x, y, z, t)$:

```
<FieldUpdater setF>
  kind = userFuncUpdater
  lowerBounds = [0 0 0]
  upperBounds = [NX NY NZ]
  writeFields = [F]
  writeComponents = [0]
  writeFieldPosVariables = [Fpos] # just a choice of name
  maxNumEvals = 64
  <Expression updateFunction>
    $ X = select(Fpos, 0) # makes the function more readable
    $ Y = select(Fpos, 1)
    expression = sin(kx * X + ky * Y + kz * select(Fpos, 2) - omega * t)
    <Term kx>
      kind = constant
      value = [10.]
    </Term>
    <Term ky>
      kind = constant
      value = [20.]
    </Term>
    <Term kz>
      kind = constant
      value = [3.]
    </Term>
    <Term omega>
      kind = constant
      value = [ $ LIGHTSPEED * math.sqrt(10^2 + 20^2 + 3^2) $ ]
    </Term>
  </Expression>
</FieldUpdater>
```

The above could be done equally well with a STFuncUpdater. However, setting a vector field **F** to a field can be done better with a **`userFuncUpdater`**, since it can evaluate the function at a different position for each component:

```
<FieldUpdater setF>
  kind = userFuncUpdater
  lowerBounds = [0 0 0]
  upperBounds = [NX NY NZ]
```

(continues on next page)

(continued from previous page)

```
  writeFields = [F F F]
  writeComponents = [0 1 2]
  writeFieldPosVariables = [FxPos FyPos FzPos]
  maxNumEvals = 64
  <Expression updateFunction>
    kind = expression
    $ XPHASE = kx * select(FxPos,0) + ky * select(FxPos, 1) + kz * select(FxPos, 2)
    $ YPHASE = kx * select(FyPos,0) + ky * select(FyPos, 1) + kz * select(FyPos, 2)
    $ ZPHASE = kx * select(FzPos,0) + ky * select(FzPos, 1) + kz * select(FzPos, 2)
    expression = sin( vector(XPHASE, YPHASE, ZPHASE) - omega * t)
    <Term kx>
      kind = constant
      value = [10.]
    </Term>
    <Term ky>
      kind = constant
      value = [20.]
    </Term>
    <Term kz>
      kind = constant
      value = [3.]
    </Term>
    <Term omega>
      kind = constant
      value = [ $ LIGHTSPEED * math.sqrt(10^2 + 20^2 + 3^2) $ ]
    </Term>
  </Expression>
</FieldUpdater>
```

Here is a **userFuncUpdater** that multiplies a 3-vector field $\mathbf{G}$ by Δt and adds it to a 3-vector field $\mathbf{F}$ (i.e., $\mathbf{F} \mapsto \mathbf{F} + \mathbf{G}\Delta t$):

```
<FieldUpdater addGdtToF>
  kind = userFuncUpdater
  lowerBounds = [0 0 0]
  upperBounds = [NX NY NZ]
  readFields = [F F F G G G]
  readComponents = [0 1 2 0 1 2]
  readFieldVarNames = [Fx Fy Fz Gx Gy Gz]
  writeFields = [F F F]
  writeComponents = [0 1 2]
  maxNumEvals = 64
  <Expression updateFunction>
    expression = vector(Fx, Fy, Fz) + dt * vector(Gx, Gy, Gz)
  </Expression>
</FieldUpdater>
```

Here is an example of a **userFuncUpdater** that uses the values of a History to perform its update (see *historyFunc*):

```
# A field updater that sets field values to the x-index of the cell
# with the (most recent) maximum value of a field, by using a History
# that finds that x-index.  Of course, one could also use a History
# that calculates voltage to use as feedback.
<FieldUpdater setFieldToIndexOfMax>
  kind = userFuncUpdater
  lowerBounds = [0 0 0]
```

(continues on next page)

(continued from previous page)

```
  upperBounds = [2 2 2]
  writeFields = [G]
  writeComponents = [0]
  <Expression updateFunction>
    # Presumably the input file defines a <History max> that records
    # the location of the maximum field value;
    # This HistoryFunc gets the x-value of that location.
    <UserFunc iOfLastMax>
      kind = historyFunc
      index = [0] # x-index of cell
      history = max
    </UserFunc>
    # get (1000*) x-index of cell with most recent maximum value of field F
    expression = 1000 * iOfLastMax(0)
  </Expression>
</FieldUpdater>
```

yeeAmpereDielVecUpdater

Works with VSimEM license.

Multifield updater that updates all three (3) components of the electric field according to Ampere's Law, but takes into account the inverse dielectric tensor given by a 6-component inverse permittivity field. You can set this inverse permittivity field with a *permittivityUpdater* or *setEpsilonUpdater*.

yeeAmpereDielVecUpdater Parameters

The `yeeAmpereDielVecUpdater` takes the `lowerBounds` and `upperBounds` parameters of *FieldUpdater*, as well as the following parameters:

readFields (*required string vector*)
The names of the three fields to use in the update. These are given, for instance, as `readFields = [yeeB source invEpsilon]`, where:

yeeB: Name of the Yee magnetic field.

source: This is the name of a field containing the current source. This can either be a 3-component field contining just the current, or a 4-component field containing the current in its last three components.

invEpsilon: 6-component symmetric inverse permittivity field, with components.

$$\left((\epsilon^{-1})_{xx}, (\epsilon^{-1})_{yy}, (\epsilon^{-1})_{zz}, (\epsilon^{-1})_{yz}, (\epsilon^{-1})_{zx}, (\epsilon^{-1})_{xy}\right).$$

This can be created with *permittivityUpdater* or *setEpsilonUpdater*.

writeFields (*required string vector*)
A single element, the name of the electric field to update.

dtCoefficients (*optional float vector, default = [1. 0.]*)
Two components $[c_0\ c_1]$ as defined in the equation above. The result of the updater will be multiplied by $(c_0 + c_1\Delta t)$, where Δt is the current time step.

yeeAmpereDielVecUpdater Example

```
<FieldUpdater yeeAmpere>
  kind = yeeAmpereDielVecUpdater
  lowerBounds = [NX_BEGIN  NY_BEGIN  NZ_BEGIN]
  upperBounds = [NX_END    NY_END    NZ_END]
  readFields = [magField  J  invEpsilon]
  writeFields = [elecField]
</FieldUpdater>
```

yeeAmpereUpdater

Works with VSimBase, VSimEM, VSimPD, VSimPA, and VSimMD licenses.

MultiField updater that updates an electric field, defined on grid edges, according to the standard second-order Yee leapfrog algorithm, using a magnetic field (on faces), and optionally a current field. This current field can be a 3-vector or the last three components of a 4-vector.

yeeAmpereUpdater Parameters

The `yeeAmpereUpdater` takes the `lowerBounds` and `upperBounds` parameters of *FieldUpdater*, as well as the *global region modification parameters* and *local region modification parameters*. In addition, **`yeeAmpereUpdater`** takes the following parameters:

`readFields` (*required string vector*)
A vector of either one or two strings. The first string is the name of magnetic field, and if provided, the second is the name of the current field to add to the result (multiplied by the specified factors).

`writeFields` (*required string vector*)
A vector contining a single element, the name of the electric field to update.

`useVecUpdater` (*optional integer, default = false*)
If true, the updater will update all three components of the electric field, beginning with the specified *`component`*. The updated field must therefore have at least `component` $+ 3$ components.

`component` (*optional integer, default = 0*)
The field component to update, or if `useVecUpdater` is `true`, the first field component to update.

`readFieldCompShifts` (*optional integer vector, default = [0 1]*)
This vector must have the same number of elements as *`readFields`*. It specifies the amount by which to increment the component indices of the first field and the (optional) second field. For example, if a magnetic field is represented by components 3–5 of the field `EandB`, then to calculate the curl of that magnetic field, one would specify `readFields = [EandB]` and `readFieldCompShifts = [3]`. If the second (current) field is not a 4-vector, then it is proper to set `readFieldCompShifts = [0 0]`.

`gridBoundary` (*optional string*)
If provided, only components on the interior of the specified GridBoundary will be updated. The method to define the interior is given in the `interiorness` parameters.

`interiorness` (*optional string, default = cellcenter*)
If the `gridBoundary` parameter is specified, this is the method the used to determine whether a component is *interior* to the boundary. The behavior depends on the `offset` specified in the updated *Field*. One of:

- **`cellcenter:`** If `offset = none`, or `offset = edge4v` and `component = 0`, then a cell is considered interior if its node is adjacent to at least one cell with center inside the boundary.

If `offset = edge`, or `offset = edge4v` and `component` is not `0`, then a cell is considered interior if the edge specified by *`component`* is adjacent to at least one cell with center inside the boundary.

If `offset = face`, then a cell is considered interior if the face specified by *`component`* is adjacent to at least one cell with center inside the boundary.

If `offset = center`, then a cell is considered interior if its center is inside the boundary.

- **`deymittra`** If `offset = none`, or `offset = edge4v` and `component = 0`, then a cell is considered interior if all nodes adjacent to (i.e. displaced by a single edge from) its node are inside the boundary.

  If `offset = edge`, or `offset = edge4v` and `component` is not `0`, then a cell is considered interior if the edge specified by *`component`* has at least one adjacent node inside the boundary, and that edge is not ignored by the Dey-Mittra algorithm given the *`dmFrac`* parameter specified in the *`gridBoundary`*.

  If `offset = face`, then a cell is considered interior if all nodes adjacent to the face specified by *`component`* are inside the boundary.

  This *`interiorness`* option cannot be specified with `offset = center`.

- **`.dmnodal`** This *`interiorness`* option is identical to `deymittra`.

`lowerSkinDepth` (*optional integer vector*)
Specifies the number of skin cells, in each direction, on the lower end of the local domain. The cells in the skin are updated before the fields specified as *`messageFields`* in the *UpdateStep or InitialUpdateStep block* are messaged. If not specified, the skin depth will be determined automatically.

`upperSkinDepth` (*optional integer vector*)
Specifies the number of skin cells, in each direction, on the upper end of the local domain. If not specified, the skin depth will be determined automatically.

Example yeeAmpereUpdater Block

```
<FieldUpdater yeeAmpere>
  kind = yeeAmpereUpdater
  lowerBounds = [NX_BEGIN NY_BEGIN NZ_BEGIN]
  upperBounds = [NX_END NY_END NZ_END]
  readFields = [yeeB]
  writeFields = [yeeE]
  useVecUpdater = true
</FieldUpdater>
```

yeeFaradayUpdater

Works with VSimBase, VSimEM, VSimPD, VSimPA, and VSimMD licenses.

Updates a magnetic field, defined on grid faces, according to the standard second-order Yee leapfrog algorithm, using an electric field defined on grid edges.

yeeFaradayUpdater Parameters

The `yeeFaradayUpdater` takes the `lowerBounds` and `upperBounds` parameters of *FieldUpdater*, as well as the *global region modification parameters* and *local region modification parameters*. In addition,

yeeFaradayUpdater takes the following parameters:

readFields (*required string vector*)
A single element, the electric field to use in performing the update.

writeFields (*required string vector*)
A vector contining a single element, the name of the magnetic field to update.

useVecUpdater (*optional integer*, *default = 0 (false)*)
If true, the updater will update all three components of the magnetic field, beginning with the specified *component*. The updated field must therefore have at least `component` $+ 3$ components.

component (*optional integer*, *default = 0*)
The field component to update, or if *useVecUpdater* is `true`, the first field component to update.

gridBoundary (*optional string*)
If provided, only components on the interior of the specified GridBoundary will be updated. The method to define the interior is given in the `interiorness` parameters.

interiorness (*optional string*, *default = cellcenter*)
If the *gridBoundary* parameter is specified, this is the method the used to determine whether a component is *interior* to the boundary. The behavior depends on the `offset` specified in the updated *Field*. One of:

- `cellcenter`:

  If `offset = none`, or `offset = edge4v` and `component = 0`, then a cell is considered interior if its node is adjacent to at least one cell with center inside the boundary.

  If `offset = edge`, or `offset = edge4v` and `component` is not `0`, then a cell is considered interior if the edge specified by *component* is adjacent to at least one cell with center inside the boundary.

  If `offset = face`, then a cell is considered interior if the face specified by *component* is adjacent to at least one cell with center inside the boundary.

  If `offset = center`, then a cell is considered interior if its center is inside the boundary.

- `deymittra`

  If `offset = none`, or `offset = edge4v` and `component = 0`, then a cell is considered interior if all nodes adjacent to (i.e. displaced by a single edge from) its node are inside the boundary.

  If `offset = edge`, or `offset = edge4v` and `component` is not `0`, then a cell is considered interior if the edge specified by *component* has at least one adjacent node inside the boundary, and that edge is not ignored by the Dey-Mittra algorithm given the `dmFrac` parameter specified in the `gridBoundary`.

  If `offset = face`, then a cell is considered interior if all nodes adjacent to the face specified by `component` are inside the boundary.

  This `interiorness` option cannot be specified with `offset = center`.

- `dmnodal`

  This *interiorness* option is identical to `deymittra`.

lowerSkinDepth (*optional integer vector*)
Specifies the number of skin cells, in each direction, on the lower end of the local domain. The cells in the skin are updated before the fields specified as `messageFields` in the *UpdateStep or InitialUpdateStep block* are messaged. If not specified, the skin depth will be determined automatically.

upperSkinDepth (*optional integer vector*)
Specifies the number of skin cells, in each direction, on the upper end of the local domain. If not specified, the skin depth will be determined automatically.

Example yeeFaradayUpdater Block

```
<FieldUpdater yeeFaraday>
  kind = yeeFaradayUpdater
  lowerBounds = [NX_BEGIN NY_BEGIN NZ_BEGIN]
  upperBounds = [NX_END NY_END NZ_END]
  readFields = [yeeE]
  writeFields = [yeeB]
  useVecUpdater = true
</FieldUpdater>
```

3.7.4 FieldMultiUpdater Block

FieldMultiUpdater

A `FieldMultiUpdater` block is similar to a `FieldUpdater` block, but always specifies multiple components of a field to update using the `components` parameter.

In some cases, the `FieldMultiUpdater` parameters correspond to the `FieldUpdater` block of the same `kind`, but with the `components` integer parameter replaced by the integer vector `components`. In that case, the `FieldMultiUpdater` creates as many occurances of the corresponding `FieldUpdater` as elements in the `components` parameter. For example, a `FieldMultiUpdater` with `components` equal to `[0 1 2]` creates three occurrences of the corresponding `FieldUpdater`. There will therefore be three update passes through the updated cells, one for each specified component. The cells updated in each pass may differ due to any *global region modification parameters*, or, if the updater has a region specification, depending on how that region is treated for each component.

The updater `kind` with corresponding `FieldMultiUpdater` blocks are:

- *curlUpdater*
- *curlUpdaterCoordProd*
- *open*
- *STFuncUpdater*
- *unaryFieldOpUpdater*
- *yeeAmpereUpdater*
- *yeeFaradayUpdater*

There are also updates that can only be specified using `FieldMultiUpdater`. They are:

- *amperePmlUpdater*
- *fieldSqrDiagUpdater*
- *perfDispPmlUpdater*
- *yeeConductorUpdater*

FieldMultiUpdater Parameters

All *FieldMultiUpdater* blocks take the `lowerBounds` and `upperBounds` parameters of *FieldUpdater*, as well as:

`kind` (*required string*)
Type of updater; one of:

- *amperePmlUpdater*
- *curlUpdater*
- *curlUpdaterCoordProd*
- *fieldSqrDiagUpdater*
- *open*
- *perfDispPmlUpdater*
- *STFuncUpdater*
- *unaryFieldOpUpdater*
- *yeeAmpereUpdater*
- *yeeConductorUpdater*
- *yeeFaradayUpdater*

`components` (*required integer vector*)
The components that should be updated.

FieldMultiUpdater Kinds

amperePmlUpdater

This implements the standard (as in Taflove) PML algorithm for an Ampere-type Maxwell update.

amperePmlUpdater Parameters

The **`amperePmlUpdater`** takes the `lowerBounds` and `upperBounds` parameters of *FieldUpdater* and the `components` parameter of *FieldMultiUpdater*, as well as the following parameters:

`minDim` (*optional integer*, *default = 1*)
If the dimension of the simulation is less than *`minDim`*, this updater will not be applied.

`readFields` (*required string vector*)
A vector containing a single element, the magnetic field to use in the update.

`writeFields` (*required string vector*)
A vector contining a two elements, the electric field and auxiliary displacement fields to update.

`sigmaX` (*optional parameter block*)
An **`STFunc`** block specifying the functional form of the conductivity for absorbtion of waves propagating in the x direction.

`sigmaY` (*optional parameter block*)
An **`STFunc`** block specifying the functional form of the conductivity for absorbtion of waves propagating in the y direction.

sigmaZ (*optional parameter block*)
An **STFunc** block specifying the functional form of the conductivity for absorbtion of waves propagating in the z direction.

Note: At least one of *sigmaX*, *sigmaY*, or *sigmaZ* must be specified. A `.sigma` function should be specified for every direction in which the PML region borders the edge of the simulation domain, e.g. if the region for this updater is along the y direction, then *sigmaY* should be specified.

kappaX (*optional parameter block*)
Effective dielectric constant for the PML absorption in the x direction. Making this greater than 1 can improve absorption of oblique-incidence waves without disturbing the analytically perfect matching. However, the best parameters to use are problem-dependent and users are encouraged to experiment.

kappaY (*optional parameter block*)
Effective dielectric constant for the PML absorption in the y direction. Making this greater than 1 can improve absorption of oblique-incidence waves without disturbing the analytically perfect matching. However, the best parameters to use are problem-dependent and users are encouraged to experiment.

kappaZ (*optional parameter block*)
Effective dielectric constant for the PML absorption in the z direction. Making this greater than 1 can improve absorption of oblique-incidence waves without disturbing the analytically perfect matching. However, the best parameters to use are problem-dependent and users are encouraged to experiment.

Example amperePmlUpdater block

```
<FieldMultiUpdater frontPMLE>
  kind = amperePmlUpdater
  lowerBounds = [ 0          0   NZ_BEGIN]
  upperBounds = [NX  NY_BEGIN  NZ_END  ]
  readFields = [magField]
  writeFields = [elecField pmlAuxE]
  components = [0 1 2]
  minDim = 2
  <STFunc sigmaY>
    kind = expression
    expression = SIGMA_MAX * ((LY_BEGIN - y) / LY_PML)^PML_EXP
  </STFunc>
  <STFunc kappaY>
    kind = expression
    expression = 1. + (KAPPA_MAX - 1) * ((LY_BEGIN - y) / LY_PML)^PML_EXP
  </STFunc>
</FieldMultiUpdater>
```

fieldSqrDiagUpdater

Works with VSimBase, VSimEM, VSimPD, VSimPA, and VSimMD licenses.

`FieldMultiUpdater` that computes an integral over the sum of the squares of the specified fields, for each specified component; i.e., for each component i,

$$F_i = a \int \left[f_i^{(1)}(\mathbf{x})^2 + f_i^{(2)}(\mathbf{x})^2 + f_i^{(3)}(\mathbf{x})^2 + \ldots + f_i^{(n)}(\mathbf{x})^2 \right] d\mathbf{x}.$$

The fields $\mathbf{f}^{(1)}, \mathbf{f}^{(2)}, \mathbf{f}^{(3)}, \ldots, \mathbf{f}^{(n)}$ used in the integral are given by the *readFields* parameter. The calculation is written to the standard output of the run. fieldSqrDiagUpdater is useful for measuring field energy and reflected components.

fieldSqrDiagUpdater Parameters

The fieldSqrDiagUpdater takes the lowerBounds and upperBounds parameters of *FieldUpdater* and the .components parameter of *FieldMultiUpdater*, as well as the following parameters:

readFields (*required string vector*)
The names of the fields for which to compute the integral.

factor (*optional float*, *default = 1.0*)
Factor a by which to multiply the integral.

writePeriod (*optional integer*, *default = 0*)
Frequency which to write out the calculation. A *writePeriod* of 1 will write out the calculation at each time step.

Example fieldSqrDiagUpdater Block

```
<FieldMultiUpdater E_energy>
  kind = fieldSqrDiagUpdater
  writePeriod = 20
  factor = E_ENERGY_FACTOR
  components = [0 1 2]
  contractFromBottomInNonComponentDir = 1
  lowerBounds = [0 0 0]
  upperBounds = [NX_TOT NY_TOT NZ_TOT]
  readFields = [elecField]
</FieldMultiUpdater>
```

perfDispPmlUpdater

implements the standard (as in Taflove) PML algorithm for an Faraday-type Maxwell update, using the controlled dispersion operators.

perfDispPmlUpdater Parameters

The perfDispPmlUpdater takes the lowerBounds and upperBounds parameters of *FieldUpdater* and the .components parameter of *FieldMultiUpdater*, as well as the following parameters:

minDim (*optional integer*, *default = 1*)
If the dimension of the simulation is less than *minDim*, this updater will not be applied.

readFields (*required string vector*)
A vector containing a single element, the electric field to use in the update.

writeFields (*required string vector*)
A vector contining a two elements, the magnetic field and auxiliary magnetic fields to update.

sigmaX (*optional parameter block*)
An *STFunc* block specifying the functional form of the magnetic conductivity for absorbtion of waves propagating in the x direction.

sigmaY (*optional parameter block*)
An *STFunc* block specifying the functional form of the magnetic conductivity for absorbtion of waves propagating in the y direction.

sigmaZ (*optional parameter block*)
An *STFunc* block specifying the functional form of the magnetic conductivity for absorbtion of waves propagating in the z direction.

Note: At least one of *sigmaX*, *sigmaY*, or *sigmaZ* must be specified. A `.sigma` function should be specified for every direction in which the PML region borders the edge of the simulation domain, e.g. if the region for this updater is along the y direction, then *sigmaY* should be specified.

kappaX (*optional parameter block*)
Effective permeability constant for the PML absorption in the x direction. Making this greater than 1 can improve absorption of oblique-incidence waves without disturbing the analytically perfect matching. However, the best parameters to use are problem-dependent and users are encouraged to experiment.

kappaY (*optional parameter block*)
Effective permeability constant for the PML absorption in the y direction. Making this greater than 1 can improve absorption of oblique-incidence waves without disturbing the analytically perfect matching. However, the best parameters to use are problem-dependent and users are encouraged to experiment.

kappaZ (*optional parameter block*)
Effective permeability constant for the PML absorption in the z direction. Making this greater than 1 can improve absorption of oblique-incidence waves without disturbing the analytically perfect matching. However, the best parameters to use are problem-dependent and users are encouraged to experiment.

Example perfDispPmlUpdater block

```
<FieldMultiUpdater frontPMLB>
  kind = perfDispPmlUpdater
  lowerBounds = [ 0         0  NZ_BEGIN]
  upperBounds = [NX  NY_BEGIN  NZ_END  ]
  readFields = [elecField]
  writeFields = [magField pmlAuxB]
  components = [0 1 2]
  minDim = 2
  Delta = DELTA
  <STFunc sigmaY>
    kind = expression
    expression = SIGMA_MAX * ((LY_BEGIN - y) / LY_PML)^PML_EXP
  </STFunc>
  <STFunc kappaY>
    kind = expression
    expression = 1. + (KAPPA_MAX - 1) * ((LY_BEGIN - y) / LY_PML)^PML_EXP
  </STFunc>
</FieldMultiUpdater>
```

yeeConductorUpdater

Works with VSimEM and VSimMD licenses.

Multifield updater that solves the following versions of the Ampere and Faraday equations, in which i is one of x, y, or z, $\sigma_{e,i}$ is the electric conductivity in the i direction divided by ϵ_0, $\sigma_{m,i}$ is the magnetic conductivity in the i direction divided by μ_0. Both $\sigma_{e,i}$ and $\sigma_{m,i}$ have units of frequency. Also, $J_{e,i}$ and $J_{m,i}$ are the electric and magnetic currents in the i direction, respectively.

yeeConductorUpdater Ampere equation:

$$\epsilon_0 \frac{\partial E_i}{\partial t} = \frac{1}{\mu_0} (\nabla \times \mathbf{B})_i - J_{e,i} - \sigma_{e,i} E_i \tag{3.13}$$

yeeConductorUpdater Faraday equation:

$$\mu_0 \frac{\partial B_i}{\partial t} = -\mu_0 (\nabla \times \mathbf{E})_i - J_{m,i} - \sigma_{m,i} B_i \tag{3.14}$$

yeeConductorUpdater Parameters

The `yeeConductorUpdater` takes the `lowerBounds` and `upperBounds` parameters of *FieldUpdater*, the *global region modification parameters*, and the `.components` parameter of *FieldMultiUpdater*. In addition, **curlUpdater** takes the following parameters:

emType (*required string*)
One of:

ampere: Expects *writeFields* to be the yee electric field and *readFields* to be the yee magnetic field. *readFields* can also contain an optional 4-component electric charge/current field.

faraday: Reverses $\mathbf{E}$ and $\mathbf{B}$, and if specified, the optional 4-component field represents magnetic charge/current.

readFields (*required string vector*)
A vector of either one or two strings. The first string is the name of the field to take the curl of, and if provided, the second is the name of the charge/current field.

writeFields (*required string vector*)
A vector containing a single element, which is the name of the field to update.

sigma (*optional float*)
The uniform, isotropic conductivity.

sigma (*optional parameter block*)
A parameter block of type **STFunc** specifying the local isotropic conductivity.

sigmaX (*optional float*)
The uniform conductivity in the x direction.

sigmaX (*optional parameter block*)
A parameter block of type **STFunc** specifying the local conductivity in the x direction.

sigmaY (*optional float*)
The uniform conductivity in the y direction.

sigmaY (*optional parameter block*)
A parameter block of type **STFunc** specifying the local conductivity in the y direction.

sigmaZ (*optional float*)
The uniform conductivity in the z direction.

sigmaZ (*optional parameter block*)
A parameter block of type **STFunc** specifying the local conductivity in the z direction.

Note: For each conductivity parameter above, either the scalar parameter for the uniform conductivity, or the **STFunc** block for the local conductivity can be specified, but not both. If an isotropic conductivity is not specified, a componentwise *sigma* parameter must be specified for each updated component. If both an isotropic conductivity and componentwise conductivities are specified, the componentwise conductivities will be used for any components for which they are specified, and the isotropic conductivity will be used for components for which they are not.

Example yeeConductorUpdater Block

```
<FieldMultiUpdater waterAmpere>
  gridBoundary = universe
  kind = yeeConductorUpdater
  emType = ampere
  components = [0 1 2]
  contractFromBottomInNonComponentDir = 1
  lowerBounds = [NX_WATER_L 0 0]
  upperBounds = [NX_WATER_H NY NZ]
  readFields = [magField SumRhoJ]
  writeFields = [elecField]
  sigma = SIGMA_WATER # conductivity/epsilon_0
</FieldMultiUpdater>
```

3.7.5 MatrixFiller Block

MatrixFiller

MatrixFiller: A block used to fill a matrix.

MatrixFiller Kinds

kind (required string) – one of:

- *stencilFiller*
- *nodeStencilFiller*
- *stFuncStencilFiller*
- *coordProdSTFuncStencilFiller*

```
<MatrixFiller interior>
  kind = nodeStencilFiller
  minDim = 0
  lowerBounds = [  0   0   0]
  upperBounds = [NXP NYP NZP]
  componentBounds = [0 1]
  gridBoundary = sphere
  rowInteriorosity = [insideBoundary]
  colInteriorosity = [insideBoundary]
  StencilElementMacro(error_dd, 1.0, 0, [ 0  0  0], 0, 0)
</MatrixFiller>
```

MatrixFiller Kinds

stencilFiller

Works with VSimBase, VSimEM, VSimPD, VSimPA, and VSimMD licenses.

Fill matrix elements using `StencilElement` blocks.

stencilFiller Parameters

`minDim` (*integer*, *required*)
If NDIM < minDim, the filler will not be used to fill the matrix.

`component` (*integer*, *required if componentBounds not specified*)
If a single-component fill, the component over which this filler operates. The number of components is set by the updater, and valid component indices are in the range [0, numberUpdaterComponents).

`componentBounds` (*integer vector*, *required if component not specified*)
The component range over which this filler operates. The number of components is set by the updater, and valid component indices are in the range [0, numberUpdaterComponents).

`StencilElement` (*code block*, *required*)
`StencilElement` block. `StencilElement` defines the non-zero values in a row of the matrix.

nodeStencilFiller

Works with VSimBase, VSimEM, VSimPD, VSimPA, and VSimMD licenses.

Fill matrix elements for node-centered fields when a grid boundary is present.

StencilElement Parameters

`minDim` (*integer*, *required*)
If NDIM < minDim, the filler will not be used to fill the matrix.

`component` (*integer*, *required if componentBounds not specified*)
If a single-component fill, the component over which this filler operates. The number of components is set by the updater, and valid component indices are in the range [0, numberUpdaterComponents).

`componentBounds` (*integer vector*, *required if component not specified*)
The component range over which this filler operates. The number of components is set by the updater, and valid component indices are in the range [0, numberUpdaterComponents).

`StencilElement` (*code block*, *required*)
The `StencilFiller` block must contain at least one `StencilElement` block. `StencilElement` defines the non-zero values in a row of the matrix.

`gridBoundary` (*string*, *default = universe*)
Name of the *GridBoundary* specifying the geometry.

`rowInteriorosity` (*string vector*, *required if grid boundary is not universe*)
Which node locations are valid for filling matrix rows. Options are any combination of `insideBoundary`, `cutByBoundary`, and `outsideBoundary`.

`columnInteriorosity` (*string vector*, *required if grid boundary is not universe*)
Which node locations are valid for columns within matrix rows. Options are any combination of `insideBoundary`, `cutByBoundary`, and `outsideBoundary`.

stFuncStencilFiller

Works with VSimBase, VSimEM, VSimPD, VSimPA, and VSimMD licenses.

Fill matrix elements using `stfuncstencilelement` blocks.

stFuncStencilFiller Parameters

`minDim` (*integer*, *required*)
If NDIM < minDim, the filler will not be used to fill the matrix.

`component` (*integer*, *required if componentBounds not specified*)
If a single-component fill, the component over which this filler operates. The number of components is set by the updater, and valid component indices are in the range [0, numberUpdaterComponents).

`componentBounds` (*integer vector*, *required if component not specified*)
The component range over which this filler operates. The number of components is set by the updater, and valid component indices are in the range [0, numberUpdaterComponents).

`STFunc` (code block, required)

The *`STFunc`* that will be calculated at the `.functionOffset` in each *`STFuncStencilElement`*, and multiplied by the `.value` in that *`STFuncStencilElement`*.

`STFuncStencilElement` (code block, required)

The `stFuncStencilFiller` block must contain at least one *STFuncStencilElement* block. *STFuncStencilElement* defines the non-zero values in a row of the matrix.

coordProdSTFuncStencilFiller

Works with VSimBase, VSimEM, VSimPD, VSimPA, and VSimMD licenses.

Fill matrix elements using *CoordProdSTFuncStencilElement* blocks.

coordProdSTFuncStencilFiller Parameters

`minDim` (*integer*, *required*)
If NDIM < minDim, the filler will not be used to fill the matrix.

`component` (*integer*, *required if componentBounds not specified*)
If a single-component fill, the component over which this filler operates. The number of components is set by the updater, and valid component indices are in the range [0, numberUpdaterComponents).

`componentBounds` (*integer vector*, *required if component not specified*)
The component range over which this filler operates. The number of components is set by the updater, and valid component indices are in the range [0, numberUpdaterComponents).

`STFunc` (*code block*, *required*)
The `STFunc` that will be calculated at the `functionOffset` in each *CoordProdSTFuncStencilElement*, and multiplied by the `value` in that *CoordProdSTFuncStencilElement*.

CoordProdSTFuncStencilElement (*code block*, *required*)
: The `coordProdSTFuncStencilFiller` block must contain at least one *CoordProdSTFuncStencilElement* block. *CoordProdSTFuncStencilElement* defines the non-zero values in a row of the matrix.

3.7.6 [CoordProd]STFuncStencilElement Block

STFuncStencilElement

For use with *stFuncStencilFiller* class in filling matrix entries.

STFuncStencilElement Parameters

minDim (*integer*, *default = 1*)
: If NDIM < minDim, the element will not be inserted into the matrix.

rowFieldIndex (*integer*, *required*)
: Row index of this element (always the `writeFieldIndex` in `linIterUpdater`).

columnFieldIndex (*integer*, *required*)
: Column index of this element (always the `readFieldIndex` in `linIterUpdater`).

cellOffset (*integer vector*, *required*)
: $[m_j \; n_j \; p_j]$, as described in the equations in *linIterUpdater*.

functionOffset (*real vector*, *required*)
: The offsets, in fractions of the grid spacing, at which the `STFunc` of the containing *stFuncStencilFiller* will be evaluated. So, for example, [0.5 0.0 0.0] corresponds to evaluation at a point half a grid-cell in the x-direction from the point being considered.

value (*real*, *required*)
: Value to be multiplied by the value of the `STFunc` of the containing *stFuncStencilFiller*.

CoordProdSTFuncStencilElement

For use with *coordProdSTFuncStencilFiller* class in filling matrix entries.

CoordProdSTFuncStencilElement Parameters

minDim (*integer*, *default = 1*)
: If NDIM < minDim, the element will not be inserted into the matrix.

rowFieldIndex (*integer*, *required*)
: Row index of this element (always the writeFieldIndex in linIterUpdater).

columnFieldIndex (*integer*, *required*)
: Column index of this element (always the `readFieldIndex` in `linIterUpdater`).

cellOffset (*integer vector*, *required*)
: $[m_j \; n_j \; p_j]$, as described in the equations in *linIterUpdater*.

functionOffset (*real vector*, *required*)
: The offsets, in fractions of the grid spacing, at which the `STFunc` of the containing *coordProdSTFuncStencilFiller* will be evaluated. So, for example, [0.5 0.0 0.0] corresponds to evaluation at a point half a grid-cell in the x-direction from the point being considered.

gridDirection (*integer*, *required*)
The direction for the (dual) grid parameters, such as grid spacing and face area, to be used in calculating the matrix entry. Each entry is multiplied by the area specified by *gridDirection* and *gridOffset*, and divided by the corresponding grid spacing and cell volume.

gridOffset (*integer vector*, *required*)
The offset for the (dual) grid parameters, such as grid spacing and face area, to be used in calculating the matrix entry. Each entry is multiplied by the area specified by *gridDirection* and *gridOffset*, and divided by the corresponding grid spacing and cell volume.

value (*real*, *required*)
Value to be multiplied by the value of the `STFunc` of the containing *coordProdSTFuncStencilFiller*.

3.7.7 VectorReader Block

VectorReader

VectorReader: A block used to fill entries of a single *Field* from values in a vector.

VectorReader Kinds

kind (*required string*)
One of:

- *fieldVectorReader*
- *nodeFieldVectorReader*

```
<VectorReader phiReader>
  kind = fieldVectorReader
  minDim = 0
  writeField = phi
  writeComponent = 0
  lowerBounds = [  0   0   0]
  upperBounds = [NXP NY NZ]
  componentBounds = [0 1]
</VectorReader>
```

VectorReader Kinds

fieldVectorReader

Works with VSimBase, VSimEM, VSimPD, VSimPA, and VSimMD licenses.

Fill entries of a single *Field* from values in a vector. A range of component indices from that field may be filled from the vector.

fieldVectorReader Parameters

minDim (*integer*, *required*)
If NDIM < minDim, the reader will not be used to fill the field.

`component` (*integer*, *required if componentBounds not specified*)
If a single-component fill, the component over which this reader operates. The number of components is set by the updater, and valid component indices are in the range [0, numberUpdaterComponents).

`componentBounds` (*integer vector*, *required if component not specified*)
The component range over which this reader operates. The number of components is set by the updater, and valid component indices are in the range [0, numberUpdaterComponents).

`writeFields` (*string*, *required*)
The names of the fields whose values will be filled from the vector. If multiple fields are given, only the first is used.

`writeComponent` (*integer*, *required if writeComponentBounds not specified*)
The component index of the field being written.

`writeComponentBounds` (*integer vector*, *required if writeComponent not specified*)
The range of component indices of the field being written.

`scaling` (*real*, *default = 1.0*)
A factor for scaling all vector values that are written to the field.

nodeFieldVectorReader

Works with VSimBase, VSimEM, VSimPD, VSimPA, and VSimMD licenses.

Fill entries of a single *Field* vector entries from values in a vector, taking into account the geometry specified by a *GridBoundary*.

nodeFieldVectorReader Parameters

`minDim` (*integer*, *required*)
If NDIM < minDim, the reader will not be used to fill the field.

`component` (*integer*, *required if componentBounds not specified*)
If a single-component fill, the component over which this reader operates. The number of components is set by the updater, and valid component indices are in the range [0, numberUpdaterComponents).

`componentBounds` (*integer vector*, *required if component not specified*)
The component range over which this reader operates. The number of components is set by the updater, and valid component indices are in the range [0, numberUpdaterComponents).

`writeFields` (*string*, *required*)
The names of the fields whose values will be written into from the vector. If multiple fields are given, only the first is used.

`writeComponent` (*integer*, *required if writeComponentBounds not specified*)
The component index of the field being written.

`writeComponentBounds` (*integer vector*, *required if writeComponent not specified*)
The range of component indices of the field being written.

`gridBoundary` (*string*, *required*)
Name of the *GridBoundary* specifying the geometry.

`interiorosity` (*string vector*, *required*)
Which node locations are valid for reading vector values. Options are any combination of `insideBoundary`, `cutByBoundary`, and `outsideBoundary`.

scaling (*real*, *default = 1.0*)
A factor for scaling all vector values that are written to the field.

3.7.8 VectorWriter Block

VectorWriter

A block used to fill vector entries from values in a single *Field*.

VectorWriter Kinds

kind (*required string*)
One of:

- *fieldVectorWriter*
- *nodeFieldVectorWriter*
- *stFuncVectorWriter*
- *stFuncNodeVectorWriter*

```
<VectorWriter loXInteriorRhoWriter>
  kind = fieldVectorWriter
  minDim = 0
  readField = rho
  readComponent = 0
  lowerBounds = [   1   1   0]
  upperBounds = [NXSC NYM NZ]
  componentBounds = [0 1]
</VectorWriter>
```

VectorWriter Kinds

fieldVectorWriter

Works with VSimBase, VSimEM, VSimPD, VSimPA, and VSimMD licenses.

Fill vector entries from values in a single *Field*. A range of component indices from that field may be read into the vector.

fieldVectorWriter Parameters

minDim (*integer*, *required*)
If NDIM < minDim, the writer will not be used to fill the vector.

component (*integer*, *required if componentBounds not specified*)
If a single-component fill, the component over which this writer operates. The number of components is set by the updater, and valid component indices are in the range [0, numberUpdaterComponents).

componentBounds (*integer vector*, *required if component not specified*)
The component range over which this writer operates. The number of components is set by the updater, and valid component indices are in the range [0, numberUpdaterComponents).

readFields (*string*, *required*)
The names of the fields whose values will be written to the vector. If multiple fields are given, only the first is used.

readComponent (*integer*, *required if readComponentBounds not specified*)
The component index of the field being read.

readComponentBounds (*integer vector*, *required if readComponent not specified*)
The range of component indices of the field being read.

scaling (*real*, *default = 1.0*)
A factor for scaling all field values that are written to the vector.

nodeFieldVectorWriter

Works with VSimBase, VSimEM, VSimPD, VSimPA, and VSimMD licenses.

Fill vector entries from values in a single *Field*, taking into account the geometry specified by a *GridBoundary*.

nodeFieldVectorWriter Parameters

minDim (*integer*, *required*)
If NDIM < minDim, the writer will not be used to fill the vector.

component (*integer*, *required if componentBounds not specified*)
If a single-component fill, the component over which this writer operates. The number of components is set by the updater, and valid component indices are in the range [0, numberUpdaterComponents).

componentBounds (*integer vector*, *required if component not specified*)
The component range over which this writer operates. The number of components is set by the updater, and valid component indices are in the range [0, numberUpdaterComponents).

readFields (*string*, *required*)
The names of the fields whose values will be written to the vector. If multiple fields are given, only the first is used.

readComponent (*integer*, *required if readComponentBounds not specified*)
The component index of the field being read.

readComponentBounds (*integer vector*, *required if readComponent not specified*)
The range of component indices of the field being read.

gridBoundary (*string*, *required*)
Name of the *GridBoundary* specifying the geometry.

interiorosity (*string vector*, *required*)
Which node locations are valid for writing vector entries. Options are any combination of `insideBoundary`, `cutByBoundary`, and `outsideBoundary`.

scaling (*real*, *default = 1.0*)
A factor for scaling all field values that are written to the vector.

stFuncNodeVectorWriter

Works with VSimBase, VSimEM, VSimPD, VSimPA, and VSimMD licenses.

Fill vector entries via evaluation of a *STFunc Block*, taking into account the geometry specified by a *GridBoundary*.

stFuncNodeVectorWriter Parameters

minDim (*integer*, *required*)
: If NDIM < minDim, the writer will not be used to fill the vector.

component (*integer*, *required if componentBounds not specified*)
: If a single-component fill, the component over which this writer operates. The number of components is set by the updater, and valid component indices are in the range [0, numberUpdaterComponents).

componentBounds (*integer vector*, *required if component not specified*)
: The component range over which this writer operates. The number of components is set by the updater, and valid component indices are in the range [0, numberUpdaterComponents).

STFunc (*code block*, *required*)
: The *STFunc Block* whose value at each point in space will be entered into the vector.

gridBoundary (*string*, *required*)
: Name of the *GridBoundary* specifying the geometry.

interiorosity (*string vector*, *required*)
: Which node locations are valid for writing vector entries. Options are any combination of `insideBoundary`, `cutByBoundary`, and `outsideBoundary`.

scaling (*real*, *default = 1.0*)
: A factor for scaling all function values that are written to the vector.

stFuncVectorWriter

Works with VSimBase, VSimEM, VSimPD, VSimPA, and VSimMD licenses.

Fill vector entries via evaluation of a *STFunc Block*.

stFuncVectorWriter Parameters

minDim (*integer*, *required*)
: If NDIM < minDim, the writer will not be used to fill the vector.

component (*integer*, *required if componentBounds not specified*)
: If a single-component fill, the component over which this writer operates. The number of components is set by the updater, and valid component indices are in the range [0, numberUpdaterComponents).

componentBounds (*integer vector*, *required if component not specified*)
: The component range over which this writer operates. The number of components is set by the updater, and valid component indices are in the range [0, numberUpdaterComponents).

STFunc (*code block*, *required*)
: The *STFunc Block* whose value at each point in space will be entered into the vector.

scaling (*real*, *default = 1.0*)
: A factor for scaling all function values that are written to the vector.

3.7.9 LinearSolver Block

LinearSolver

A block used to solve a linear equation.

Note: When running in parallel we suggest avoiding the direct solver.

LinearSolver Kinds

`kind` (*required string*)
One of:

- *iterativeSolver*
- *directSolver*

```
<LinearSolver mySolver>
  kind = iterativeSolver
  <BaseSolver myBS>
    kind = gmres
  </BaseSolver>
  <Preconditioner precond>
    kind = multigrid
    mgDefaults = SA
  </Preconditioner>
  tolerance = 1.e-8
  maxIterations = 1000
</LinearSolver>
```

LinearSolver Kinds

iterativeSolver

Works with VSimBase, VSimEM, VSimPD, VSimPA, and VSimMD licenses.

Solve a linear equation using iterative methods.

iterativeSolver Parameters

`BaseSolver` (*code block*, *required*)
BaseSolver parameters include:

`kind` (*required*)

One of the following options: Keep in mind that every problem will respond differently to these solvers and preconditioner options, so you may want to try a few types to find your best fit.

- gmres

  Generalized minimal residual (GRMRES) method. Nice for when bad convergence is an issue. Opposite of BiCGSTAB in that it's good for fewer iterations and small vector update time.

  – **`kspace` (integer, default = 30):** The size of the Krylov vector space used.

  – **`orthog` (string, one of `classic` or `modified`):** The type of orthogonalization used.

- cg

Conjugate gradient (CG) method. Good choice for when matrix is symmetric and positive definite.

- cgs

  Conjugate gradient squared (CGS) method.

- tfqmr

  Transpose free quasi-minimal residual (TFQMR) method.

- bicgstab

  Biconjugate gradient stabilized (BiCGSTAB) method. Can be applied to large number of problems, good for when total number of iterations is large. (If you are experiencing a break-down with CGS, switching to BiCGSTAB method may be advisable.)

`Preconditioner` (*code block*, *required*)

Preconditioner parameters include:

`kind` (required) - one of

- **none** No preconditioner.
- **jacobi** Jacobi (or diagonal) preconditioner. Basic iterative method, good for mostly diagonal matrices.
- neumann
- **leastSquares** Least-squares preconditioner.
- **symmetricGaussSeidel** Gauss-Seidel preconditioner for symmetric matrices. Similar to Jacobi, but only requires one storage vector (advantageous for very large problems). Elements cannot be computed in parallel.
- **domainDecomposition** Multigrid preconditioner. Most powerful pre-conditioner method; good for large problems.
  - `mgDefaults`
  - `maxLevels`
  - `smootherType`
  - `smootherSweeps`
  - `smootherPrePost`
  - `coarseType`
  - `dampingFactor`
  - `threshold`
  - `increaseDecrease`

`tolerance` (*real*, *default = 10e-06*)

Tolerance to be used in determining convergence.

`maxIterations` (*integer*, *default = 1000*)

Maximum number of iterations to reach convergence.

`convergenceMetric` (*string*, *default = r0*)

The residual measure used for determining convergence. One of

- r0
- rhs

- Anorm
- noscaled
- sol

Example iterativeSolver Block

```
<LinearSolver mySolver>
  kind = iterativeSolver
  <BaseSolver>
    kind = gmres
  </BaseSolver>
  <Preconditioner>
    kind = multigrid
    mgDefaults = SA
  </Preconditioner>
  tolerance = 1.e-10
  maxIterations = 1000
</LinearSolver>
```

directSolver

directSolver: Works with VSimBase, VSimEM, VSimPD, VSimPA, and VSimMD licenses.

Solve a linear equation using direct methods.

directSolver Parameters

solverType (*string*)
Solver type to use. Currently, the only option is *superLU*.

Example directSolver Block

```
<LinearSolver mySolver>
  kind = directSolver
  solverType = superLU
</LinearSolver>
```

3.7.10 Scalar Block

Scalar

A scalar is a grid independent quantity that can be updated in each time step. One scalar carries one value that is advanced in time. Binary operations between scalars, scalar and field, as well as unary operations, are defined via different updaters. A scalar is initialized to 0 unless an STFunc is used to set its initial value. Scalars can be recorded via History, or dumped into a MultiField file. External circuit models and ODE solvers can be implemented with scalars in MultiFields.

The synchronization of scalars across ranks is different from fields. For a scalar, its value on the head node is broadcast to all other ranks by setting 'messageScalars' in <UpdateStep>. When recorded by History, it is also the head node value that is saved into history file.

Scalar object code block contained between the tags:

```
<Scalar *nameOfThisScalar*>

</Scalar>
```

Scalar Parameters

kind (*string*, *default = default*)
: Type of field, one of: `regular`. This is the only available option for scalar kind. It defines a scalar with a single value.

Scalar Blocks

initialCondition (*optional*, *block*)
: An STFunc block that is used as an initial condition for the scalar. This STFunc must be named as 'initialCondition'. This STFunc is evaluated at spatial origin (0,0,0) and time t=0. The result is used as the inital value of the scalar. If this block is not provided, the scalar is initialized to 0.

Example scalar Block

```
<Scalar A1>
  <STFunc initialCondition>
    kind = expression
    expression = 5.0
  </STFunc>
</Scalar>

<Scalar A2>
  kind = regular
</Scalar>
```

3.7.11 Updater Blocks for Scalars

Updater Kinds for Scalars

field2ScalarUpdater

Works with VSimBase, VSimEM, VSimPD, VSimPA, and VSimMD licenses.

Scalar updater that applies a contraction operation on one field, and writes the result to scalars.

field2ScalarUpdater Parameters

The **field2ScalarUpdater** takes the `lowerBounds` and `upperBounds` parameters of *FieldUpdater*, as well as the following parameters:

readField (*required string*)
: The name of the single field on which to operate.

reduceMethod (*required string*)
: Contraction operation to apply to the field. `sum` is the only one option available now. It sums over a given components of a field and write the total sum result to a scalar.

readComponents (*required integer vector*)
: The components to use in the operand fields.

writeScalars (*required string vector*)
: The scalars where results are written into. There shoul be same number of scalars as number of components in *readComponents*. As each scalar corresponds to operation on one component of a field.

Example field2ScalarUpdater Block

```
# A7 = sum(F1_x), A8 = sum(F1_z)
<Updater fieldSum>
  kind = field2ScalarUpdater
  reduceMethod = sum
  readField = F1
  lowerBounds = [0 0 0]
  upperBounds = [NX1 NY1 NZ1]
  readComponents = [0 2]
  writeScalars = [A7 A8]
</Updater>
```

scalarBinOpUpdater

Works with VSimBase, VSimEM, VSimPD, VSimPA, and VSimMD licenses.

MultiField updater that applies a mathematical operation on two scalars (A1 and A2) specified through the *readScalars* parameter, and writes the result to the *writeScalars* parameter.

scalarBinOpUpdater Parameters

The **scalarBinOpUpdater** takes the following parameters:

readScalars (*required string vector*)
: A vector of the names of the two scalars on which to operate.

writeScalars (*required string vector*)
: A vector containing a single element, the name of scalar to update.

binOp (*required string*)
: Operation to apply to the scalars; one of `add`, `subtract`, `multiply` or `divide`. Operations are:

```
add:       (aCoeff*A1)+(bCoeff*A2)
subtract:  (aCoeff*A1)-(bCoeff*A2)
multiply:  (aCoeff+A1)*(bCoeff+A2)
divide:    (aCoeff+A1)/(bCoeff+A2)
```

aCoeff (*optional float*)
: Coefficient for first scalar (A1, as described above). Default values:

```
add:      1.0
subtract: 1.0
multiply: 0.0
divide:   0.0
```

bCoeff (*optional float*)
: Coefficient for second scalar (A2, as described above). Default values:

```
add:      1.0
subtract: 1.0
multiply: 0.0
divide:   0.0
```

Example scalarBinOpUpdater Block

```
<Updater scalarAdd>
  kind = scalarBinOpUpdater
  binOp = add
  readScalars = [A1 A2]
  writeScalars = [A2]
  aCoeff = 1.5
  bCoeff = 0.5
</Updater>
```

scalarFieldBinOpUpdater

Works with VSimBase, VSimEM, VSimPD, VSimPA, and VSimMD licenses.

MultiField updater that applies a mathematical operation on scalars and one field, and writes the result to a field.

scalarFieldBinOpUpdater Parameters

The **scalarFieldBinOpUpdater** takes the `lowerBounds` and `upperBounds` parameters of *FieldUpdater*, as well as the following parameters:

readScalars (*required string vector*)
: A vector of the names of the scalars on which to operate. There should be at least one scalar. The number of components to updated (NUM) depends on the setting of *`readComponents`* and *`writeComponent`*. If the number of readScalars is less than NUM, the last scalar is paded to NUM. If the number of readScalars is more than NUM, any scalars above NUM are disregarded. The binary operation happens between one scalar and one component of the field

readField (*required string*)
: A vector of the name of the single field on which to operate.

writeField (*required string*)
: A vector containing a single element, the name of field to update.

binOp (*required string*)
: Operation to apply to the field and scalars; one of `add`, `subtract`, `multiply` or `divide`. Those operations take place between one scalar and one component of the readField. Operations are:

```
add:        (aCoeff*A)+(bCoeff*Fj)
subtract:   (aCoeff*A)-(bCoeff*Fj)
multiply:   (aCoeff+A)*(bCoeff+Fj)
divide:     (aCoeff+A)/(bCoeff+Fj)
```

readComponents (*optional integer vector*)
The components to use in the operand fields.

writeComponent (*optional integer*)
The component to update.

Note: The *readComponents* and *writeComponent* parameters work together, and if one is specified, the other must be as well. If neither are specified, then all the components are updated. In that case, both of the *readField* and the *writeField* must all have the same number of components.

aCoeff (*optional float*)
Coefficient for the scalar (A, as described above). Default values:

```
add:      1.0
subtract: 1.0
multiply: 0.0
divide:   0.0
```

bCoeff (*optional float*)
Coefficient for the field (F, as described above). Default values:

```
add:      1.0
subtract: 1.0
multiply: 0.0
divide:   0.0
```

Example scalarFieldBinOpUpdater Block

```
<FieldUpdater scalarFieldAdd>
  kind = scalarFieldBinOpUpdater
  lowerBounds = [0 0 0]
  upperBounds = [NX1 NY1 NZ1]
  binOp = add
  readScalars = [A1 A2]
  readField = F1
  writeField = F2
  readComponents = [0 1 2]
  writeComponents = [0 1 2]
  aCoeff = 2.0
  bCoeff = 3.0
</FieldUpdater>
```

unaryScalarOpUpdater

Works with VSimBase, VSimEM, VSimPD, VSimPA, and VSimMD licenses.

Scalar updater that performs one of the following operations on a scalar $\mathbf{A}$, another scalar $\mathbf{B}$, and an **STFunc** f:

$$B = f(0, 0, 0, A) \tag{3.15}$$

$$B = f(0, 0, 0, t) * A \tag{3.16}$$

$$B = B + f(0, 0, 0, t) * A \tag{3.17}$$

$$B = B * f(0, 0, 0, t) * A \tag{3.18}$$

$$B = \frac{B}{f(0, 0, 0, t) * A} \tag{3.19}$$

The above scalar A is the *readScalars*, and scalar B is the *writeScalars*.

unaryScalarOpUpdater Parameters

The **unaryScalarOpUpdater** takes the following parameters:

readScalars (*optional string vector*)
A single element, the name of the scalar to use in the operation ($\mathbf{A}$, above). There can be either zero or one read scalar. If there is no read scalar specified, a scalar with default value 1 will be used.

writeScalars (*required string vector*)
A single element, the name of the scalar to update ($\mathbf{B}$, above). There must be exactly one write scalar.

STFunc (*required parameter block*)
A parameter block of type **STFunc** (with any name) must be specified. This describes f. This function is always evaluated at spatial origin (0,0,0) for this updater. So it is perfered a function that only depends on time t.

operation (*required string*)
One of:

- apply: See (3.15).
- set: See (3.16).
- add: See (3.17).
- multiply: See (3.18).
- divide: See (3.19).

dtCoefficients (*optional float vector, default = [1.0 0.0]*)
Two components [c_0 c_1]: the function f will be multiplied by $(c_0 + c_1\Delta t)$, where Δt is the current time step. If c_1 is not specified it is assumed to be zero.

Example unaryScalarOpUpdater Block

```
<Updater scalarSet>
  kind = unaryScalarOpUpdater
  operation = set
  writeScalars = [A3]
  dtCoefficients = [1.0 0.0]
  <STFunc phiFunc>
     kind = historySTFunc
     feedback = phiHist
```

(continues on next page)

(continued from previous page)

```
      expression = 1.0
   </STFunc>
</Updater>

<Updater scalarApply>
   kind = unaryScalarOpUpdater
   operation = apply
   readScalars = [A5]
   writeScalars = [A5]
   dtCoefficients = [1.0 0.0]
   <STFunc phiFunc>
      kind = expression
      expression = sin(TWOPI*t/30)
   </STFunc>
 </Updater>
```

userFuncScalarUpdater

Works with VSimBase, VSimEM, VSimPD, VSimPA, and VSimMD licenses.

Sets scalar values to the result of a user-given function.

The Updater of `kind = userFuncScalarUpdater` is a very flexible updater on scalars based on the result of a user-given <Expression> (see *expression*). The expression takes as arguments scalars values, as well as time and time step. This updater is basically the scalar version of the field updater **`userFuncUpdater`**. See it for more details about possible applications.

userFuncScalarUpdater Parameters

The **`userFuncScalarUpdater`** takes the following parameters:

`readScalars` (*optional string vector*, *default = []*)
: The names of the scalars to read. A read scalar name can be used as a variable in the definition of *`updateFunction`*.

`writeScalars` (*required string vector*)
: The names of the scalars to write.

`updateFunction` (*required parameter block*)
: A parameter block of type **`Expression`** and name *`updateFunction`* is required to define the function used to update the scalars. This block takes the same parameters as a *expression UserFunc*, except for `input` blocks and the `inputOrder` parameter. The input variables for the `expression` are defined by the above parameters, so they are not to be specified in the *`updateFunction`* block. In addition, the expression can take the following input valiables:

  `t`: The time to which the updater been told to update its writeFields; c.f. toDtFrac in a MultiField UpdateStep).

  `dt`: The most recent time step (Δt).

  `n`: The current simulation step (an integer).

  (These names are reserved—they cannot be used as scalar names.)

  The Expression must return a vector with the same length as the number of writeScalars—each write scalar value will be set to the corresponding component of the return vector.

userFuncScalarUpdater Example

Here is a userFuncUpdater to set two scalars A4 and A5

```
<Updater userfuncUp>
  kind = userFuncScalarUpdater
  readScalars = [A1 A2 A3]
  writeScalars = [A4 A5]
  <Expression updateFunction>
    expression = vector(A1 + 3.0*A2 + A3/5.0 + t + n/10 + 2.0*dt, n)
  </Expression>
</Updater>
```

3.7.12 UpdateStep and InitialUpdateStep Block

UpdateStep and InitialUpdateStep

`UpdateStep` and `InitialUpdateStep`: Blocks to define how fields are updated in a **`MultiField`** block.

`InitialUpdateStep`: Used to define an update step to be performed at `time = 0`; Vorpal will also run InitialUpdateStep after a restore if the *`alsoAfterRestore`* flag is set to `true`.

`UpdateStep`: For every subsequent update after InitialUpdateStep, use an UpdateStep block.

Steps to be performed at every time step are indicated between the tags:

```
<UpdateStep *nameOfThisStep*>

</UpdateStep>
```

Note: By default, steps are ordered as they appear in the input file. However, you may overrule the order by setting the parameter updateStepOrder = [*step1 step2* . . .].

InitalUpdateSteps does not use *`toDtFrac`*; these are updated with time 0 at initialization; and after restore (if `alsoAfterRestore = true`), they are updated with the current MultiField time.

Regarding toDtFrac, when MultiField is at t_{n-1} and is told to update itself to time t_n by calling update (t_n), MultiField will call update (t') for all updaters in the update step, where

$$t' = t_{n\check{}1}(1\check{}\text{toDtFrac}) + t_n \cdot \text{toDtFrac}.$$

In other words, `toDtFrac = 1` tells MultiField to update the updaters (in the update step) to its same time; whereas, `toDtFrac = 0.5` updates the updaters to the half-way time.

Note: The parameter toDtFrac does not refer to a time step, but to an absolute time.

For example, suppose that an updater is in only one update step, with `toDtFrac = 0.5`. During the first update, that updater will be updated by only a half-time step; however, all subsequent updates will be updated by a whole time step.

Note: Field updaters may appear in more than one update step. It is common in EM simulations with particles to perform the B-update with `toDtFrac = 0.5` then the E-update with `toDtFrac = 1`, and then the B-update with `toDtFrac = 1`.

Field-particle-overlap

Update steps are divided into two groups:

First group: Contains all the update steps before the first update step with an updater that requires the field rhoJ (here the name must be rhoJ exactly).

Second group: Remaining updaters after the first update step.

Note: The update order for updaters within the same update step is sometimes difficult to predict, because of skin/body separation. The skin updates will all be done in the order in which the updaters are listed, and the same is true for the body updates. However, some of the regions covered by the skin and body updates may differ for different updaters. For example, some updaters cannot be separated: consider a skinnable updater u1 and a non-skinnable updater u2. If updaters = [*u1 u2*] then *u1* will be done before *u2* on the skin cells of *u1*, while *u2* will be done before *u1* on the body cells of *u1* (because the entire *u2* update is treated as a skin update). Two updaters can both be skinnable but have different skin and body regions.

When two updaters are of the same kind, and read and write the same fields then updates should be performed in the listed order.

UpdateStep and InitialUpdateStep Parameters

`updaters` (*string vector*, *required*)
Names of updaters to be executed during this step.

`messageFields` (*string vector*)
Name of the field to be messaged. Parallel processing often requires that domain boundary fields be messaged between processors. Only one field can be specified. Typically it is one of the updater's `writeFields`. The `dummyUpdater` can be used if more than one `writeField` should be messaged. Periodic boundary conditions will also be applied to only this field.

`toDtFrac` (*float*, *required*)
Updates the updaters to either the same time, or to the half-way time.

`alsoAfterRestore` (*optional*, *default = false*)
Specifies whether to perform this update step when restoring from a dump. This is handy, for example, for nodal fields, which can be easily computed from the Yee fields. One of:

- `true`
- `false`

Example Update Step Block Using the nodalE field in the MessageFields Parameter

```
<UpdateStep  step08>
  toDtFrac = 1.0
  messageFields = [nodalE]
```

(continues on next page)

(continued from previous page)

```
  updaters = [plasmaDielectric]
</UpdateStep>
```

3.7.13 PmlRegion Block

PmlRegion

Used to describe where (if any) PMLs exist in the simulation. For every `PmlRegion`, the region will be divided into slabs; for each slab two `FieldMultiUpdaters` will be created, a Faraday PML updater, and an Ampere PML updater (each with three components). The `lowerBounds` and `upperBounds` will be given to the occurrences of *`FieldMultiUpdater`*, as well as the PML sigma functions.

Vorpal first looks for an *`STFunc`* sigmaX (this is what the PML updater wants in the end). If there isn't one in the `PmlRegion` input block, Vorpal looks for an *`STFunc`* sigmaX1. If Vorpal doesn't find an *`STFunc`* sigmaX1, it looks for a sigmaFormX, from which it can create sigmaX1, using `sigmaFactorX` if present. Otherwise, Vorpal looks for a *`sigmaForm`*, from which it can create sigmaX1, using `sigmaFactorX` at present. You can mix all three methods. The resulting sigmaX1 is renamed sigmaX and put in the resulting PML `MultiUpdater`.

As a rule of thumb, you should make the outer region the outer boundary of the PMLs, and the inner region the normal Maxwell update area; generally this will give the correct result. In typical simulations, the inner region may be specified to be the normal Yee update region, and the outer region to be the outer boundary of the PML. This rule applies also when the PML is only a single slab, or is only slabs on the right and left. Before setting itself up, the PML considers the inner region to be the intersection of the outer region with the input-file-specified inner region. Specifying the inner slab might seem a bit tricky in some cases. In Vorpal a PML slab can have a direction; and the simulation gets this direction from the relation between the inner and outer slabs. In the above, for instance, the slab from [105 10] to [110 210], on the right side of the simulation, attenuates waves only in the x direction; the PML algorithm can be computed more efficiently for it than for the upper-right corner [105 0] to [110 10]. Because of this, you must be careful while specifying the inner region when a PML does not surround it completely.

In the end, Vorpal will give an error if a PML updater does not have the sigmaWn STFuncs that it needs.

Note: instantiations of `UniPmlUpdater` are created for the side slabs, and instantiations of `PMLUpdater` are created for the corner regions.

The slabs specified in PmlRegion will be expanded and contracted to and from the simulation boundary, depending on the component and `emType` (Faraday or Ampere).

Note: PMLs fail to be reflectionless for some materials, including photonic crystals. It is recommended to use the Matched Absorbing Layer (MAL) instead. PMLs may also fail when combined with other active boundary conditions, including but not limited to; ports, particles at the boundary, or structures that are not normal to the boundary. See *Perfectly Matched Layer* in VsimReference.

PmlRegion Parameters

`faradayUpdaterName` (*string*, *required*)
: Name of the Yee Faraday updater. (The PML Faraday updater will be put in the same update step.)

`ampereUpdaterName` (*string*, *required*)
: Name of the Yee Ampere updater.

eFieldName (*string*, *required*)
Name of the Yee E field.

bFieldName (*string*, *required*)
Name of the Yee B field.

eAuxFieldName (*string*, *default = pmlEAuxField*)
Name of the PML auxiliary E field. The Field attribute will be created automatically if it needed.

bAuxFieldName (*string*, *default = pmlBAuxField*)
Name of the PML auxiliary B field. The Field attribute will be created automatically if it needed.

rhoJFieldName (*string*)
Name of the density-current field (current at components 1, 2, 3, density at 0).

rhoJmagFieldName (*string*)
Name of the density-current field associated with "magnetic" current (untested).

useUniPml (*optional*, *default = true*)
Indicates whether to use the more efficient uniaxial PML algorithm, for which only one sigmaW is non-zero, when possible.

sigmaForm (*string*)
Function of w, meant to be a form for producing sigmaWn, the conductivity profile. w is replaced by x, y, or z, depending on W given by *sigmaFactorW*. It is multiplied by *sigmaFactorW*.

sigmaFormW (*string*)
Function of w, meant to be a form for producing sigmaWn, the conductivity profile. w is replaced by x, y, or z, depending on W given by sigmaFactorW. A string that is multiplied by sigmaFactorW. (W is one of {X,Y,Z}).

sigmaFactorW (*float vector, default = [1.0 1.0]*)
(W is one of {X,Y,Z}). – components of sigmaFactor [sigmaFactorX1 sigmaFactorX2] are used to multiply the expressions of sigmaFormX or sigmaForm for the X1 and X2 slabs.

STFunc

Block can be used to specify the sigma function directly. The name of the block must correspond to sigmaW or sigmaWn where W is X,Y,Z and n is 0 or 1 (lower or upper). See below for examples.

Sigma may be specified through a combination of *sigmaForm*, *sigmaFormW* and *STFunc* blocks.

Region (*code blocks*)
PmlRegion requires the use of two Region blocks to give the inner and out bounds of the PML region. The inner region describes the region inside PML (usually the region describing the normal Maxwell update; can/may extend beyond the outer region). The outer region describes the region bounding PML (which will be slightly altered for each component).

Region block parameters include:

`lowerBounds` (integer vector, required)

Lower bounds of the region.

`upperBounds` (integer vector, required)

Upper bounds of the region.

energyWritePeriod (*integer*, *default = 0*)
If non-zero, the energy absorbed by the PML every energyWritePeriod steps to the comms text file. If zero, the PML region does not keep track of absorbed energy.

MPML

The basic PML has an increasing conductivity, or imaginary part of the dielectric constant (uniaxial PML, see *[Ged96]*). While the PML is good at absorbing propagating waves, it is not very good at absorbing evanescent waves. The MPML (Modified PML, see *[RGH02]*) increases the real part of the dielectric constant along with the conductivity (imaginary part). This yields better absorption of evanescent waves; for example, it is a better absorber at the end of a waveguide.

For an MPML, choose `kind=mpml` in the <PmlRegion>. Then there is an additional option to set:

`relPermForm` (*string*, *required*)
Like `relSigmaForm`, but specifies the relative permeability (real part of the dielectric constant), which typically should increase from 1 at the MPML interface to some maximum (perhaps 2 to 10) as a power law.

`useSmallSigmaApproximation` (*boolean*, *default = false*)
Whether to use a time-advance algorithm that assumes $\sigma\Delta t \ll 1$. An option for experts only.

`addSigmaForm` (*string*)
Should be set to 0 (this is an experimental option).

Example PMLslab Block

This example demonstrates a two-dimensional block specifying a PML 5 cells thick on the left and right sides of the inner region, and 10 cells thick above and below the inner region. (The region will be divided into several slabs (rectangular regions) and updaters will be created for each region.)

```
<Region  outer>
  lowerBounds = [0 0]
  upperbounds = [110 220]
</Region>
<Region  inner>
  lowerBounds  = [5 10]
  upperbounds = [105 210]
</Region>
```

Example PMLslab Block

This example demonstrates describing a slab on only the right hand side.

```
<Region outer>
  lowerBounds = [105 0]
  upperbounds = [110 220]
</Region>
<Region inner>
  lowerBounds = [105 0]
  upperbounds = [105 220]
</Region>
# inner slab is degenerate in the x direction,
# and on the left side of the outer slab,
# indicating that the wave will enter the PML from the left side
```

Example PML slab Block

This is another example that demonstrates describing a slab on only the right hand side.

```
<Region outer>
  lowerBounds = [105 0]
  upperbounds = [110 220]
</Region>
<Region inner>
  lowerBounds = [0 0]
  upperbounds = [105 220]
</Region>
```

Example Block Describing a Conductivity Function for Each Direction

```
# Function  is 0 zero at x-grid number 105,
# and  increases as x increases.
# c is the  speed of light and DX the cell-length
# in the x  # direction.
<STFunc  sigmaX2>
  kind = expression
  expression = 1.0 * c / DX * ( (x -  105*DX)/(5*DX) )\^{ }2
</STFunc>
# Conductivity for each slab is specified by
# an STFunc  block
# Conductivity for each slab is specified by
# an STFunc  block:
# sigmaX1  -x (left slab)
# sigmaX2  +x (the right slab)
# sigmaY1  -y (bottom slab)
# sigmaY2  +y (top slab)
# sigmaZ1  -z (back slab)
# sigmaZ2  +z (top slab)
```

Example Block for Specifying When There is a Bottom Slab

```
<STFunc  sigmaY1>
  kind = expression
  expression = 1.5 * c / DY * ( (210*DX -  y)/(10*DX) )\^{ }2
</STFunc>
# You only  need the above when there is a bottom slab.
# sigmaXn  should depend only on x, and likewise for other
# directions.
# Not  recommended - sigmaXn may be allowed to depend
# on x, y,  z, and t
```

Example sigmaForm Block Reuse Conductivity Profile for Different Directions

```
#  specifying sigmaForm
sigmaForm = 1.5 * c/DX * w\^{ }2
sigmaFactorX = [ 2.0 0.5 ]
# from  which Vorpal can make a sigmaWn by using:
<STFunc  sigmaX1>
    kind = expression
    expression = 2.0 * (1.5 * c/ DX * ( (5*DX -  x)/(5*DX) )\^{ }2 )
```

(continues on next page)

(continued from previous page)

```
</STFunc>
<STFunc  sigmaX2>
    kind =  expression
    expression  = 0.5 * (1.5 * c/ DX * ( (x - 105*DX)/(5*DX) )\^{ }2 )
</STFunc>
# Vorpal  calculates the real coordinates of PML edge and PML
# thickness  and replaces w with a linear function of x (or normal
# direction)that  goes from 0 at the edge to 1 at the end
# then  multiplies replacement value by factor paramVec
# sigmaFactorX.
#
#
# To have  different functional forms in the X, Y, and Z
# directions  specify a sigmaFormW where W is X, Y, or Z.
# Again,  make sigmaFormX (e.g.)  a function of the dummy
# variable w (w is  recognized as the dummy variable if it is
# neither  preceded followed by another letter, number, or
# underscore).
# Again, sigmaFactorX is used to allow different factors  for
# sigmaX1 and sigmaX2.
```

3.7.14 Region Indices in Multifield Updaters

Understanding Region Indices Used With the MultiField Updater

The region alteration flags facilitate description of the possibly different regions of the three components of an updater (such as a Yee Faraday updater). So far, Vorpal protocol has been to compute the magnetic field on simulation boundary faces, but not to compute the electric field on those boundaries (instead setting the electric field by a boundary condition). If the simulation has 10 x 10 x 10 cells, with lower and upper bounds [0 0 0] and [10 10 10] then typically the bounds of updaters for the various field components (in the Yee scheme) are:

- E_x: [0 1 1] and [10 10 10]
- E_y: [1 0 1] and [10 10 10]
- E_z: [1 1 0] and [10 10 10]
- B_x: [0 0 0] and [11 10 10]
- B_y: [0 0 0] and [10 11 10]
- B_z: [0 0 0] and [10 10 11]

The actual field components that are in the simulation volume (including the boundary) are:

- E_x: [0 0 0] and [10 11 11]
- E_y: [0 0 0] and [11 10 11]
- E_z: [0 0 0] and [11 11 10]
- B_x: [0 0 0] and [11 10 10]
- B_y: [0 0 0] and [10 11 10]
- B_z: [0 0 0] and [10 10 11]

To specify the update regions above, you can specify the bounds [0 0 0] and [10 10 10] for all updaters, and add the flag expandToTopInComponentDir = true for the magnetic field updaters, and contractFromBottomInNonComponentDir for the electric field updaters. If the region does not touch the global boundary of the simulation, no changes are made.

Note: Regarding updating beyond the domain, the options cellsToUpdateBelow/AboveDomain allow the updater to update beyond the domainRgn. The domainRgn is the region for which a particular processor is responsible (the main simulation region if running serial on one processor). However, in many cases a processor stores guard cells for a field – cells that technically belong to another processor; the main simulation region usually also has guard cells. Usually the guard cells are updated by that other processor, and the values sent (messaged) to other processors that need those values. Sometimes, when possible, it's more efficient to calculate guard cell quantities than to receive the values from another processor. This option facilitates such transactions. To understand how to use the cellsToUpdate-Below/AboveDomain options, it helps to understand how an updater figures out which cells to update.

- The updater finds the largest region which it is allowed to update; usually this is the local DomainRgn (for example, lowerBounds = [4 6 7], upperBounds = [10 12 14]), but it will be expanded according to the following rules:
  - If the DomainRgn is at the bottom of the simulation in direction d (and d is not a periodic direction), the updater will be allowed to write as far below the DomainRgn in direction d as it can without going beyond field data (Vorpal may incorrectly estimate how far this is). The top of the simulation is treated similarly. .. note:: Boundary conditions often update field values beyond the global DomainRgn; this feature allows that.
  - If the DomainRgn is not at the bottom of the simulation in direction d (or d is periodic), then the updater will be allowed to write below the DomainRgn by the number of cells given by (the dth element of) cellsToUpdateBelowDomain. For example, if cellsToUpdateBelowDomain = [1 2 2], the lowerBounds will be adjusted to [3 4 5] (that is, the largest allowed region will have lowerBounds = [3 4 5]).
- The updater takes the intersection of the region specified in the input file with the largest allowed region for the particular processor, found in the previous step; this is the updater's update region.
  - **Note:** In the case that direction d is periodic, the region specified in the input file is considered to include translations of itself in direction d. For example, consider a 1D simulation with N cells. If the updater bounds are specified in the input file as 0 to N, and cellsToUpdateBelowDomain = [1], then the allowed region will be -1 to N, and the updater will update cell -1 because cell -1 is equivalent to cell N-1, and cell N-1 is included in the updater bounds. In other words, the intersection of [-1,N] and [0,N] is [-1,]. However, if the update region had been [0,N-5], excluding cell N-1, then the updater would not update cell -1. Because this might be confounding, and Vorpal's logic may not be perfect, this notion is implemented when the updater region (specified in the input file) is altered, before the updater is created. Vorpal will try to extend the updater bounds in periodic directions as it sees fit. In other words, [0,N] would be extended to [-1,N], and this would be the region that appears in MultiField's attribute set, which it writes to the standard output, as well as to the per-rank streams. However, [0,N-5] would be unchanged. Then, a straightforward intersection is applied.

The updater then tries to make sure that the updater only needs field data that it can get (so that it doesn't try to go past the array, causing a segfault).

- The component/direction-specific options, such as cellsToExtendBelowDomainInComponentDir, alter the updater attribute set, before the updater is created, and add, say, cellsToExtendBelowDomain, as appropriate.

3.7.15 DielectricShape Block

DielectricShape Block

`DielectricShape`:

Works with VSimEM license.

This block is required when using a multiDielectricUpdater. The block sets the permittivity to a specified GridBoundary.

`DielectricShape` Parameters

`boundary` (*required string*)
: The name of the GridBoundary to which to set the permittivity to.

`permittivity` (*optional float*, *default = 1.0*)
: The value of the permittivity within the specified *`boundary`*.

`isFlipped` (*optional int*, *default = true*)
: Whether the boundary interior is flipped.

`DielectricShape` Example

```
<FieldUpdater setInvEps>
  kind = multiDielectricUpdater
  lowerBounds = [0  0  0]
  upperBounds = [46  51  51]
  permittivityField = invEps

  <DielectricShape cylinder0Unionsphere0Shape>
    boundary = cylinder0Unionsphere0
    permittivity = 9.9
  </DielectricShape>

  <DielectricShape cube0Minuscylinder00Shape>
    boundary = cube0Minuscylinder00
    permittivity = 9.0
  </DielectricShape>

  backgroundPermittivity = 1.0
</FieldUpdater>
```

3.8 ScalarDepositor and VectorDepositor

3.8.1 ScalarDepositor

More flexible way of depositing charge from charged particles in a simulation into `depFields` than the classical way of depositing charge into the zeroth component of a **`SumRhoJ`** field.

Note: The ScalarDepositor should not be used in a **`MultiField`** block of `kind = emMultiField`, or errors will occur in the simulation.

Note: Higher order particles are incompatible with the polar/cylindrical axis. If particles in the simulation pass near the polar/cylindrical coordinate axis at r = 0, only areaWeightingCP or esirk1stOrderCP can be used.

ScalarDepositor Parameters

`kind` (*string*, *default = areaWeighting*)
: Kind of deposition algorithm; choices are:

- **areaWeighting** Works with VSimBase, VSimEM, VSimPD, VSimPA, and VSimMD licenses.
- **esirk1stOrder** Works with VSimBase, VSimEM, VSimPD, VSimPA, and VSimMD licenses.
- **esirk2ndOrder, ..., esirk7thOrder** Works with VSimPD and VSimPA licenses.
- **areaWeightingCP (for coordProdGrid)** Works with VSimBase, VSimEM, VSimPD, VSimPA, and VSimMD licenses.
- **esirk1stOrderCP (for coordProdGrid)** Works with VSimBase, VSimEM, VSimPD, VSimPA, and VSimMD licenses.
- **esirk2ndOrderCP, ..., esirk7thOrderCP (for coordProdGrid)** Works with VSimPD and VSimPA licenses.

depField (*string*)
Scalar defined in the **MultiField** block that will contain the deposited charge.

Note: When using higher order deposition (esirk2ndOrder, ..., esirk7thOrder), maxIntDepHalfWidth must be set accordingly in the **Grid** block (see *Additional Attributes for Particle Simulations*).

Example ScalarDepositor Block

```
<ScalarDepositor chargeDep>
  kind = areaWeighting
  depField = myEmField.rho
</ScalarDepositor>
```

Example Reference to ScalarDepositor in a Species Block

```
<Species  electrons>
  kind = relBorisDF
  chargeDeps = [chargeDep]
  .
  .
  .
</Species>
```

3.8.2 VectorDepositor

Flexible way of depositing current from charged particles in a simulation into occurrences of *depField*, instead of the classical way of depositing current into the last three components of a **SumRhoJ** field. With a VectorDepositor you can simply employ a VectorDepositor and a three-vector J in an EM PIC simulation instead of using the four-vector SumRhoJ, as has been classically used, thus saving on memory.

Note: The VectorDepositor should not be used in **MultiField** block of kind = emMultiField, or errors will occur in the simulation.

Note: Higher order particles are incompatible with the polar/cylindrical axis. If particles in the simulation pass near the polar/cylindrical coordinate axis at r=0, only areaWeightingCP or esirk1stOrderCP can be used.

VectorDepositor Parameters

`kind` (*string*, *default = areaWeighting*)
Type of deposition algorithm; choices are:

- `areaWeighting`
- `esirk1stOrder`
- `esirk2ndOrder`
- `esirk7thOrder`
- `areaWeightingCP` (for **`coordProdGrid`**)
- `esirk1stOrderCP` (for **`coordProdGrid`**)
- `esirk2ndOrderCP` (for **`coordProdGrid`**)
- `esirk7thOrderCP` (for **`coordProdGrid`**)

`depField` (*string*)
Vector depfield defined in the MultiField block that will contain the deposited current.

Note: When using higher order deposition (`esirk2ndOrder`, ..., `esirk7thOrder`), `maxIntDepHalfWidth` must be set accordingly in the **`Grid`** block (see *Additional Attributes for Particle Simulations*).

Example VectorDepositor Block

```
<VectorDepositor~currDep>
    kind = areaWeighting
    depField = myEmField.J
</VectorDepositor>
```

Example Reference to `VectorDepositor` in a Species Block

```
<Species  electrons>
    kind = relBorisDF
    currDeps = [ currDep ]
    .
    .
    .
</Species>
```

3.9 SumRhoJ

3.9.1 SumRhoJ

SumRhoJ (singleton) Block that contains the charges and currents generated by the charged particles and fluids in the simulation. You can add to the charge and/or current density using an *Initial and Boundary Conditions* as previously described. Recall that a BoundaryCondition can be applied throughout a region, which can be useful for adding an external driving current to a simulation.

You do not need to use a **SumRhoJ** block unless you need to add a boundary condition or source.

SumRhoJ blocks include a Source block. In fact, a **SumRhoJ** block is most commonly used for adding a Source block whose purpose is to add a current source to a simulation.

Example SumRhoJ Block

```
#
#  Drive cavity with current source
#
<SumRhoJ  sumRhoJ>
    <Source currentSource>
    #  Apply everywhere
    lowerBounds = [0 0 0]
    upperBounds = [NX NY NZ]
    kind = varadd   # Value added to current field
    components = [3]  # J_z, rho is 0
        <STFunc component3>
            kind = expression
            expression =  AMP*sin(CAVOMEGA*t)*exp(-0.5*(t-T_0)^2/T_SIG^2)*H(STOPTIME-
↪t)*H(RADINIT^2 - x^2  - y^2)*J0(KAPPA*sqrt(x^2+y^2))
        </STFunc>
    </Source>
</SumRhoJ>
```

3.10 Fluid

3.10.1 Fluid Block

Block used to add charged and neutral fluids to a simulation. A fluid in Vorpal obeys the same initial and boundary conditions as a **Field**. As with a **Field**, initial and boundary condition blocks *Initial and Boundary Conditions* are nested within the Fluid blocks.

Fluid Kinds

- *coldRelFluid*
- *eulerFluid*
- *neutralGas*

3.10.2 coldRelFluid

Works with VSimPA license.

A charged, cold, relativistic fluid. The velocity is updated using an upwinding advection. The density is updated using a finite volume approach with the velocity found from upwinding. In both cases, alternating direction is used.

coldRelFluid Parameters

`charge` (*real*, *default -1.6e-19*)
Charge of charge carrier for the fluid.

`mass` (*real*, *default 9.1e-31*)
Mass of the charge carrier for the fluid.

`advectMomentum` (*integer*, *default = 1 (true)*)
Whether to advect the momentum.

`accelerate` (*integer*, *default = 1 (true)*)
Whether to accelerate momentum.

`convectDensity` (*integer*, *default = 1 (true)*)
Whether to convect density.

`weightCurrents` (*integer*, *default = 1 (true)*)
Whether to weight currents.

`correctDensity` (*integer*, *default = 0 (false)*)
Whether to apply second-order density corrections.

3.10.3 eulerFluid

`eulerFluid`

Works with VSimPD license:

A neutral, scalar-pressure fluid based on the algorithm described in pages 361 to 362 of Laney's Computational Gasdynamics *[Lan98]*.

eulerFluid Parameters

`ratioSH` (*real*)
Ratio of specific heats; affects the pressure calculation.

`epsilon` (*real*)
Value of epsilon for the fluid.

3.10.4 neutralGas

Works with VSimPA and VSimPD licenses.

(Non-dynamic) gas to be simulated.

neutralGas Parameters

`gasKind` (*string*, *no default value*)
Type of gas represented; used for interaction processes such as ionization, negative ion detachment, etc. The available values for `gasKind` will vary depending on the type of interaction. For more information on available gaskinds, please see *Working with neutralGas Fluids and the gasKind Parameter*.

`dumpTrack` (*string*, *default = false*)
Track the energy, momentum, and number of events for electron/gas interactions (including elastic, excitation, and ionization interactions). The results are written into separate h5 files. For example, when tracking an excitation interaction with dumpTrack, the additional output fields are *_ExcitationErgFld, *_ExcitationMomFld, and *_ExcitationEvtFld, where * is the name of the neutralGas block. The units of the energy field are eV, the units of the momentum field are MKS, and the units of the event field are the unitless number of events per cell. This option requires a corresponding Monte Carlo collision block to describe the various interactions.

`constantDensity` (*string*, *default = false*)
This flag will determine whether or not the fluid density is depleted or augmented via reaction processes.

`numMacroPPCRxns` (*integer*, *default = 1*)
The effective number of macro particles per cell used when performing collisions using the Reaction framework. This will only have an effect if kinetic species reacting with or produced by the reactions are variable weight. If particles are constant weight then the fluid's effective weight will be the same.

See Also

Initial and Boundary Conditions

3.11 Species

3.11.1 Species Blocks

Species

One of several different methods for modeling particles in Vorpal. Each Species block is defined by a kind parameter that corresponds to a particular Species algorithm, as well as parameters that describe characteristics of the Species block. Available kinds for Species blocks are discussed in documentation sections following the list of general Species Parameters. Vorpal represents macroparticles. See *Macroparticles* for more about macroparticles.

For some types of simulations, as discussed in the cellSpecies description, cellSpecies uses optimization algorithms that can result in performance surpassing that can be achieved using other species. Tech-X recommends using cellSpecies whenever your simulation data characterization can conform to functionality supported by cellSpecies.

Note: Vorpal considers the velocities of any non-relativistic particle kind to be velocity. All other particle kinds are γv.

Species Parameters

In addition to the name of the kind, which indicates the algorithm, Species parameters include:

kind (*string*)
Specifies the species algorithm to be used. See Species kinds in the sections following this list of general Species parameters.

charge (*double*)
Value describing the charge of a particle in the species.

mass (*double*)
Positive value describing the mass of a particle in the species.

emField (*string*)
User-defined combo EmField with which the particle species will interact.

bgEmField (*string*)
User-defined combo bgEmField used for particles of relBorisDF type.

fields (*string vector*)
User-defined Fields from a MultiField block with which particle Species will interact. The vector is of the form:

[Electric_Field Magnetic_Field],

where vector items require a space for separation. The order of these vector components is significant. This is used in a Multifields approach to define the electric and magnetic fields as an alternative to defining them through the combo emField.

bgElecField (*string*)
User-defined Field that explicitly specifies the background electric field via a Multifields Field. Used with particles of relBorisDF type and (in combination with bgMagField) as an alternative to bgEmField.

bgMagField (*string*)
User-defined Field that explicitly specifies the background magnetic field via Multifields Field. Used with particles of relBorisDF type and (in combination with bgElecfield) as an alternative to bgEmField.

chargeDeps (*string*)
User-defined ScalarDepositor that deposits charge into a MultiField's depField.

currDeps (*string vector*)
User-defined VectorDepositor that deposits current into a MultiField's depField. Where a complex electric and magnetic field has been used (see cmplxRelBorisDF) one would use currDeps to define both real and imaginary current depositors as follows:

[Real_Current_Depositor Imag_Current_Depositor]

nominalDensity (*double*)
Positive value suggesting the nominal density for the particles.

nomPtclsPerCell (*double*)
Positive value suggesting the nominal macro particles per cell in the simulation.

numPtclsInMacro (*double*)
Positive value setting the number of particles in a macro particle.

depositCurrentThoughFieldsNotUsed (*boolean*, *default = true*)
Whether a species should deposit current (to SumRhoJ) even if the species does not use the electric or magnetic fields. This option should be set to true with freeRel or freeRelVW species (which do not "see" EM fields) if the particles should deposit current (hence generate fields). If this option is not set to true, freeRel particles will have no effect on EM fields. Other (non-free) species will deposit current regardless of this option.

overwriteTag (*boolean*, *default = false*)
Whether to regenerate the tag when adding a particle. This option takes effect only with tagged species. When this option is on, the tag is generated inside the species, rather than with the velocityGenerator. This allows generation of a unique tag when several particle sources are present in the species or if particles of this species

can be generated by a MonteCarlo interaction. This option also guaranties that the tags are generated in a unique way after restore. If *overwriteTag* is not set or set to false, we cannot guarantee that Vorpal will generate unique tags after restore.

For example, the tag may be generated at the same time that a particle's velocity is generated, but with overwriteTag, that tag will be overwritten by the <Species>.

useSequentialTags (*boolean*, *default = false*)
Whether to generate sequential tags for particles (this option only applies if `overwriteTag = true`); e.g., if there are 10 tagged particles in the simulation, they will have tags 0 through 9. In parallel simulations, communication between processors is required (at every time step that particles might be loaded or emitted) to generate sequential tags; this may significantly slow a simulation.

If tagged particles are loaded or emitted at only a few time steps, it is recommended to use `applyTimes` (or similar) within the <ParticleSource>; that will prevent any communication except during the valid applyTimes.

consolidatePtclGrps (*integer*)
An integer flag determining whether Vorpal should consolidate particle groups.

useSegmentedMove (*integer*, *default = 1*)
Used to speed up certain types of Vorpal simulations. useSegmentedMove can be set to 1 (default) or 0. This parameter is designed to allow simulations that do NOT need the segmented move decomposition (i.e. lwfa simulations (no gridBndry) with either absorbers or periodic boundaries) to accelerate. Those simulations that use more complex boundaries like gridBndry, reflectors, etc. should use the segmented move, i.e. do not put `useSegmentedMove = 0` in your species blocks. If you choose not to use the segmented move, be aware that higher order particles will probably not work very well for the current deposition though the interpolation should be fine.

mode (*integer*)
Specifies a given mode number for which Vorpal should solve; mode number is from a specified mode expansion, for example, fields are assumed to be expressed as: $E(x, y, z) = e^{ikz}E(x, y)$, given: $E(x, y) = E_r(x, y) + iE_i(x, y)$. At present, this is only for the Species kind cmplxRelBorisDF.

maxcellxing (*integer*)
In particle simulations in which the particles could ordinarily cross more than one cell in a time step and cause out-of-bounds memory access (such as can happen in electrostatic simulations with non-relativistic dynamics), you can impose a velocity limit to prevent the out-of-bounds memory access from occurring by using the maxcellxing variable. E.g., `maxcellxing = 1` will limit the particle velocity so that the particle cannot cross more than one cell per step. That is, if for any direction i, velocity_i * dt > dx_i, then the velocity is reduced in magnitude so that velocity_i * dt<= dx_i for all i. Since imposing a velocity limit introduces error into the simulation, for cases in which you must use the velocity limit for a significant number of particles, you should lower the time step for the simulation.

labels (*stringVector*)
Specify labels for particle components. For example: `labels = [x,y,z,Red,Green,Blue]`. This is useful whenever *relBorisTagged Species Kind*, *relBorisVWTagged Species Kind*, or *relBorisVWScale Species Kind* is used, as these require special labeling for components 6-8 to display properly. See *Velocities and Internal Variables of Particles* for a table of the correct labels for each species kind.

dumpPeriodicity (*integer*, *optional*)
How often to dump the data; indicates data is to be dumped whenever the time step has increased by this amount. The command line parameter -d overrides this variable. Must have this defined in an input file or on the command line.

For more dumping options, see the Dumping Fields, Particles, and GridBoundaries section in Output Data in the VSim User Guide.

Macroparticles

Since simulating all of the physical particles in a simulation would be computationally prohibitive, Vorpal makes use of the Particle-in-Cell (PIC) model which uses macroparticles to model a large group of physical particles. This allows a particle simulation to be completed in a reasonable amount of time while still capturing any kinetic effects from the particle distribution in space and time. Depending on the type of particle species used, the ratio of physical particles to macroparticles will be fixed or it can vary from macroparticle to macroparticle. This method does not require modification of the equations of motion since the charge-to-mass ratio of the macroparticles remains fixed.

The Vorpal macroparticle model is based on PIC algorithms from Hockney and Eastwood *[HE88]* and Birdsall and Langdon *[BL91]*.

Delta-F

Some of the particle species in Vorpal make use of the Delta-F algorithm. The approximate phase-space density in a phase-space volume V_{ps} that can be found from the weights of delta-f particles using the formula

$$f = f_0 + \frac{V_{cell} \cdot \mathsf{nominalDensity}}{V_{cell} \cdot \mathsf{nomPtclsPerCell}},$$

where:

- nomPtclsPerCell are the parameters of the block <Species …>
- V_{cell} is cell volume as defined by parameters of the block <Grid> at the top level of the Vorpal input file

The Equilibrium phase-space density, f_0, is defined in the block <SVTFunc equilibDist> inside of the block <Species …> where in particular, the approximate density of the real particles in volume V can be approximated using the equation

$$\rho = \rho_0 + \frac{V_{cell} \cdot \mathsf{nominalDensity}}{V_{cell} \cdot \mathsf{nomPtclsPerCell}} \Sigma w_i,$$

where:

- the summation Σw_i happens over all particles found in the volume
- ρ_0 is the background density, which is the integral of f_0 over all possible velocities

Velocities and Internal Variables of Particles

`VelocityGenerator:` Used by **`xvLoaderEmitter`** particle sources to determine the velocities (and all internal variables) of particles when loaded or emitted into the simulation. All **`xvLoaderEmitter`** blocks must have a VelocityGenerator block. See the documentation for each velocity generator for more details and parameters.

Relativistic and Non-Relativistic Velocity

Vorpal uses either relativistic or non-relativistic velocity based on particle species:

`non-relativistic`
Simple velocity; applies to velocities for particle species:

- `nonRelBoris`

- `nonRelES`

`relativistic``
: Applies to all other particle species.

Internal Variables Associated with Particles

Internal variables associated with particles may include such quantities as particle-weight and tag. For variably-weighted particles, Vorpal tracks the weight as an additional internal variable. The velocity vector component designations are:

- **`component0:`** x coordinate
- **`component1:`** y coordinate
- **`component2:`** z coordinate
- **`component3:`** x component of the velocity or the γv.
- **`component4:`** y component of the velocity or the γv.
- **`component5:`** z-components of the velocity or the γv.
- **`component6-8:`** component**n**: Consult table below.

Velocity Vector Component Designations for component:

Particle Species	component6	component7	component8
freeRel	N/A	N/A	N/A
nonRelBoris	N/A	N/A	N/A
nonRelES	N/A	N/A	N/A
relBoris	N/A	N/A	N/A
relBorisVW	w	N/A	N/A
relBorisDF	w	N/A	N/A
relBorisBallisticVW	w	N/A	N/A
relBorisTagged	tag	N/A	N/A
relBorisVWTagged	tag	w	N/A
relBorisVWScale	tag	scale	w
relBorisEffMassExtd	valleyIndex	w	N/A
nonRelEsEffMassExtd	valleyIndex	w	N/A

Legend:

symbol	value
v	velocity
γv	gamma*v
w	weight
tag	particle tag
scale	particle scale factor

Variable Weight Particles

For the particular case of variable weight particles, you must specify one or the other, never both, of the following:

- currentDensityFunc

- Behavior of component3 in the velocity generator

One should use currentDensityFunc to set the weight of the particles in an emission source, not component3. (If you attempt to specify both a currentDensityFunc and a component3 inside a VelocityGenerator, then the value of component3 in the VelocityGenerator will be overridden by the currentDensityFunc.)

When loading particles, component3 is required to determine the weight of the particles.

Species Kinds

Species Kinds

Basic Dynamic Species kinds include those kinds listed below. Other case-specific implementations of Basic Dynamic Species kinds are discussed in Specialized Basic Dynamic Species Kinds.

Contents

- *cmplxRelBorisCylDF*
- *cmplxRelBorisDF*
  - *Example Species Kind cmplxRelBorisDF Block*
- *envBoris*
  - *envBoris Species Additional Parameters*
- *freeRel Species Kind*
- *freeRelVW Species Kind*
- *noMove Species Kind*
- *noMoveVW Species Kind*
- *nonRelBoris Species Kind*
- *nonRelES Species Kind*
- *nonRelESEffMassExtd Species Kind*
  - *nonRelESEffMassExtd Species Additional Parameters*
- *nonRelESVW Species Kind*
- *relBoris Species Kind*
- *relBorisBallisticVW Species Kind*
- *relBorisCyl Species Kind*
- *relBorisCylVW Species Kind*
- *relBorisCylVWTagged Species Kind*
- *relBorisDF Species Kind*
- *relBorisCylDF Species Kind*
- *relBorisEffMassExtd Species Kind*
  - *relBorisEffMassExtd Species Additional Parameters*
- *relBorisFuncVW Species Kind*

- *relBorisTagged Species Kind*
- *relBorisVW Species Kind*
- *relBorisVWScale Species Kind*
- *relBorisVWTagged Species Kind*

cmplxRelBorisCylDF

cmplxRelBorisCylDF

Works with VSimPD license.

Should be used in lieu of the cmplxRelBorisDF species kinds whenever polar, cylindrical, or tubular grids are used. An extension of the relBorisDF when one has made an expansion in either a single component direction in Cartesian coordinates or in the phi direction for cylindrical coordinates and then is left solving for the coefficients of that expansion. The coefficients are the real and imaginary field values.

Note: cmplxRelBorisCylDF is only permissible with `doLinearDF = true` and that it requires an understanding of setting up the proper fields in a MultiField block.

cmplxRelBorisDF

cmplxRelBorisDF

Works with VSimPD license.

An extension of the relBorisDF when one has made an expansion in either a single component direction in Cartesian coordinates or in the phi direction for cylindrical coordinates and then is left solving for the coefficients of that expansion. The coefficients are the real and imaginary field values.

Note: cmplxRelBorisDF is only permissible with `doLinearDF = true` and that it requires an understanding of setting up the proper fields in a MultiField block.

Example Species Kind cmplxRelBorisDF Block

```
<Species  electrons>
    kind = cmplxRelBorisDF
    charge = ELECCHARGE
    mass = ELECMASS
    # Must specify mode number of problem
    mode = K
    # Order matters in fields vector
    fields = [emField.nodalER emField.nodalEI emField.nodalBR emField.nodalBI]
    # Order matters in currDeps vector
    currDeps = [ JRDep JIDep ]
    bgEmField = B0
    doLinearDf = true
        .
```

(continues on next page)

(continued from previous page)

```
        .
        .
</Species>
```

envBoris

envBoris

Works with VSimPA license.

Describes particles that respond to both the Lorentz force and the ponderomotive force that arises in response to an envelope field in the laser envelope model.

In addition to the usual species parameters, an envBoris species requires some unique parameters, listed below.

Note: The envelope model is constructed for use with the MultiField input file architecture. See the *MultiField* section for more details.

envBoris Species Additional Parameters

In addition to the parameters in the list of *Species Parameters*, the envBoris Species requires:

envelopeFields (*string vector*, *required*)
The names of the active and alternate envelope fields to be used for the particle phase space advancement.

susceptibilityDep (*string*, *required*)
The name of the ScalarDepositor to be used for the plasma susceptibility in the laser envelope model.

envelopeMultiField (*string*)
The name of the MultiField containing the envelope fields to be used for particle phase space advancement.

forceField (*string*)
The name of the ponderomotive force field to be used for particle advancement.

freeRel Species Kind

freeRel

Works with VSimBase, VSimEM, VSimPD, VSimPA, and VSimMD licenses.

A free streaming particle model, used to calculate free streaming of a relativistic species. This does not add any extra parameters.

freeRelVW Species Kind

freeRelVW

Works with VSimBase, VSimEM, VSimPD, VSimPA, and VSimMD licenses.

A free streaming particle model, used to calculate free streaming of a variable-weight relativistic species. With variable weight methods, each particle has independent ratio of macroparticles to real particles. This does not add any extra parameters.

noMove Species Kind

noMove

Works with VSimBase, VSimEM, VSimPD, VSimPA, and VSimMD licenses.

Supports non-moving species. noMove can be used in a simulation to create a new non-moving particle that will always stay at the location defined by the particle source.

noMoveVW Species Kind

noMoveVW

Works with VSimBase, VSimEM, VSimPD, VSimPA, and VSimMD licenses.

Supports non-moving species with variable weight method. noMoveVW can be used in a simulation to create a new non-moving particle that will always stay at the location defined by the particle source.

nonRelBoris Species Kind

nonRelBoris`

Works with VSimBase, VSimEM, VSimPD, VSimPA, and VSimMD licenses.

Non-relativistic push using the Boris algorithm, used to calculate an electromagnetic Boris update to the particle velocities for non-relativistic particles.

nonRelES Species Kind

nonRelES

Works with VSimBase, VSimEM, VSimPD, VSimPA, and VSimMD licenses.

Calculates electrostatic updates to the particle velocities for non-relativistic particles; no magnetic fields affect the particles.

nonRelESEffMassExtd Species Kind

nonRelESEffMassExtd

Works with VSimSD licence.

Particle species that uses the electrostatic update to push electrons and holes in a diamond material. The essential difference to regular relBoris species is that in diamond, electrons and holes have effective masses that are different from their values in vacuum. Generally, the effective masses are represented by two-dimensional tensors. However, for holes the tensor is diagonal with equal diagonal elements. For electrons, the tensor is also diagonal but one of the components is different (longitudinal effective mass) from the other two (transverse effective masses). Moreover, the position of the longitudinal effective mass component depends on the conduction band valley an electron is in. The model includes six band valleys along each of the main coordinate axes in reciprocal space. The different effective masses are used both in the ballistic push and in the Monte Carlo scattering processes. The longitudinal and transverse effective masses, together with the different conduction band valleys lead to important effects when modeling electron emission with conservation of transverse electron momentum.

nonRelESEffMassExtd Species Additional Parameters

In addition to the parameters in the list of *Species Parameters*, there are:

useEffectiveMasses (*bool*, *optional*)
  If this flag is set, the particle pusher will use the effective masses when pushing particles (default is false). When it is not set, a scalar mass will be used for the electrons (the provided for the species).

isHole (*bool*, *optional*)
  The isHole flag is used to specify if the species should be treated as electrons or holes (for the purpose of scattering and what effective masses to use). Default value is false (treat the species as electrons).

scatterFlag (*bool*, *optional*)
  This flag is used to specify if the scattering should be turned on or not. Default value is 1 (which is to do the scattering). This flag is used in the low energy regime (the high energy regime is the charge generation regime while the low energy regime is the charge transport one when electrons and holes have energies in the drift-diffusion transport phase of the order of 1 eV or less).

updateDebugType (*int*, *optional*)
  In the charge generation phase, the particles are pushed with the regular particle ballistic push and the value of the parameter updateDebugType is set to 0 which is its default value. A scalar mass should be used in the charge generation phase and scattering is already taken care of via the ScatterRegion sink.

  In the low energy phase, we want to use an update method that includes the various scattering processes implemented. To do this, set updateDebugType to 2 (the value that should be used to enable a particle update with low energy scattering processes).

nonRelESVW Species Kind

nonRelESVW
  Works with VSimBase, VSimEM, VSimPD, VSimPA, and VSimMD licenses.

  Calculates electrostatic updates to the particle velocities for non-relativistic particles; no magnetic fields affect the particles. With variable weight methods, each particle has independent ratio of macroparticles to real particles. This does not add any extra parameters.

relBoris Species Kind

relBoris
  Works with VSimBase, VSimEM, VSimPD, VSimPA, and VSimMD licenses.

  Calculates an electromagnetic Boris update to the particle velocities.

relBorisBallisticVW Species Kind

relBorisBallisticVW
  Works with VSimBase, VSimEM, VSimPD, VSimPA, and VSimMD licenses.

  Creates *ballistic* particles. Particles are ballistic if they do not undergo acceleration. Using this kind requires the additional parameter moveDimVec, which specifies the directions in which the particle is allowed to move. For example, `moveDimVec = [1 0 0]` will allow movement only along the x direction, whereas `moveDimVec = [0 1 1]` will allow movement along the y and z directions. If `moveDimVec` is not specified, three-dimensional movement is the default.

relBorisCyl Species Kind

relBorisCyl

Works with VSimPD and VSimMD licenses.

Should be used in lieu of the relBoris species kinds whenever polar, cylindrical, or tubular grids are used. Provides a Boris pusher for these coordinate systems.

relBorisCylVW Species Kind

relBorisCylVW

Works with VSimPD and VSimMD licenses.

Should be used in lieu of the relBorisVW species kinds whenever polar, cylindrical, or tubular grids are used. Provides a Boris pusher with variable-weight particles for these coordinate systems.

relBorisCylVWTagged Species Kind

relBorisCylVWTagged

Works with VSimPD and VSimMD licenses.

Should be used in lieu of the relBorisVWTagged species kinds whenever polar, cylindrical, or tubular grids are used. Provides a Boris pusher with variable-weight particles for these coordinate systems. This species also allows for assigning a unique tag identifier to each macroparticle for trajectory tracking.

relBorisDF Species Kind

relBorisDF

Works with VSimPD license.

Relativistic Boris delta-F. A particle species that solves for the perturbation of the equilibrium distribution, Δf, given that $f = f_0 + \Delta f$. Removes a significant amount of noise from the particle-in-cell simulations. Requires vmin and vmax parameters and includes an optional doLinearDF flag that allows use of either linear DF or nonlinear DF.

relBorisCylDF Species Kind

relBorisDF`:
Works with VSimPD license.

Should be used in lieu of the relBorisDF species kinds whenever polar, cylindrical, or tubular grids are used. A relativistic Boris delta-F. A particle species that solves for the perturbation of the equilibrium distribution, Δf, given that $f = f_0 + \Delta f$. Removes a significant amount of noise from the particle-in-cell simulations. Requires vmin and vmax parameters and includes an optional doLinearDF flag that allows use of either linear DF or nonlinear DF.

relBorisEffMassExtd Species Kind

relBorisEffMassExtd

Works with VSimSD licence.

Particle species that uses the electromagnetic Boris update to push electrons and holes in a diamond material. The essential difference to regular relBoris species is that in diamond, electrons and holes have effective masses that are different from their values in vacuum. Generally, the effective masses are represented by two-dimensional tensors. However, for holes the tensor is diagonal with equal diagonal elements. For electrons, the tensor is also diagonal but one of the components is different (longitudinal effective mass) from the other two (transverse effective masses). Moreover, the position of the longitudinal effective mass component depends on the conduction band valley an electron is in. The model includes six band valleys along each of the main coordinate axes in reciprocal space. The different effective masses are used both in the ballistic push and in the Monte Carlo scattering processes. The longitudinal and transverse effective masses, together with the different conduction band valleys lead to important effects when modeling electron emission with conservation of transverse electron momentum.

relBorisEffMassExtd Species Additional Parameters

In addition to the parameters in the list of *Species Parameters*, there are:

`useEffectiveMasses` (*bool*, *optional*)
If this flag is set, the particle pusher will use the effective masses when pushing particles (default is false). When it is not set, a scalar mass will be used for the electrons (the provided for the species).

`isHole` (*bool*, *optional*)
The isHole flag is used to specify if the species should be treated as electrons or holes (for the purpose of scattering and what effective masses to use). Default value is false (treat the species as electrons).

`scatterFlag` (*bool*, *optional*)
This flag is used to specify if the scattering should be turned on or not. Default value is 1 (which is to do the scattering). This flag is used in the low energy regime (the high energy regime is the charge generation regime while the low energy regime is the charge transport one when electrons and holes have energies in the drift-diffusion transport phase of the order of 1 eV or less).

`updateDebugType` (*int*, *optional*)
In the charge generation phase, the particles are pushed with the regular particle ballistic push and the value of the parameter updateDebugType is set to 0 which is its default value. A scalar mass should be used in the charge generation phase and scattering is already taken care of via the ScatterRegion sink.

In the low energy phase, we want to use an update method that includes the various scattering processes implemented. To do this, set updateDebugType to 2 (the value that should be used to enable a particle update with low energy scattering processes)

relBorisFuncVW Species Kind

`relBorisFuncVW`
Works with VSimPD and VSimMD licenses.

Particle species that uses the electromagnetic Boris update to the particle velocities where the macroparticle to real particle ratio is controlled by a user-specified STFunc. Results using this species are only valid when the particle positions are displaced by a small amount from a equilibrium position by the electromagnetic fields. Useful for modeling the appearance of a plasma whose general time and density dependence are known. Requires STFunc block of arbitrary name that sets the weight of the particles; this one required block is unique to the species.

relBorisTagged Species Kind

relBorisTagged

Works with VSimBase, VSimEM, VSimPD, VSimPA, and VSimMD licenses.

Applies an electromagnetic Boris update to the particle velocities. This species assigns a unique tag identifier to each macroparticle for trajectory tracking.

Note: A Species of `kind = relBorisTagged` should include the argument `labels = [x,y,z,ux,uy,uz,tags]` in order to correctly label all components in the output.

---

relBorisVW Species Kind

relBorisVW

Works with VSimPD and VSimMD licenses.

Implementation of the relBoris update includes updates specific to variable weight algorithms. With variable weight methods, each particle has independent ratio of macroparticles to real particles.

relBorisVWScale Species Kind

relBorisVWScale

Works with VSimPD and VSimMD licenses.

Used to scan multiple power levels searching for multipacting resonances. Each particle has a scaling parameter that multiplies the electromagnetic field, allowing multiple power or voltage levels to exist in a simulation simultaneously. The use of multiple coexisting power or voltage levels is only consistent with small amounts of charge and current which have little to no effect on the applied fields. Thus, generally speaking, this species is not expected to use or require charge and/or current depositors. This species assigns a unique tag identifier to each macroparticle for trajectory tracking.

Note: The `currDeps` and `chargeDeps` arguments may be neglected if `kind = relBorisVWScale`.

Note: A Species of `kind = relBorisVWTagged` should include the argument `labels = [x,y,z,ux,uy,uz,tags,scale,weight]` in order to correctly label all components in the output.

---

relBorisVWTagged Species Kind

relBorisVWTagged

Works with VSimBase, VSimEM, VSimPD, VSimPA, and VSimMD licenses.

Implementation of the relBoris updater that includes updates specific to variable weight algorithms. In variable weight methods, each particle has an independent ratio of macroparticles to real particles. This species also allows for assigning a unique tag identifier to each macroparticle for trajectory tracking.

Note: A Species of `kind = relBorisVWTagged` should include the argument `labels = [x,y,z,ux,uy,uz,tags,weight]` in order to correctly label all components in the output.

---

cellSpecies (deprecated in 8.0)

Along with interpolation, acceleration, and deposition implementations, cellSpecies provides another particle data structure that improves performance of particle-dominated plasma simulations. Simulations that feature regions with high particles-per-cell counts will benefit from using cellSpecies.

By maximizing data reuse on a per-cell basis, cellSpecies simulation performance improves on that of simulations using analogous Basic Dynamic Species with the standard particle data infrastructure. When used under the conditions listed below, cellSpecies implementation-specific kinds accelerate simulations. For example, the performance benefit of using the cellSpecies `kind = cell` instead of the Basic Dynamic Species `kind = relBoris` can be significant.

For electrostatic simulations, Tech-X recommends setting the value of `.maxcellxing` to `1`. Defining `maxcellxing = 1` ensures that particle velocities will always be less than the velocity required to cross more than one cell boundary at a time. Use of cellSpecies in electrostatic simulations without maxcellxing is allowed, but may cause undesirable results – the program may encounter an error if a particle gains sufficient velocity such that it moves more than one cell in any direction in a single time step.

When to Use cellSpecies Kinds

The use of cellSpecies kinds to achieve improved performance depends on the types of other features also used in the simulation. cellSpecies supports:

- All particle sinks described in *ParticleSink*
- All Monte Carlo interactions described in *MonteCarloInteractions*
- Moving window simulations (available via the top-level simulation parameter `downShiftDir`)

cellSpecies does not support:

- comboEmFields
- Segmented moves

Note: Because cellSpecies does not have a sorting function, it is a good idea to disable particle sorting (for example, by using the `-ns` flag on the command line). Otherwise, use of the ptclSort flag will cause MPI-related program crashes on some systems.

Using the cell kind, parameters specific to cellSpecies

A new input file syntax can be used with cellSpecies where the kind of the species block is always cell. Different options for the species are determined by additional parameters:

`kind` (*string*)
This parameter is set to cell (as opposed to sortSpecies).

`pusher` (*string*)
Determines the kind of pusher to use to advance the particle positions and velocities.

`relBoris`

Relativistic Boris push.

`nonRelBoris`

Non-relativistic Boris push.

`nonRelEs`

Non-relativistic electrostatic push.

`freeRel`

Free streaming particles with relativistic velocities.

`ballisticX`

Push particles in the `x` direction only, with no acceleration.

`pulseLaunch`

Use this pusher to launch a laser pulse from a moving plane. The particles must be set up on a plane perpendicular to the direction of propagation of the laser pulse. The shape of the laser pulse is set by a functional field which is referred to in the `fields` parameter. Two additional parameters must be used with this `pusher`:

`vmax`

Maximum velocity reached by the particle, typically a small fraction of the speed of light

`Emax`

Maximum amplitude of the electric field applied to the species

`stencil` (*string*)
Interpolation method to use in charge density and current deposition.

`spline1stOrder`

Use linear interpolation.

`spline2ndOrder`

Use piecewise parabolic interpolation.

`variableWeightParticle` (*integer*)
Whether to use variable weight particles (as opposed to constant weight).

`taggedParticle` (*integer*)
Whether to use tagged particles.

`ptclSubGroupCapacity`
Represents the size (in number of particles) of the fundamental particle data structure per cell. The default value is `64`. Although this parameter is strongly simulation-dependent, the basic rule for determining the value for this parameter is to set ptclSubGroupCapacity equal to 150% of the typical value of particles-per-cell. This flag can have a large impact on the performance of the cellSpecies algorithms; unless the simulation domain is exceedingly large, using too large a value for ptclSubGroupCapacity is preferable to using too small a value. The default value `64` is well-suited to plasmas with an average number of macroparticles per cell on the order of `40`.

Example cellSpecies Block

```
<Species electrons>

  kind = cell
  pusher = nonRelES
  stencil = spline1stOrder
```

(continues on next page)

(continued from previous page)

```
  charge = -1.6022e-1
  mass = 9.109e-31
  emField = myEmField
  ptclSubGroupCapacity = 64

  maxcellxing=1

  nominalDensity = 3e+22
  nomPtclsPerCell = 40.

  ...

</Species>
```

See also

- *EmField*
- *EmField Parameters*

3.11.2 ParticleSink Blocks

ParticleSink

Nested block to create particle boundary for a **`Species`**. Typically, *ParticleSinks* removes particles in a boundary region although other behaviors are possible.

Multiple types of particle sinks may be specified in the input file, and you can specify the locations of these sinks within multiple input blocks. If you specify multiple particle sinks for the same grid cell, only one of them will be created. In the case of overlapping physical sinks, the one specified in the last sink input block within the corresponding species block is the one that Vorpal creates. Vorpal issues a warning when it detects overlapping particle sinks.

Any grid cell may contain both a messaging sink and a physical sink. The messaging sinks are auto-generated by Vorpal during simulation setup, and they are used to enforce periodic boundary conditions on particles, to communicate particles between processors in parallel runs, etc. The physical sinks are used to absorb particles that strike a boundary, leave a specified region of the grid, or for more specialized purposes. In most cases, you have complete control over the type and location of physical sinks. Vorpal checks physical sinks first in each cell, after which the messaging sinks are applied.

You must set up the particle boundary conditions so that they completely surround the space in which particles are loaded. Otherwise, particles will drift out of the simulation and try to reference fields that do not exist. This leads to a segmentation fault when Vorpal runs the simulation. The most common cause of Vorpal crashes is improperly set up particle boundary conditions.

If you do not explicitly specify all particle boundary conditions with **ParticleSink** input blocks, you must specify periodic boundary conditions with the *`periodicDirs`* parameter in the **`Decomp`** input block, allowing particles to “wrap around” to the opposite side of the simulation.

Note: The interplay of messaging sinks and physical sinks is important to understand when you create a species input block that includes the specification of particle absorbers around the domain (typically in the guard cells). When specifying periodic boundary conditions with the syntax `periodicDirs = [1 1 1]` in the **`Decomp`** block, be aware that Vorpal implements this directive for all particle species by auto-generating messaging sinks around the simulation domain (in the guard cells). Hence, this requested boundary condition will be overridden for any species that explicitly specifies absorbers in these same guard cells.

ParticleSink Kinds

AbsAndSav

Works with VSimBase, VSimEM, VSimPD, VSimPA, and VSimMD licenses.

This particle sink removes particles at boundary, but stores them for use by other code components such as **History**.

AbsAndSav Parameters

minDim (*integer*)
Minimum dimensionality for which this sink is applicable (1, 2, or 3).

lowerBounds (*integer vector*)
Gives lower bounds of the particle sink in cell indices.

upperBounds (*integer vector*)
Gives upper bounds of the particle sink in cell indices.

AbsAndSav Example Block

```
 <ParticleSink keeperSideWallXeDblIons>
  kind = absAndSav
  lowerBounds = [CK2_ZMIN_INDX CK2_RMIN_INDX]
  upperBounds = [CK2_ZMAX_INDX CK2_RMAX_INDX]
</ParticleSink>
```

AbsCutCell

Works with VSimPD and VSimMD licenses.

This particle sink removes particles at cut-cell (conformal) boundary, deleting them from the simulation.

AbsCutCell Parameters

minDim (*integer*)
Minimum dimensionality for which this sink is applicable (1, 2, or 3).

lowerBounds (*integer vector*)
Gives lower bounds of the particle sink in cell indices.

upperBounds (*integer vector*)
Gives upper bounds of the particle sink in cell indices.

gridBoundary (*string*)
Name of the gridBoundary in the simulation.

useCornerMove (*string*, *default = true*)
Set to `true` to specify corner move dynamics when removing the particle at a cut cell. If `false`, uses the parallel move from 1 cell deep. Using `true`, e.g., corner move, is more robust in some circumstances.

AbsCutCell Example Block

```
<ParticleSink absorber>
 kind = absCutCell
 minDim = 3
 gridBoundary = diodecavity
 lowerBounds = [0  1  0]
 upperBounds = [NX_TOT  2  NZ_TOT]
</ParticleSink>
```

absorber

Works with VSimBase, VSimEM, VSimPD, VSimPA, and VSimMD licenses.

This particle absorber kind absorbs particles at boundary, removing them from the simulation. They will not be saved.

absorber Parameters

minDim
Minimum dimensionality for which this sink is applicable (1, 2, or 3).

lowerBounds
Gives lower bounds of the particle sink in cell indices.

upperBounds
Gives upper bounds of the particle sink in cell indices.

absorber Example Block

```
<ParticleSink rightAbsorber>
 kind = absorber
 minDim = 1
 # Bounds specified with physical indexing
 lowerBounds = [NX -1 -1]
 upperBounds = [NX1 NY1 NZ1]
</ParticleSink>
```

AbsSavCutCell

Works with VSimPD and VSimMD licenses.

This particle sink combines the functions of the `absAndSav` and `absCutCell` sinks. When using this sink one should avoid having the grid boundary be inline exactly with the computational mesh as this may prevent some removed particles from being available for use by other objects.

absSavCutCell Parameters

minDim (*integer*)
: Minimum dimensionality for which this sink is applicable (1, 2, or 3).

lowerBounds (*integer vector*)
: Gives lower bounds of the particle sink in cell indices.

upperBounds (*integer vector*)
: Gives upper bounds of the particle sink in cell indices.

gridBoundary (*string*)
: Name of the gridBoundary the absorber is applied to. When using one should avoid having the grid boundary be inline exactly with the computational mesh as this may prevent some removed particles from being available for use by other objects.

useCornerMove (*string*, *default = true*)
: Set to `true` to specify corner move dynamics when removing the particle at a cut cell. If `false`, uses the parallel move from 1 cell deep. Using `true`, e.g., corner move, is more robust in some circumstances.

absSavCutCell Example Block

```
<ParticleSink leftAbsorber>
  kind = absSavCutCell
  # Bounds specified with physical indexing
  lowerBounds = [0 -1 -1]
  upperBounds = [2 NY1 NZ1]
  gridBoundary = mybndry
</ParticleSink>
```

absSavTriCutCell

Works with VSimPD and VSimMD licenses.

This particle sink removes particles at a cut-cell (conformal) boundary, deleting them from the simulation, and (if requested) recording them. This sink absorbs particles on the triangulated surface (a set of triangles approximating the surface).

When possible (see below), this cut-cell sink calculates and records the exact time that particles cross the boundary, as well as the surface normal and grid cell. (Other cut-cell sinks currently do not do this.) This makes it possible, e.g., for *secondaryEmitter* to emit a secondary particle with the exact remaining time in the timestep. Thus a *secondaryEmitter* that preserves particle properties (but perhaps switches species) will not change a particle's trajectory.

In principle, this sink removes particles that cross the surface into the absorber (specified by the `gridBoundary` and `absorberIsInGridBoundary` options). However, in some cases, the behavior is more complicated. If a particle is initially emitted inside the absorber, it typically gets one time step to escape the absorber; if it doesn't escape, then the particle is absorbed at its initial location (which may not be near the absorber's surface).

When recording absorbed particles, be aware that particles that are immediately absorbed after being secondary-emitted (based on previously-absorbed primary particles) may not be recorded as absorbed (hence further secondaries won't be emitted). If this is a problem, it is in cases possible to fix it by making the secondary particles a different species.

absSavTriCutCell Parameters

minDim (*integer*, *optional*, *default = 1*)
: Minimum dimensionality for which this sink is applicable (1, 2, or 3).

lowerBounds (*integer vector*)
: Gives lower bounds of the particle sink in cell indices.

upperBounds (*integer vector*)
: Gives upper bounds of the particle sink in cell indices.

gridBoundary (*string*)
: Name of the gridBoundary that the describes the surface of absorption.

absorberIsInGridBoundary (*boolean*, *optional*, *default = false*)
: Whether the absorbing material is the interior of the *gridBoundary* (if `true`), or in the exterior (if `false`).

depositCurrentToCorner (*boolean*, *optional*, *default = true*)
: This option should generally be `true` in electromagnetic simulations, especially for absorption at metallic (conducting) surfaces; it avoids the creation of artificial stationary charges. In electromagnetic simulation, absorbing a charge in the middle of a cell automatically leaves (or rather, leads to electromagnetic fields that act as if there were) a charge that remains at the emission location forever. If this option is `true`, then electrical current will be deposited from the absorption location to the nearest cell-corner that is inside the absorber.

recordParticleData (*boolean*, *optional*, *default = true*)
: Whether absorbed particles should be recorded (either for use in Histories or for secondary emission). The record is cleared after each time step.

absorbLocation (*string*, *optional*, *default = justOutsideAbsorber*)
: Where particles should be absorbed, when they are absorbed "at" the surface: `justOutsideAbsorber`, `justInsideAbsorber`, or `nearestTheSurface` (i.e., either inside or outside the absorber). In pathological cases, this request may not be honored.

absSavTriCutCell Example Block

```
<ParticleSink coneSurfSink>
  kind = absSavTriCutCell
  recordParticleData = true
  gridBoundary = coneSurf
  lowerBounds = [0 0 0]
  upperBounds = [8 8 8]
  depositCurrentToCorner = true
</ParticleSink>
```

AbsSaveDump

works with VSimBase, VSimEM, VSimPD, VSimPA, and VSimMD licenses.

This particle sink creates a region similar to `absAndSav`, however the particles' data are also written to a file. The `absSaveDump` is useful for writing all the particle data into an array. However, if you really want to output the values of a single coordinate (i.e. the x position, or the y-momentum) of the particles that cross into a particle sink, you can use the **`History`** feature instead.

The `absSaveDump` particle sink has the additional feature that those particles are saved to a file. The saved output file can then be used outside of VSim as input to another program. Originally absSaveDump was used to make a file record of particles crossing a surface, so that those particles could be used as input for a particle mapping program.

The saved particle information is, with one exception, in the standard VSim particle dump format: position (or displacement), followed by momentum then other variables such as the weight. The exception is that the displacement in the direction perpendicular to the sink surface is instead given as the time at which the particle crossed into the sink region, for example: (t, x2, x3, p1, p2, p3, weight). All values are given as differences relative to the first dumped particle, which acts as a reference particle.

Note: As in the case of the absAndSav particle sink, you can specify whether particles are actually absorbed or just dumped by setting `removePtclFlag = 0` in the input file.

AbsSaveDump Parameters

minDim (*integer*)
Minimum dimensionality for which this sink is applicable (1, 2, or 3).

lowerBounds (*integer vector*)
Gives lower bounds of the particle sink in cell indices.

upperBounds (*integer vector*)
Gives upper bounds of the particle sink in cell indices.

fluxSpeciesDumpName (*string*)
Sets the name of the HDF5 dataset which the flux emitter class will read in. Also sets part of the file name to be read in. The default is the name of the species (given in the species block header) plus flux. The default value serves for most purposes.

useRunNameAsPrefix (*integer*)
If `1`, the flux files dumped by the sink will be of form <*runName*><*fluxSpeciesDumpName*>_<*#*>.h5. If `0`, the flux files dumped by the sink will be of form <*fluxSpeciesDumpName*>_<*#*>.h5

independentDumpIndicing (*string*, *default = 0*)
Determines whether to use the same dump index as the rest of Vorpal or set an independent dump index. Allowed values are `0` or `1`.The default value is `0`, which causes the global Vorpal dump index to be used. Use a different index if you want to sequence the resulting files separately for use by an external program. For example, if the simulation dump name is `test2`, the default flux file dumped by an `absSaveDump` sink within a species named `electrons` will be called `test2_electronsFlux_0.h5`.

absSaveDump Example Block

```
<ParticleSink  redElecSwSwitchAbsorber>
    kind = absSaveDump minDim = 1
    # base name of dump files
    fluxSpeciesDumpName = redElectronsFlux
    # Whether to use global Vorpal dump index
    independentDumpIndicing = 1
    # Set whether particles should be removed (absorbed)
    # after being recorded
    #doNotRemoveParticlesFlag = 1
    # Bounds specified
    lowerBounds = [20 0 0] upperBounds = [21 20 20]
</ParticleSink>
```

absStairStep

Works with VSimBase, VSimEM, VSimPD, VSimPA, and VSimMD licenses.

This particle sink removes the particles once they cross the stair step boundary that corresponds to the conformal boundary. It will work with a coordProdGrid.

absStairStep Parameters

minDim (*integer*)
Minimum dimensionality for which this sink is applicable (1, 2, or 3).

lowerBounds (*integer vector*)
Gives lower bounds of the particle sink in cell indices.

upperBounds (*integer vector*)
Gives upper bounds of the particle sink in cell indices.

gridBoundary (*string*)
The name of the stair-stepped grid boundary.

absStairStep Example Block

```
<ParticleSink stairStepBndry>
  kind = absStairStep
  gridBoundary = sphere
  lowerBounds = [-1   -1  -1]
  upperBounds = [NX1 NY1 NZ1]
</ParticleSink>
```

absSavStairStep

Works with VSimBase, VSimEM, VSimPD, VSimPA, and VSimMD licenses.

This particle sink removes the particles once they cross the stair step boundary that corresponds to the conformal boundary. It saves the particle information so it can be used in other blocks such as history blocks. It will work with a coordProdGrid.

absStairStep Parameters

minDim (*integer*)
Minimum dimensionality for which this sink is applicable (1, 2, or 3).

lowerBounds (*integer vector*)
Gives lower bounds of the particle sink in cell indices.

upperBounds (*integer vector*)
Gives upper bounds of the particle sink in cell indices.

gridBoundary (*string*)
The name of the stair-stepped grid boundary.

absStairStep Example Block

```
<ParticleSink stairStepBndry>
  kind = absSavStairStep
  gridBoundary = sphere
  lowerBounds = [-1   -1  -1]
  upperBounds = [NX1 NY1 NZ1]
</ParticleSink>
```

DiffuseBndry

Works with VSimBase, VSimEM, VSimPD, VSimPA, and VSimMD licenses.

This particle sink reflects particles with velocity taken from thermal distribution.

diffuseBndry Parameters

minDim (*integer*)
Minimum dimensionality for which this sink is applicable (1, 2, or 3).

lowerBounds (*integer vector*)
Gives lower bounds of the particle sink in cell indices.

upperBounds (*integer vector*)
Gives upper bounds of the particle sink in cell indices.

direction (*double vector*)
Inward normal from the reflecting surface.

surface (*string*)
Physical location of the reflecting surface.

vsig (*double vector*)
Thermal velocities at the boundary.

acc (*float*)
The thermal accommodation coefficient.

useObliqueReflection (*integer*)
A flag, if set sink will work with oblique boundaries.

diffuseBndry Example Block

```
<ParticleSink rightplate>
  kind = diffuseBndry
  direction = [1. 0. 0.]
  surface = X_RIGHT_WALL
  vsig = [VEL_373K VEL_373K VEL_373K]
  lowerBounds = [$NX-NX_SINK$   -1   -1]
  upperBounds = [$NX+1$ NY_P NZ_P]
</ParticleSink>
```

SpecularBndry

Works with VSimBase, VSimEM, VSimPD, VSimPA, and VSimMD licenses.

This particle source specularly reflects particles at one or more planes defined by the user. Generally, one defines either a single plane as a ‘plate’, two intersecting planes as a ‘rail’, and three intersecting planes as a ‘corner’. To create a full reflecting box, you need six plates, eight rails and eight corners. It is best practice to define the plates first, followed by the rails and finally the corners.

specularBndry Parameters

`minDim` (*integer*)
Minimum dimensionality for which this sink is applicable (1, 2, or 3).

`lowerBounds` (*integer vector*)
Gives lower bounds of the particle sink in cell indices.

`upperBounds` (*integer vector*)
Gives upper bounds of the particle sink in cell indices.

`numSurfaces` (*integer; 1, 2 or 3*)
Number of reflecting surfaces in the sink. One surface corresponds to a plate. Two surfaces correspond to a rail and three surfaces correspond to a corner.

`direction` (*double vector*)
The direction of the outward normal for the reflecting plane in the case that *`numSurfaces`* = 1.

`xdirection` (*double vector*)

and

`ydirection` (*double vector*)

and

`zdirection` (*double vector*)
The direction of the outward normal for the reflecting plane in the x,y,z direction in the case that *`numSurfaces`* = 2 or 3.

`surface` (*double*)
The location of the reflecting plane in the case that *`numSurfaces`* = 1.

`xsurface` (*double vector*)

and

`ysurface` (*double vector*)

and

`zsurface` (*double vector*)
The location of the reflecting plane in the x,y,z direction in the case that *`numSurfaces`* = 2 or 3.

specularBndry Example Block

```
<ParticleSink leftbottomrail>
  kind=specularBndry
  minDim=3
  numSurfaces=2
```

(continues on next page)

(continued from previous page)

```
  xdirection=[-1. 0. 0.]
  xsurface=0.
  zdirection=[0. 0. -1.]
  zsurface=0.
  lowerBounds=[0 0 0]
  upperBounds=[1 NY 1]
</ParticleSink>
```

transparentBndry

In this particle sink, particles are either transferred through or reflected at the boundaries. The fraction of particles that are either transferred or reflected through the boundary is based on the transparency parameter which is expected to be between 0 and 1.0. Those that are not transferred or reflected are absorbed by the boundary. For every particle at the boundary, a random number is picked and compared with the given transparency parameter.

In transmit mode if the random number is below the transparency parameter, the particle is transferred cross the boundary. For example, if transparency = .75, 75% of particles are transmitted through the boundary.

In reflect mode if the random number is above the transparency parameter, the particle is reflected at a user specified velocity. For example, if transparency = .75, 25% of particles are reflected by the boundary.

transparentBndry Parameters

`minDim` (*integer*)
Minimum dimensionality for which this sink is applicable (1, 2, or 3).

`lowerBounds` (*integer vector*)
Gives lower bounds of the particle sink in cell indices.

`upperBounds` (*integer vector*)
Gives upper bounds of the particle sink in cell indices.

`direction` (*double vector*)
The direction of the outward normal for the reflecting plane in the case that particle is reflected at the boundary.

`surface` (*double*, *required*)
The location of the reflecting plane in the case that particle is reflected at the boundary.

`transparency` (*double*, *required*)
The transparency parameter which is a value between 0 and 1.0. Particle at the boundary are transferred through the boundary or reflected at the boundary based on the transparency parameter and operating mode.

`absTransparentPtclFlag` (*flag*, *optional*, *default = false*)
A flag to enable transmit mode of the boundary. Cannot be set to true if reflectPtcls is also true.

`reflectPtcls` (*flag*, *optional*, *default = false*)
A flag to enable reflect mode of the boundary. Cannot be set to true if absTransparentPtclFlag is also true. If set the average and thermal velocities of the reflected particles are given by vbar and vsig respectively.

`vbar` (*double vector*)
Average velocity of the particles in the x, y, and z directions. Only applied to reflected particles.

`vsig` (*double vector*)
Positive value denoting the thermal velocity in the x, y, and z directions. Only applied to reflected particles.

transparentBndry Example Block

```
<ParticleSink rightXeIonAbsorber>
  kind = transparentBndry
  # Bounds specified with physical indexing
  lowerBounds = [ABS_LOC_LB -1 -1]
  upperBounds = [ABS_LOC_UB NY1 NZ1]
  direction = [-1.0 0.0 0.0]
  surface = $XMIDm1 + 0.25*DX$
  # this is the param that decides percentage transmitted, or absorbed
  transparency = 0.8
  # CANNOT HAVE absTransparentPtclFlag and reflectPtcls = true.
  # if true this will absorb percentage particles  = transparency
  # absTransparentPtclFlag = true
  # if true the percentage reflected particles = 1 - transparancey
  reflectPtcls = true
  # thermal velocity of the reflected particles (so only applies if reflectPtcls =␣
↪true)
  vsig = [VTH_TRNS VTH_TRNS VTH_TRNS]
  # average velocity of the reflected particles (so only applies if reflectPtcls =␣
↪true)
  vbar = [-1.e6 0.0 0.0]
</ParticleSink>
```

zeroWgtAbsorber

Works with VSimBase, VSimEM, VSimPD, VSimPA, and VSimMD licenses.

This particle sink is designed to be used in combination with **Collision** blocks. When variable weight particles combine during a collision, some particles in the simulation have zero weight. The `zeroWgtAbsorber` is used to remove these (non-physical) particles from the simulation.

zeroWgtAbsorber Parameters

minDim (*integer*)
Minimum dimensionality for which this sink is applicable (1, 2, or 3).

lowerBounds (*integer vector*)
Gives lower bounds of the particle sink in cell indices.

upperBounds (*integer vector*)
Gives upper bounds of the particle sink in cell indices.

zeroWgtAbsorber Example Block

```
<ParticleSink myZeroWgtAbsorber>
  kind = zeroWgtAbsorber
  minDim = 1
  lowerBounds = [0 0 0]
  upperBounds = [NX NY NZ]
</ParticleSink>
```

3.11.3 ParticleSource Blocks

ParticleSource

`ParticleSource`:

ParticleSource blocks must be declared within a particle species block in order to load or emit particles of that species. VSim has many allowable ParticleSource kinds:

xvLoaderEmitter
randDensSrc
bitRevDensSrc (deprecated in 8.0)
bitRevDensSrcVW (deprecated in 8.0)
gridDenSrcVW (deprecated in 8.0)
fileDensSrc
manualSrc
gaussSource
boostLoader
gridLoader
fluxPtclEmitter
switchSpeciesSrc
gaussDensSrc (deprecated in 8.0)
planarPtclEmitter (deprecated in 8.0)
secElec
secFieldEmitter
secondaryEmitter
simpleSec
sputter
userDefinedSecElec
userFuncSecElec

Using macroDensFunc or relMacroDenFunc

When `macroDensFunc` or `relMacroDenFunc` is used, please note that *density* indirectly specifies the total number of attempts to load a particle. If *macroDensFunc* determines that half of the particle load attempts will be successful (as with *y/LY* over the domain {0,LY}), density should be double the number density one actually wishes to appear in the simulation. If one uses a more complex function, for example Gaussian, this normalization factor may take a more complex form.

Working in Cylindrical Coordinates

Care must be taken with 2D ZR cylindrical coordinate systems. The loading algorithm will attempt to load particles almost uniformly in both of the dimensions. It does not take into account the larger volume/area represented at large radius compared with lower radius, so if we allow the particles to uniformly fill this space and allow them to be equally weighted, we end up with a high density of macro- and physical particles near the axis.

To learn how to compensate for this, please see: cylindricalparticles.

ParticleSource parameters

recordParticleData (*boolean*, *optional*, *default = false*)

If `true`, then at each time step all sourced particles will be recorded; they can be accessed via histories (e.g., *speciesAbsPtclData2* or *speciesBinning*) or, e.g., for secondary emission (*secondaryEmitter*). The record is cleared after each time step.

ParticleSource Kinds

Generalized Sources:

xvLoaderEmitter

A generic, flexible particle source designed to provide a wide variety of options for creating particles either by *density-loading* in a volume and/or *flux-emitting* from a surface. This particle source requires the specification of at least two sub-blocks; a position generator and a velocity generator.

This is the most commonly used particle source in VSim.

This particle source is available with any VSim license.

xvLoaderEmitter Sub-Blocks

PositionGenerator (block)

Is used to generate physical positions of particles when they are loaded or emitted.

VelocityGenerator (block)

Is used to generate velocities of a particle when loaded or emitted.

In addition to defining the **PositionGenerator** and **VelocityGenerator** blocks, you can also define Space-Time functions to specify a non-Cartesian shape for loading and emission, respectively.

STFunc Block (optional block)

<STFunc relMacroDenFunc> (default = 1): The value for **relMacroDenFunc** should be a number between `0` and `1`, inclusive. A uniform random number is selected for each *particle load attempt*, and if the random number is less than the value of **relMacroDenFunc**, then it is loaded, otherwise it is not loaded. Use this to create a non-uniform loading density. This function is ignored when only flux-emitting (load = false).

<STFunc relMacroFluxFunc> (default = 1): The value for **relMacroFluxFunc** should be a number between `0` and `1`, inclusive. A uniform random number is selected for each *particle emission attempt*, and if the random number is less than the value of **relMacroFluxFunc**, then it is emitted, otherwise it is not emitted. Use this to create a non-uniform emission density. This function is ignored when only density-loading (emit = false).

<STFunc currentDensityFunc>: This STFunc block determines the weight of variable weight particles when emitting. Causes an exception if used with fixed-weight particles. If this STFunc is omitted in variable-weight particles emission, then a default constant currentDensityFunc is set up inside Vorpal based on the emission parameters, where the current density amplitude is calculated as below:

$J_0 = charge * numPtlcsInMacro * nomMacroPtclsPerAreaStep/dt$

Here charge is the charge value specified by the user in the `Species` block, nomMacroPtclsPerAreaStep is the nominal macro particles emitter per time step per emission area. For zero charge neutral species type, equivalent charge value is used in above calculation. The nomMacroPtclsPerAreaStep is calculated in Vorpal based on the user emission parameter such as `sweepRate` or `nomMacroPtclsPerStep`. See `PositionGenerator` for details. This ensures that the emitted particle has a weight value of 1.0.

xvLoaderEmitter Parameters

`load` (string, default = `true`): Used to invoke density-loading features.

`emit` (string, default = `true`): Used to invoke flux-emitting features.

`applyTimes`: Interval over which particles are created.

`loadOnShift` (string, default = `false`): A string, `true` or `false`, used in shifting window simulations to insure that particles are loaded into the new cells added when the simulation shifts. **`loadOnShift`** should be set to true if a moving window is used in the simulation. If true, then must also have `load = true`.

`loadAfterInit` (string, default = `false`): Used to continue loading particles after the initialization, for example a continuous bulk source ionization. **`loadAfterInit`** should be set to true only if more particles should be loaded into the **`loadSlab`** with each time step. If true, then must also have `load = true`.

`useCornerMove` (string, default = `true`): Set to `true` to specify corner move dynamics on the initial step, when emitting from cut cells. If `false`, uses the parallel move from 1 cell deep. Using `true`, e.g., corner move, is more robust in some circumstances. Ignored if not using a `kind = cutCellPosGen` type of <PositionGenerator>.

`useStairStepMove` (string, default = `false`): Set to `true` if moving particles from exactly on the emission surface. Primarily used for debugging, since non-charge conserving for emission from metallic surfaces. May have applications when emitting from dielectric surfaces, but not yet fully explored.

`emitBasedOnLocalForce` (string, default = `false`): Set to `true` in order to prevent emission when the local electric field has sign such that the force returns particles to the surface.

`thermalLoad` (string): Allows Vorpal to simulate a thermal, constant density plasma in a finite domain, without using periodic boundary conditions. To invoke the loader on the left boundary of the x-axis, within a ParticleSource block of `kind = xvLoaderEmitter`, specify the desired load region (e.g. the left-most layer of cells) and set the flag `thermalLoadXLeft = true`. The other five flags are used in an analogous way, each within a separate ParticleSource block. The **`thermalLoad`** flags are automatically consistent with any spatial or velocity distribution that is supported by `kind = xvLoaderEmitter`. At the beginning of any time step that falls within the **`applyTimes`** = [. . .] limits, particles will be loaded into the specified region as though they had actually been loaded in the corresponding region that is one cell to the left (or right) along the x (or y or z) axis, and then drifted into the simulation domain. Use restricted to electrostatic PIC simulations. Flags include:

- `thermalLoadXLeft`
- `thermalLoadXRight`
- `thermalLoadYLeft`
- `thermalLoadYRight`
- `thermalLoadZLeft`
- `thermalLoadZRight`

Also See

See *ParticleSink*

randDensSrc

This particle source is an emitter, which uses bit-reversal with a standard pseudo-random number generator. While this source was originally designed for Cartesian simulations, it may also be used in a ZR cylindrical coordinate system.

This particle source is available with a VSimMD or VSimPD license.

randDensSrc Sub-Blocks

STFunc Block (optional block):

`macroDensFunc` (*default = 1*)
The value for `<STFunc macroDensFunc>` should be a number between `0` and `1`, inclusive. A uniform random number is selected for each *particle load attempt*, and if the random number is less than the value of `macroDensFunc`, then it is loaded, otherwise it is not loaded. Use this to create a non-uniform loading density.

NAFunc Block (optional block):

`velocitySequence_0`
The `<NAFunc velocitySequence_0>` block defines sequences of numbers to represent the physical characteristics of particles defined by a ParticleSource. Initial particle velocities, for example, can be defined using an `NAFunc` which samples values from an analytic velocity distribution. An `NAFunc` block can be named one of:

- velocitySequence_0
- velocitySequence_1
- velocitySequence_2
- velocitySequence_3
- velocitySequence_4
- velocitySequence_5

The `velocitySequence_0`, `velocitySequence_1`, and `velocitySequence_2` blocks always define the initial 3-component velocities of individual particles in the particle velocity space.

The `velocitySequence_3` block defines the next particle component, which will vary from one species kind to another. See *Velocities and Internal Variables of Particles* for a table of internal variables by species kind. For example, it may set initial values for particle weights. For fixed-weight particles these `NAFunc` blocks can be set to a constant value using `kind=constNAFunc`. For variable-weight particles more general distributions can be specified, as outlined in the *NAFunc Block* documentation.

randDensSrc Parameters

`density` (*double*)
Positive value describing the density of the particles. This value is typically equivalent to the specified nominal density, however sometimes a scaling factor is used to load the particles more slowly, and prevent high frequency oscillations.

applyPeriod (*integer*)
Positive value n, directing the simulation to apply the source at every nth time step.

applyTimes (*double vector*)
Bracketed times for when the source/emitter will generate particles.

lowerBounds (*double vector*)
Lower bound (expressed in physical units, not grid units) of the physical extent of the source.

upperBounds (*double vector*)
Upper bound (expressed in physical units, not grid units) of the physical extent of the source.

vbar (*double vector*)
Average velocity of the particles in the x, y, and z directions.

vsig (*double vector*)
Positive value denoting the thermal velocity in the x, y, and z directions.

Example Particle Source of Kind randDensSrc

```
<ParticleSource InitElecDeposition>
  kind = randDensSrc
   lowerBounds = [$XSTART+DX$ YSTART]
   upperBounds = [$XEND-DX$ YEND]
   density = $NOMDEN/100.$ # used to prevent high frequency oscillations when loading␣
↪particles
   applyTimes = [0. LOADTIME]

  # Unit probability
  <STFunc macroDensFunc>
    kind = constantFunc
    amplitude = 1.
  </STFunc>
</ParticleSource>
```

Example Use of velocitySequence in a Particle Source

```
# Species kind = relBorisVWTagged, which has components x,y,z,ux,uy,uz,tag,weight
<ParticleSource gridSrc>

  kind = gridDenSrc
  density = PTCL_DENSITY
  numPerDir = [1 1 1]
  lowerBounds = [XSTART YSTART ZSTART]
  upperBounds = [XEND    YEND   ZEND]

  # Particle distribution uniform over initial phase space
  <NAFunc velocitySequence_0> # ux
    kind = bitRevNAFunc
  </NAFunc>
  <NAFunc velocitySequence_1> # uy
    kind = randGamma
    mean = MEAN_VEL
    sigma = SIGMA_VEL
  </NAFunc>
```

(continues on next page)

(continued from previous page)

```
  <NAFunc velocitySequence_2> # uz
    kind = randKappa
    sigmas = [$0.8*SIGMA_VEL$ SIGMA_VEL $1.2*SIGMA_VEL$]
  </NAFunc>
  <NAFunc velocitySequence_3> # tag
    kind = randGaussLimit
    mean = MEAN_VEL
    sigma = SIGMA_VEL
    lowerLimit = $2.*MEAN_VEL/3.$
    upperLimit = $4.*MEAN_VEL/3.$
  </NAFunc>
  <NAFunc velocitySequence_4> # weight
    kind = sysRandom
  </NAFunc>

</ParticleSource>
```

bitRevDensSrc (deprecated in 8.0)

Note: This source has been deprecated. Please use the xvLoaderEmitter kind with the bitRevSlabPosGen PositionGenerator instead.

This particle source loads particles in a random distribution based on a bit-reversed algorithm. The number of particles specified may not be what is obtained, as the random number generation process will only get close to the right number on each processor.

This particle source works with all VSim licenses.

bitRevDensSrc Parameters

density (*double*)
: Positive value describing the density of the particles.

lowerBounds (*double vector*)
: Lower bound (expressed in physical units, not grid units) of the physical extent of the source.

upperBounds (*double vector*)
: Upper bound (expressed in physical units, not grid units) of the physical extent of the source.

velocitySequence_0 (*NAFunc*)
: A sequence of initial velocities for the particles must be specified, in up to three orthogonal directions in the velocity space. Individual velocity sequences can be specified using an `NAFunc`; different methods for their specification are described in the *NAFunc Block* documentation.

velocitySequence_1 (*NAFunc*)
: A sequence of initial velocities for the particles must be specified, in up to three orthogonal directions in the velocity space. Individual velocity sequences can be specified using an `NAFunc`; different methods for their specification are described in the *NAFunc Block* documentation.

velocitySequence_2 (*NAFunc*)
: A sequence of initial velocities for the particles must be specified, in up to three orthogonal directions in the velocity space. Individual velocity sequences can be specified using an `NAFunc`; different methods for their specification are described in the *NAFunc Block* documentation.

velocitySequence_3 (*NAFunc*)

If variable-weight particles are used, the initial distribution of particle weights can also be specified using an NAFunc. See *NAFunc Block* for details.

Example bitRevDensSrc Block

```
<ParticleSource rampSrc>
   kind = bitRevDensSrc
   density = DENSITY
   lowerBounds = [X_LEFT_WALL 0. 0.]
   upperBounds = [X_RIGHT_WALL LY LZ]

# Particle distribution uniform over initial phase space
<NAFunc velocitySequence_0>
   kind = randGauss
   mean = 0.
   sigma = VEL_500K
 </NAFunc>
 <NAFunc velocitySequence_1>
   kind = randGauss
   mean = 0.
   sigma = VEL_500K
 </NAFunc>
 <NAFunc velocitySequence_2>
   kind = randGauss
   mean = 0.
   sigma = VEL_500K
 </NAFunc>
 <NAFunc velocitySequence_3>
   kind = randExp
   mean = 1.
 </NAFunc>

 </ParticleSource>
```

bitRevDensSrcVW (deprecated in 8.0)

Note: This source has been deprecated. Please use the xvLoaderEmitter kind with the bitRevSlabPosGen Position-Generator instead.

This particle source loads particles in a random distribution based on a bit-reversed algorithm. The number of particles specified may not be what is obtained,as the random number generation process will only get close to the right number on each processor.

This particle source works with all VSim licenses.

bitRevDensSrcVW Parameters

density (*double*)

Positive value describing the density of the particles.

lowerBounds (*double vector*)
Lower bound (expressed in physical units, not grid units) of the physical extent of the source.

upperBounds (*double vector*)
Upper bound (expressed in physical units, not grid units) of the physical extent of the source.

doShiftLoad (*integer, default = :samp:'0' (off)*)
For moving windows, move the particles with the windows.

vbar (*double vector*)
Average velocity of the particles in the x, y, and z directions.

vsig (*double vector*)
Positive value denoting the thermal velocity in the x, y, and z directions.

macroDensFunc (*STFunc block*)
STFunc block with the explicit name "macroDensFunc" of kind = expression is used to describe the probability of loading a macroparticle as a function of space and time.

weightFunc (*STFunc block*)
STFunc block with the explicit name "weightFunc" of kind = expression is used to describe the particle weighting as a function of space and time.

Example weightFunc Block

```
<STFunc weightFunc>
  kind = expression
  expression = 10.
</STFunc>
```

Example bitRevDensSrcVW Block

```
 <ParticleSource channelSrc>
  kind = bitRevDensSrcVW
  density = DENSITY2  # Because we have only one loader
  lowerBounds = [0.0 YBEG_LOAD ZBEG_LOAD]
  upperBounds = [1.  YEND_LOAD ZEND_LOAD]
  doShiftLoad = 1
  vbar = [0. 0. 0.]
  vsig = [0. 0. 0.]

  # Product of ramp up and down plus channel
  <STFunc macroDensFunc>
    kind = multFunc

    <STFunc cosRamp>
      kind = cosineRamp
      direction = [1. 0. 0.]
      startPosition = STARTRAMP
      endPosition = STARTFLAT
      startAmplitude = 0.
      endAmplitude = 1.
    </STFunc>

    <STFunc channel>
```

(continues on next page)

(continued from previous page)

```
      kind = radialCosChannel
      direction = [1. 0. 0.]
      channelPosition = [0. 0. 0.]
      startRadius = 0.
      endRadius = 4.e-5
      startAmplitude = 0.5
      endAmplitude = 1.
    </STFunc>

  </STFunc>

</ParticleSource>
```

gridDenSrcVW (deprecated in 8.0)

Note: This source has been deprecated. Please use the xvLoaderEmitter kind with the gridPosGen PositionGenerator instead.

This particle source loads particles along the nodes of the grid. Unlike gridDenSrc this also includes a function to set the particle weight for variable weight particles.

This particle source is available with all VSim licenses.

gridDenSrcVW Parameters

density (*double*)
: Positive value describing the density of the particles.

applyPeriod (*integer*)
: Positive value n, directing the simulation to apply the source at every nth time step. This option works only with emitter algorithms.

applyTimes (*double vector*)
: Bracketed times for when the source/emitter will generate particles.

lowerBounds (*double vector*)
: Lower bound (expressed in physical units, not grid units) of the physical extent of the source.

upperBounds (*double vector*)
: Upper bound (expressed in physical units, not grid units) of the physical extent of the source.

doShiftLoad (*integer, default = 0 (off)*)
: For moving windows, move the particles with the windows.

vbar (*double vector*)
: Average velocity of the particles in the x, y, and z directions.

vsig (*double vector*)
: Positive value denoting the thermal velocity in the x, y, and z directions.

numPerDir (*integer vector*)
: Number of macro particles to load in each direction.

weightFunc (*STFunc block*)
: A STFunc describing the particle weighting.

Example Particle Source of kind gridDenSrcVW

```
<ParticleSource constSrc>
  kind = gridDenSrcVW
  density = NOMDENS
  doShiftLoad = 1

  lowerBounds = [$.2*LX$    Y0 Z0]
  upperBounds = [$10*LX$   YEND ZEND]

  numPerDir=[2 2 2]

  vbar = [0. 0. 0.]
  vsig = [0. 0. 0.]

# weight function
<STFunc weightFunc>
  kind = multFunc

  <STFunc cosFT>
    kind = cosineFlattop
    direction = [1. 0. 0.]
    startPosition = $0.2*LX$
    startFlattop  = $0.2*LX + 0.2*LX$
    endFlattop    = $10.*LX - 0.2*LX$
    endPosition   = $10.*LX $
    startAmplitude = 0.0
    flattopAmplitude = 1.
    endAmplitude   = 0.0
  </STFunc>
  <STFunc channel>
      kind = leakyChannel
      direction = [1. 0. 0.]
      channelPosition = [0. 0. 0.]
      maxParabRadius = MAXPARABRADIUS
      maxRadius = MAXRADIUS
      centerAmplitude = DENSRAT
      quadCoef = QUADCOEF
  </STFunc>

</STFunc>

</ParticleSource>
```

Specialized Sources:

fileDensSrc

Loads particle phase space coordinates contained in a text file specified by “file” in the example block below. Regardless of the dimension of the simulation, the imported file should have at least six columns, separated by spaces (no commas). For constant weight particles, the six columns specify x, y, z, P_x, P_y, P_z , where $P_i = \gamma v_i$. For variable weight particles, a 7th column needs to be included to specify the weight of the particle.

In the file, each row corresponds to a single particle.

By default, the distribution of particles in the file will be loaded at the first time step. To repeat loading, use the `applyPeriod` attribute.

fileDensSrc Parameters

`file` (*string*, *required*)
The text file containing the particle phase space data to be loaded.

`fileNameBase` (*string*, *optional*)
Base name for series of text files to be loaded over a series of time steps. If this parameter is used then *`file`* is not needed.

`fileExtension` (*string*, *optional*)
File extension for series of text files to be loaded. Required if *`fileNameBase`* is used.

`applyPeriod` (*integer*, *optional*)
The time step intervals to load the particles. So, if this value is 2, particles will be loaded every other timestep. If 3, particles will be loaded every third timestep, etc. If used with *`fileNameBase`* the files loaded will be named fileNameBase_N.fileExtension where N is an integer multiple of the *`applyPeriod`*.

`shiftPtclPosition` (*double vector*, *optional*)
Allows the user to shift the position of the particles. The units of `shift_i` are meters. For example, to shift each particle by the same random distance in x between 0 and 1 micron, one can define `shift_x = __import__("random").random()*1E-6`.

Example fileDensSrc Block

```
<ParticleSource electronSrc>
  kind = fileDensSrc
  file = textFilePtclSource.dat
  #applyPeriod = 3
  #shiftPtclPosition = [shift_x shift_y shift_z]
</ParticleSource>
```

manualSrc

This particle source creates particles and adds them to a species. Each of the particles positions and velocities must be specified manually in the particle source block.

This particle source is available with all VSim licenses.

ParticleSource kind used to set manual particle generation. By specifying parameters p1, p2, p3. . . , you can describe exactly what particles are emitted and with what characteristics. These parameters should be specified in order, though you can specify any number of particles.

Each particle entry is a vector containing the following data:

$x, y, z, \gamma v_x, \gamma v_y, \gamma v_z$

Vorpal uses SI units, so positions have units of meters and velocities have units of m/s. For non-relativistic particles, $\gamma = 1$, so the vector specifies position and velocity.

To control when particles are to be emitted, use *`applyPeriod`* in combination with the `manualSrc` emitter. To emit particles only at the beginning of a simulation set `applyPeriod = 0`. To generate particles every *n* time steps, set `applyPeriod = n`.

manualSrc Parameters

applyPeriod (*integer*)
: Positive value n, directing the simulation to apply the source at every nth time step.

applyTimes (*double vector*, *optional*)
: Bracketed times for when the emitter will generate particles.

file (*string*, *optional*)
: Particle data may be attached in a separate text file, as with *fileDensSrc*.

Example manualSrc Blocks

```
<ParticleSource  myManualSource>
    kind = manualSrc
    applyPeriod = 0
    p1 = [ 1.e-3  -2.5e-4 0. 7.e+10 0.0 0.0 ]
    p2 = [-1.e-3   2.5e-4 0. 7.e+10 0.0 0.0 ]
    p3 = [ 1.e-3   2.5e-4 0. 7.e+10 0.0 0.0 ]
</ParticleSource>
```

```
<ParticleSource electronInitDist>
    kind = manualSrc
    file = electronDist.dat
    applyPeriod = 0
</ParticleSource>
```

gaussSource

This particle source loads particles distributed according to a Gaussian probability in all variables, including spatial. The number of particles specified may not be what is obtained, as the random number generation process will only get close to the right number on each processor.

This particle source is available with all VSim licenses.

gaussSource Parameters

numPtcls (*integer*)
: The number of particles in the Gaussian beam.

xbar (*double vector*)
: Average x, y, and z positions of the particles.

xsig (*double vector*)
: Sigma spread in the x, y, and z directions.

vbar (*double vector*)
: Average velocity of the particles in the x, y, and z directions.

vsig (*double vector*)
: Positive value denoting the thermal velocity in the x, y, and z directions.

GaussSource example block

```
<ParticleSource myGaussSource>
  kind = gaussSource
  numPtcls = 100
  xbar = [0.0 0.0 0.0]
  xsig = [LX_SIG LY_SIG LZ_SIG]
  vbar = [VBAR 0.0 0.0 1.0]
  vsig = [VTH VTH VTH 0.0]
</ParticleSource>
```

boostLoader

A loader specifically designed to load particles in a boosted frame simulation from a laboratory frame distribution. The boostLoader reproduces some of the functionality of xvLoaderEmitter. It creates particles by *density-loading* in a volume, but is not capable of *flux-emitting* from a surface. In a boosted frame simulation the reference frame of the simulation is moving with a Lorentz factor gamma_boost relative to the laboratory (rest) frame. This loader allows the user to specify the properties of an electron beam in the laboratory frame, which are then internally Lorentz transformed in the moving frame of the simulation. When gamma_{boost} is set to 1, the boostLoader loads beam particles identically to xvLoaderEmitter.

This particle source is available with any VSimPA license.

boostLoader Sub-Blocks

PositionGenerator
PositionGenerator (block) Is used to generate physical positions of particles when they are loaded. The loadSlab must be specified in the moving frame of the simulation (The loadSlab specify the region where the particles are loaded in the frame of the simulation).

VelocityGenerator
VelocityGenerator (block) Is used to generate velocities of a particle when loaded. All velocities, including particle weight as a function of (x,y,z,t) must be specified in the laboratory frame. Particle positions and velocities are transformed internally in the moving frame of the simulation.

boostLoader Parameters

applyTimes
Interval over which particles are created.

gammaBoost
Lorentz factor of the moving frame of the simulation.

loadOnShift (*string*, *default = false*)
A string, `true` or `false`, used in shifting window simulations to insure that particles are loaded into the new cells added when the simulation shifts. `loadOnShift` should be set to true if a moving window is used in the simulation. If true, then must also have `load = true`.

loadAfterInit (*string*, *default = false*)
Used to continue loading particles after the initialization, for example a continuous bulk source ionization. `loadAfterInit` should be set to true only if more particles should be loaded into the `loadSlab` with each time step. If true, then must also have `load = true`.

gridLoader

This particle source loads particles along the nodes of the simulation grid. It is a general loader that can work with a moving window.

This particle source is available for all VSim licenses.

gridLoader Sub-Blocks

Slab
Slab (block) Is required to be named *initLoadSlab* and is the slab into which particles are loaded at `t = 0`. At later times, the load slab is the initial load slab moved by the sweep rate times the time.

VelocityGenerator
VelocityGenerator (block) Is used to generate velocities of a particle when loaded. All velocities, including particle weight as a function of (x,y,z,t) must be specified in the laboratory frame. Particle positions and velocities are transformed internally in the moving frame of the simulation.

gridLoader Parameters

applyTimes (*double vector*)
Times for when the source will generate particles.

macroPerDir (*double vector*)
The number of macro-particles in each direction. The product of these is the number of particles loaded per cell.

sweepVel (*double vector*)
Velocities at which the loading region is moving.

Example Particle Source Block of Kind gridLoader

```
<ParticleSource channelSrc>
 kind = gridLoader
 applytimes = [0. 1.]
 <Slab initLoadSlab>
   lowerBounds = [STARTRAMP_BOOST YSTART_LOAD  ZSTART_LOAD]
   upperBounds = [ENDPLASMA_BOOST YEND_LOAD    ZEND_LOAD]
 </Slab>
 sweepVel = [$ -VX_BOOST $  0.  0.]
 macroPerDir = [NUM_PER_DX  NUM_PER_DY NUM_PER_DZ]

 # All velocity components (0,1,2) default to zero, which is what we want.
 # Here, the particle weights (component 3) are set to create a short
 # density ramp.
 <VelocityGenerator rampVelGen>
   kind = funcVelGen
   # specify relativistic drift along x
   <STFunc component0>
     kind = constantFunc
     amplitude = -UX_BOOST
   </STFunc>
   # specify weight of the variably-weighted particles
   <STFunc component3>
```

(continues on next page)

(continued from previous page)

```
      kind = multFunc
      <STFunc cosFT>
        kind = expression
        expression = 0.5 * H(pp(t, x) - STARTRAMP_BOOST) \
                    * H(STARTFLAT_BOOST - pp(t, x)) \
                    * (1 + cos(PI * (STARTFLAT_BOOST - pp(t, x)) \
                    / RAMPLEN_BOOST)) + H(pp(t, x) - STARTFLAT_BOOST) \
                    * H(ENDFLAT_BOOST - pp(t, x)) \
                    + 0.5 * H(pp(t, x) - ENDFLAT_BOOST) \
                    * H(ENDPLASMA_BOOST - pp(t, x)) \
                    * (1 + cos(PI * (pp(t, x) - ENDFLAT_BOOST) \
                    / RAMPLEN_BOOST))
      </STFunc>
      <STFunc channel>
        kind = leakyChannel
        direction = [1. 0. 0.]
        channelPosition = [0. 0. 0.]
        maxParabRadius = MAXPARABRADIUS
        maxRadius = MAXRADIUS
        centerAmplitude = DENSRAT
        quadCoef = QUADCOEF
      </STFunc>
    </STFunc>
  </VelocityGenerator>
</ParticleSource>
```

fluxPtclEmitter

This particle source reads in and emits particles for which one coordinate is given as a time of emission. It was previously used in conjunction with an absSaveDump particle absorber.

This particle source kind can be used with any VSim license.

fluxPtclEmitter Parameters

fluxSpeciesDumpName (*string*, *required*)
Sets the name of the HDF5 dataset which the flux emitter class will read in. Also sets part of the read-file name. Usually, this name will be the species name followed by the string Flux.

useRunNameAsPrefix (*integer*, *default = 1 (true)*)
Denotes whether the standard Vorpal dump name, or run name, is expected to be prepended to the flux particle file names. Useful if the flux particle files were created by a simulation with a different output name.

emitDim (*integer*, *default = 0*)
The direction of emission. One of 0, 1, 2

emitPlaneLocation (*float*, *default = 0.*)
The physical location of the emission plane.

initialDumpIndex (*integer*)
The index of the flux particle file to start reading from.

fluxFileReadPeriod (*integer*, *default = 0*)
The number of time steps between reads.

emitDistanceShift (*float, default = 0.*)
: Shift all the particles by this distance before loading.

relativeValueFlag (*boolean, default = 1*)
: Whether relative values are used in the flux particle file or not.

removeFluxFilesAfterReadFlag (*boolean, default = 0*)
: Set whether to remove the data files after reading them.

delaySteps (*integer, default = 0*)
: Allow for delaying the entry of particles by delaySteps steps.

density (*double*)
: Positive value describing the density of the particles.

applyPeriod (*integer*)
: Positive value n, directing the simulation to apply the source at every nth time step. This option works only with emitter algorithms.

applyTimes (*double vector*)
: Bracketed times for when the source/emitter will generate particles.

lowerBounds (*double vector*)
: Lower bound (expressed in physical units, not grid units) of the physical extent of the source.

upperBounds (*double vector*)
: Upper bound (expressed in physical units, not grid units) of the physical extent of the source.

doShiftLoad (*integer, default = 0 (off)*)
: For moving windows, move the particles with the windows.

vbar (*double vector*)
: Average velocity of the particles in the x, y, and z directions.

vsig (*double vector*)
: Positive value denoting the thermal velocity in the x, y, and z directions.

nomMacroPtclsPerStep (*double*)
: A reference (nominal) number of macroparticles emitted from the entire emitter each time step.

seed (*integer*)
: Specifies the seed for the random number generator. *seed* must be a positive integer.

Example fluxPtclEmitter Code Block

```
<ParticleSource leftYellowFluxEmitter>
 kind = fluxPtclEmitter
 fluxSpeciesDumpName = yellowElectronsFlux
 useRunNameAsPrefix = 0

 #initialDumpIndex = 0
 #fluxFileReadPeriod = 10
 removeFluxFilesAfterReadFlag = 0

 ##emitDistanceShift =-$NX_PML * DX$
 #  dataSetName = redElectronsFluxData
 #relativeValueFlag = 0

 emitDim = 0
 emitPlaneLocation = -0.15
```

(continues on next page)

(continued from previous page)

```
 applyTimes = [0. 100000000000000.]

</ParticleSource>
```

Notes on fluxPtclEmitter Particle Source

Use of a Vorpal simulation with the output base name test2 would result in Vorpal looking for a dump file whose name is prefixed by that base name such as:

```
test2_Globals_1.h5
```

If the base name is set and you set the *fluxSpeciesDumpName* parameter to a text string such as:

```
electronsFlux
```

then, by default, Vorpal will look for a file whose name is prefixed by the concatenated base name and the string you have specified, such as:

```
test2_electronsFlux_0.h5
```

However, if you set:

```
useRunNameAsPrefix  = 0
```

then Vorpal will use only the *fluxSpeciesDumpName* parameter string as the filename prefix like this:

```
electronsFlux_0.h5
```

switchSpeciesSrc

This is a particle source that will emit already existing particles as a different particle species. Must be coupled with a particle absorber from the source species.

This kind of particle source is available with a VSimPA license.

switchSpeciesSrc Parameters

switchSpecies (*string*)
Species to which the particles will be switched.

ptclAbsorber (*string*)
Name of the particle absorber from the source species. Particles that impact this absorber will be switched to the species defined in switchSpecies.

minDim (*integer*)
Bracketed times for when the source/emitter will generate particles.

Example Particle Source of Kind switchSpeciesSrc

```
<ParticleSource beamElectronsSwitchEmitter>
  kind = switchSpeciesSrc
  switchSpecies = ElectronBeam
  ptclAbsorber = beamElectronsSwitchAbsorber
  minDim = 1
</ParticleSource>
```

gaussDensSrc (deprecated in 8.0)

Note: This source has been deprecated. Please use the xvLoaderEmitter kind with the gridPosGen PositionGenerator and the beamVelocityGen VelocityGenerator instead.

This particle source emits particles based on a Gaussian profile.

gaussDensSrc Parameters

density (*double*)
Positive value describing the density of the particles.

applyPeriod (*integer*)
Positive value n, directing the simulation to apply the source at every nth time step. This option works only with emitter algorithms.

applyTimes (*double vector*)
Bracketed times for when the source/emitter will generate particles.

lowerBounds (*double vector*)
Lower bound (expressed in physical units, not grid units) of the physical extent of the source.

upperBounds (*double vector*)
Upper bound (expressed in physical units, not grid units) of the physical extent of the source.

doShiftLoad (*integer, default = 0 (off)*)
For moving windows, move the particles with the windows.

vbar (*double vector*)
Average velocity of the particles in the x, y, and z directions.

vsig (*double vector*)
Positive value denoting the thermal velocity in the x, y, and z directions.

nomMacroPtclsPerStep (*double*)
A reference (nominal) number of macroparticles emitted from the entire emitter each time step.

seed (*integer*)
Specifies the seed for the random number generator. `seed` must be a positive integer.

Example gaussDensSrc Particle Source Block

```
<ParticleSource gaussSrc>
 kind = gaussDensSrc
 density = DENSITY
 lowerBounds = [0.0 0.0 0.0]
 upperBounds = [LX LY LZ]
 vbar = [0.0 0.0 0.0]
 vsig = [ELECTHERMSPEED ELECTHERMSPEED ELECTHERMSPEED]
 <STFunc macroDensFunc>
  kind = expression
  expression = 0.99999-5.5263e7*FIELD/DENSITY*(-PERTHW2I*(x-.5*LX)*exp(-2.*(x-.
↪5*LX)*(x-.5*LX)*PERTHW2I))
 </STFunc>
</ParticleSource>
```

planarPtclEmitter (deprecated in 8.0)

Note: This source has been deprecated. Please use the xvLoaderEmitter kind with an emitSurface Slab in the PositionGenerator instead.

This particle beam is emitted from flat surface. This kind adds an additional parameter, relativisticFlag. This integer flag tells the emitter whether the particles are of relativistic type (i.e. using a relativistic mover class such as relBoris for updating the particles) or not. The emitted particles' velocities are adjusted accordingly.

This particle source can be used with all VSim licenses.

planarPtclEmitter Parameters

density (*double*)
Positive value describing the density of the particles.

applyPeriod (*integer*)
Positive value n, directing the simulation to apply the source at every nth time step. This option works only with emitter algorithms.

applyTimes (*double vector*)
Bracketed times for when the source/emitter will generate particles.

lowerBounds (*double vector*)
Lower bound (expressed in physical units, not grid units) of the physical extent of the source.

upperBounds (*double vector*)
Upper bound (expressed in physical units, not grid units) of the physical extent of the source.

doShiftLoad (*integer, default = 0 (off)*)
For moving windows, move the particles with the windows.

vbar (*double vector*)
Average velocity of the particles in the x, y, and z directions.

vsig (*double vector*)
Positive value denoting the thermal velocity in the x, y, and z directions.

numMacroPtclsPerStep (*double*)
A reference (nominal) number of macroparticles emitted from the entire emitter each time step.

numMacroPtclsPerCellPerStep (*double*)
Number of macro particles to emit per cell per timestep.

current (*double*)
This switches from a current density profile to a current profile.

nonUniformMacroPtclsFlag (*double*)
If set allows for variation in macro particle density.

seed (*integer*)
Specifies the seed for the random number generator. *seed* must be a positive integer.

positionFunction (*string*, *optional*)
String to specify how particles are loaded. Available choices are random, bitReversed, lattice and equispaced.

usePositionFuncInEmitDir (*integer*, *optional*)
Flag for whether to use the normPositionFunc for time of emission.

weightMinimum (*float*, *optional*)
The lowest weight allowed if loadZeroWeights is false.

weightFunc (*STFunc Block*, *optional*)
Determines the relative density of the loaded particles by setting the probability that an individual is loaded.

macroFluxFunc (*STFunc Block*, *optional*)
The space time function that determines the macroparticle flux.

currentDensityFunc (*STFunc Block*, *optional*)
The space time function that determines the current.

fluxDensityFunc (*STFunc Block*, *optional*)
The space time function that determines the flux.

Example Particle Source of kind planarPtclEmitter

```
<ParticleSource leftBeamEmitter>
  kind = planarPtclEmitter

  ## specify the number of particles emitted from the entire emitter each time step
  nomMacroPtclsPerStep = 5

  ## current density function
<STFunc currentDensityFunc>
  kind = expression
  expression = t / DT / 10.
</STFunc>

  ## specify a macroparticle position choosing function: bitreversed, random, or␣
↪lattice
  ## (default is bitreversed)
  positionFunction = bitReversed

  # Bounds specified with physical coordinates, one dimension must have zero extent
  lowerBounds = [STARTX $Y_CENTER - 1.5 * E_RMS_WIDTH_Y$ $Z_CENTER - 1.5 * E_RMS_
↪WIDTH_Z$]
  upperBounds = [STARTX $Y_CENTER + 1.5 * E_RMS_WIDTH_Y$ $Z_CENTER + 1.5 * E_RMS_
↪WIDTH_Z$]

  applyTimes = [0. $1.*DT$]
```

(continues on next page)

(continued from previous page)

```
  ## particle velocities
<NAFunc velocitySequence_0>
  kind = randGauss
  mean =  $E_BEAM_GAMMA * E_BEAM_SPEED$
  sigma = $E_BEAM_GAMMA * 1.e6$
</NAFunc>
<NAFunc velocitySequence_1>
  kind = randGauss
  mean = 0.
  sigma =0.0
</NAFunc>
<NAFunc velocitySequence_2>
  kind = randGauss
  mean = 0.0
  sigma = 0.0
</NAFunc>

## some other optional parameters

## specify the total current (constant, may later implement as a function of time)
#current = 6.0e-2

## vary the macroparticle density (weights may still be variable if desired)
#nonUniformMacroPtclsFlag =1

## can also give the macroparticle flux as a function
<STFunc macroFluxFunc>
  kind = expression
  expression = 1.
</STFunc>

## can also give the weight as a function, otherwise it is 1
<STFunc weightFunc>
 kind = expression
 expression = 1.
</STFunc>

</ParticleSource> # end planarPtclEmitter
```

Secondary Sources:

secElec

This kind of ParticleSource algorithm allows Vorpal to model the generation of secondary electrons being produced by impact with electrons or ions on metal surfaces.

For electrons, the determination of number, energy spectrum, and angular distribution of these secondary electrons can be found in [furman2002probabilistic].

When the incident particles are ions, secondary electron emission and neutral desorption yields are calculated due to ion-target interactions. The secondary electron yield for most models depends on the projectile energy and angle of incidence, and the target material. Ion-induced secondary electron emission is proportional to electronic stopping, so stopping power dE/dx is calculated to compute secondary electron yields. Similarly, neutral desorption depends on nuclear stopping powers. The stopping power calculation is appropriate for both cold solids and dense plasmas, and is comprised of contributions from bound electrons, free electrons, and nuclear collisions. Further details are provided in *[SVC+06]*.

Note: When dealing with incident particles of electrons; To emit secondary electrons in a different species than the primary impacting electrons, include the secElec ParticleSource block in the secondary species and reference the ptclAbsorber from the primary species. For example: ptclAbsorber = primaryElectrons.topAbsorber

---

Note: When dealing with incident particles of ions, there are two ways to emit secondary electrons in a different species.

1) Include the secElec ParticleSource block in the secondary species and reference the ptclAbsorber from the primary species. For example: ptclAbsorber = primaryIons.topAbsorber

2) Include the secElec ParticleSource block in the primary ion species and use the parameter `secondarySpecies` to reference the secondary species. This parameter ONLY works for incident ions. For example: secondarySpecies = electrons

---

This particle source is available with a VSimMD or VSimPD license.

secElec Parameters

`ptclAbsorber` (*string*, *required*)
Name of the absorber that absorbs the electrons and from which the secondaries will be emitted.

`material` (*string*, *optional*, *default* = *copper*)
When using elections as the incident particle, material choices are one of:

- `copper`
- `stainless` (stainless steel)

When the incident particle is an ion, the choices are:

- `hydrogen`
- `helium`
- `carbon`
- `nitrogen`
- `oxygen`
- `sodium`
- `aluminum`
- `silicon`
- `phosphorus`
- `argon`
- `iron`
- `nickel`
- `copper`
- `silver`

- `gold`
- `uranium`
- `air`
- `water`
- `stainless` (stainless steel)

`direction` (*vector*, *optional*)
Direction vector should point along the outward-facing normal of the ParticleSource.

`emissionProb` (*real*, *optional*)
Specifies the probability for emission of secondaries from a material surface. For the constant probability model, a single electron is emitted with a given probability independent of all other factors, such as the incident energy, material and primary ion properties, etc.

`ptclCountType` (*string*, *optional*, *default = emitCounting*)
Describes how the emission of multiple secondary electrons is handled. This parameter is optional, and defaults to emitCounting. Valid types are:

- `emitCounting` (default)

  Emit multiple macroparticles.

- `noCounting`

  Emit single particle.

- `vwCounting`

  Emit single variable-weight particle and increase its weight by the number of secondaries.

- `taggedVwCounting`

  Emit single variable-weight, tagged particle and increase its weight by the number of secondaries.

`emittingSurface` (*float*, *optional*)
Physical position of the emitting surface in the direction of emission. If this parameter is not specified, it is calculated to be the appropriate edge of the grid. It is ignored when emitting from a grid boundary.

`secondarySpecies` (*string*, *optional*)
This parameter is only valid for primary ions and will be ignored if primaries are electrons. secondarySpecies sets the electron species to emit the secondaries into.

`suppressEnergy`
By default, emission does not occur if the electric field has sign that would immediately force the emitted particle back into the surface (default value is `suppressEnergy=0`). This parameter can be used to control this feature. If an emitted particle is desired, regardless of the sign of the electric field, then set this parameter to a very large number, e.g., `suppressEnergy=1.0e32`.

More specifically, emission occurs when the particle charge, times the local field strength, times a characteristic length based on the grid, e.g., q*E*dx, is less than the `suppressEnergy`. The local field strength is interpolated on the emission surface. In higher dimensions, the characteristic length is the diagonal across the cell. The units of the `suppressEnergy` are electronVolts.

`ignoreProb` (*float*, *default = 0*)
Float value representing probability to ignore an absorbed electron so it can be used for another process. This should be a value between `0` and `1`.

`gridBoundary` (*string*)
If the particle absorber is on a gridBoundary, the name of the gridBoundary used by that particle absorber must also be specified in the ParticleSource block.

Example secElec Block

```
<ParticleSource leftSecondaryEmitter>
    kind = secElec
    minDim = 1
    ptclAbsorber = primaries.leftAbsorber
    direction = [-1. 0. 0.]
    material = copper
</ParticleSource>
```

secFieldEmitter

Secondary particle emitter where the SEY (secondary yield) and emitted particle spectrum are defined by pair of Expressions (see *Introduction to UserFuncs and Expressions*). These Expressions are functions of primary velocity components (vperp, vpara), a scalar field (phi) and time (t). The secondary particles can be in the same species as the incoming electrons, or a separate species.

secFieldEmitter Parameters

`ptclAbsorber` (*string*)
Name of the absorber that absorbs the primaries and from which the secondaries will be emitted.

`gridBoundary` (*string*, *optional*)
Name of a gridBoundary absorber that absorbs the primaries and from which the secondaries will be emitted.

`Expression` (*block*, *required*)
An expression (see *Introduction to UserFuncs and Expressions*) that defines SEY. It must be named `SEYFunc`. It can take four arguments `vperp` (primary velocity component along the surface normal, in m/s), `vpara` (primary velocity component parallel to the surface, in m/s), `phi` (the value of the scalar field at the location where the particle is absorbed) and `t` (time in second). Not all the arguments need to be used, but all arguments must come from these four.

`Expression` (*block*, *required*)
An expression (see *Introduction to UserFuncs and Expressions*) that defines secondary velocity (beta) distribution. It must be named `SpectrumFunc` and return a 3 component vector. It can take four arguments `vperp` (primary velocity component along the surface normal, in m/s), `vpara` (primary velocity component parallel to the surface, in m/s), `phi` (the value of the scalar field at the location where the particle is absorbed) and `t` (time in second). Not all the arguments need to be used, but all arguments must come from these four. The 3 component velocity vector returned is defined locally to the emitting surface. The first component is vnorm, which is along the surface normal (nvec) direction. The second component is along the direction of tvec = v x nvec, where v is the velocity vector of the incident primary particle. The third component is along the direction of nvec x tvec.

`secondarySpecies` (*string*, *optional*)
Species name of the secondaries emitted. If this is not specified, the emitted particles are of the same species as the incident particle.

`emittingSurface` (*float*, *optional*)
Physical position of the emitting surface in the direction of emission. If this parameter is not specified, it is calculated to be the appropriate edge of the grid. It is ignored when emitting from a grid boundary

simpleSec

This particle source is a simplified secondary electron emitter. When the ptclAbsorber is a gridBoundary, particles are re-emitted for the surface in the direction such that their velocity tangential to the surface is unchanged, but the normal component is reversed, however their energy is then adjusted as described below. If the ptclAbsorber is a slab, not a gridBoundary, then the direction is selected by the setting of the `direction` attribute to be the into the surface, out of the simulation region in which the particles exist, and the emission will be normal to the slab surface.

`simpleSec` can be used different ways for fixed or variable weight particles:

First, if using constant weight particles, it emits a single secondary electron of either fixed energy given by the user provided `emittedEnergy` parameter if `randomEnergy` is 0 or uniformly distributed in the range 0 to `emittedEnergy` if `randomEnergy=1`. `randomEnergy` defaults to 0. This emission may be suppressed by `suppressEnergy`, which may be set to an arbitrarily high number if emission is always required regardless of local field.

Second, if using variable weight particles, it modifies the weight of the particle based on the user provided `<STFunc sey>` curve.

Note: To emit particles in a different species than the primary impacting species, include the simpleSec ParticleSource block in the secondary species and reference the ptclAborber from the primary species. For example: ptclAbsorber = electrons.topAbsorber

Note: If trying to use the simpleSec source with ions, the simpleSec ParticleSource assumes non-relativistic **electrons** for the calculation of the incoming energy used in the SEY function. When using fixed-weight particles, or a constant SEY, there is no net effect. However, if using variable-weight particles, one must realize that the energy is based on non-relativistic electrons.

This kind of particle source is available with a VSimMD or VSimPD license.

simpleSec Parameters

`ptclAbsorber` **(string, required):** Name of the absorber that absorbs the electrons and from which the secondaries will be emitted.

`emittedEnergy` **(float, required):** The energy of the emitted electrons or the top of the range of energies if **`randomEnergy`** is set to `1`.

`randomEnergy` **(boolean, optional):** If **`randomEnergy`** is set to `1`, the ParticleSource will emit the secondary electrons with a random energy between `0` and the energy given in the input file from the `emittedEnergy` parameter.

`suppressEnergy` **(float, optional):** By default, emission does not occur if the electric field has sign that would immediately force the emitted particle back into the surface (default value is **`suppressEnergy=0`**). This parameter can be used to control this feature. If an emitted particle is desired, regardless of the sign of the electric field, then set this parameter to a very large number, e.g., **`suppressEnergy=1.0e32`**.

More specifically, emission occurs when the particle charge, times the local field strength, times a characteristic length based on the grid, e.g., q*E*dx, is less than the **`suppressEnergy`**. The local field strength is interpolated on the emission surface. In higher dimensions, the characteristic length is the diagonal across the cell. The units of the **`suppressEnergy`** are electronVolts.

`emittingSurface` **(string, optional):** The physical position of the emitting surface in the direction of emission

`direction` (float vector, optional): A vector denoting the normal pointing out of the simulation domain (eg into the metal or domain edge), i.e., away from the direction of emission from the `emittingSurface`. `direction` is ignored if the ptclAbsorber is a gridBoundary.

`gridBoundary` (string, optional): Only necessary if the ptclAbsorber is on a gridBoundary. The name of the gridBoundary used in the ptclAbsorber must also be specified in the ParticleSource block.

`STFunc sey` (block, optional): A *STFunc Block* that defines the SEY curve. This block must be named `sey`. This function should be a function of only x, where x gets interpreted as the impact energy of the impacting electron. This only works for variable weight particles.

`ptclCountType` (string, optional, default = `noCounting`): Describes how the emission of multiple secondary electrons is handled. This parameter is optional, and defaults to noCounting. Valid types are:

- **`noCounting` (default):** Emit single particle.
- **`vwCounting`:** Emit single variable-weight particle and increase its weight by the number of secondaries.
- **`taggedVwCounting`:** Emit single variable-weight, tagged particle and increase its weight by the number of secondaries.

Example simpleSec Block to emit secondary electrons in the same Species as the primary electrons

```
<ParticleSource simpleSecondaryEmitter>
  kind = simpleSec
  ptclAbsorber = cutcellBndry
  gridBoundary = plane
  emittedEnergy = EMIT_ENERGY
  ptclCountType = vwCounting
  <STFunc sey>
    kind = expression
    expression = 2.*H(x-LOWER_ENERGY)*H(UPPER_ENERGY-x)
  </STFunc>
</ParticleSource>
```

Example simpleSec Block to emit secondary electrons in a different Species as the primary electrons

```
<Species electrons>
.
.
.
.
<ParticleSink topAbsorber>
  kind = absAndSav
  minDim = 2
  lowerBounds = [0  NY  -1]
  upperBounds = [NX NY1 NZ1]
</ParticleSink>
.
.
.
</Species>
```

(continues on next page)

(continued from previous page)

```
<Species secondaryElectrons>
.
.
.
.
<ParticleSource simpleSecondaryEmitter>
  kind = simpleSec
  ptclAbsorber = electrons.topAbsorber
  emittedEnergy = EMIT_ENERGY
  direction= [0. -1. 0.]
  emittingSurface = $YSTART + LY-0.5*DY$
</ParticleSource>
.
.
.
</Species>
```

secondaryEmitter

This general secondary emitter emits (secondary) particles based on the absorption or emission of other (primary) particles. Typically, a secondary particle is emitted when a primary particle is absorbed by a ParticleSink; however, to allow correlated emission of different secondary species, the primary particle may be an emitted (rather than absorbed) particle. For example, suppose a primary ion incident upon an insulating surface has a certain probability of secondary-emitting an electron, and, for charge conservation, it is important to secondary-emit an ion at the same time (the secondary ion may remain immobile on the surface). A secondaryEmitter can be used to emit a secondary electron with appropriate probability when the primary ion hits the surface; another secondaryEmitter can then be used to emit a (possibly immobile) secondary ion whenever a secondary electron is produced, ensuring charge conservation (while it is possible to emit a secondary ion based on the same primary ion, if the emission probability is not 1, then there may be times when a secondary electron is emitted but not a secondary ion, and vice-versa). This secondaryEmitter allows great flexibility in determining the parameters of the secondary particle; hence it also requires a lengthy specification of the secondary particles' properties.

While most other secondary emitters emit secondary particles at a randomly-chosen time within the given timestep, this emitter emits particles at the recorded absorption time.

This particle source is available with a VSimMD or VSimPD license.

secondaryEmitter Parameters

`ptclAbsorber` (*string*)
To emit secondaries from primary particles that are absorbed (or, more generally, that run into a ParticleSink), this option should specify the name of the ParticleSink (which must be of a kind that stores absorbed particles). When used with a `gridBoundary` the `ptclAbsorber` should probably always be of type *absSavTriCutCell* (with `recordParticleData=true`), since that is currently the only cut-cell absorber that records the exact time of absorption and the surface normal.

`ptclSource` (*string*)
(This option may be specified instead of *`ptclAbsorber`*.) To emit secondaries from primary particles that are emitted from a ParticleSource, this option should specify the name of the particle source that emits the primary particles whose emission triggers secondary emission. If the ParticleSource emits a species other than the Species block that contains the ParticleSource, then *`sourceSpecies`* must also be specified. This is typically used to allow secondary emission of a Species based on secondary emission of another Species. The

ParticleSource block must specify `recordParticleData=true` because secondaries will be emitted based on the recorded particles.

`sourceSpecies` (*string*)
The name of the Species emitted by the ptclSource; this is required only if a *`ptclSource`* is specified that emits a Species other than the Species block containing the `ParticleSource`.

`lowerBounds` (*integer vector*, *optional*, *default = global simulation bounds*)
The lower (global cell) bounds within which a primary particle can produce secondaries.

`upperBounds` (*integer vector*, *optional*, *default = global simulation bounds*)
The upper (global cell) bounds within which a primary particle can produce secondaries.

`gridBoundary` (*string*, *optional*)
A gridBoundary specifying the surface from which secondaries are emitted; it is strongly recommended that this be the same gridBoundary at which primary particles are absorbed. The gridBoundary supplies the surface-normal where secondary emission occurs. If, in pathological cases, there is no surface in the cell where emission occurs, the normal will be taken to be opposite the primary particle's velocity.

`emissionDirection` (*vector*, *optional*)
if no `gridBoundary` is specified, this gives the outward surface normal direction for secondary emission.

`depositCurrentFromCorner` (*boolean*, *optional*, *default = true*)
This option should generally be `true` in electromagnetic simulations, especially for emission from metallic (conducting) surfaces; it avoids the creation of artificial stationary charges. In electromagnetic simulation, emitting a charge from the middle of a cell automatically creates (or rather, leads to electromagnetic fields that act as if there were) an opposite charge that remains at the emission location forever. If this option is `true`, then electrical current will be deposited from a corner of the emission cell that is inside the absorber to the emission location.

`functionVariables` (*vector of strings*, *optional*, *default = empty vector*)
a list of quantities, describing the primary particle, upon which the Expressions `sey`, `velocityAndTag`, and `internVars` can depend. For example, including `velocity_0` in this list allows the first component of the primary particle's velocity to be used in the `sey` function. Valid options are the same as for `ptclAttributes` in *speciesBinning*.

`sey` (*block*, *required*)
The expression `<Expression sey>` (see *Introduction to UserFuncs and Expressions*) that defines the secondary emission yield (SEY). It must be named `sey`, and can take arguments listed in *`functionVariables`* which depend on the particular primary particle. It must return a scalar value which is the (expected or average) number of physical secondary particles to be emitted for each physical primary particle. The interpretation of what to do if the SEY calls for a fractional secondary macroparticle, or multiple secondary macroparticles, depends on the *`ptclCountType`* option.

`emitIntoGridBoundaryInterior` (*boolean*, *optional*, *default = true*)
Typically the *`gridBoundary`* interior is the physical domain in which particles move, while the exterior absorbs particles (and this option should be `true`); in case the *`gridBoundary`* interior is the absorbing material, with particles moving in the region exterior to the *`gridBoundary`*, this option can be set to `false` so that secondary particles will be emitted into the exterior of *`gridBoundary`*. If `false`, the *`ptclAbsorber`* should probably specify `absorberIsInGridBoundary=true` (currently, only ParticleSink *absSavTriCutCell* has that option).

`ptclCountType` (*string*, *optional*, *default = emitCounting*)
Describes how the emission of fractional or multiple secondary particles is handled. This parameter is optional, and defaults to emitCounting. In the following, a primary macroparticle represents P physical particles and the expected number of secondary physical particles is $S = P \times SEY$. Valid options are:

- `emitCounting` (default for constant-weight secondaries)

Emit multiple secondary macroparticles with default weight. I.e., if M is the default number of physical particles per secondary macroparticle, then either floor(S/M) or ceil(S/M) secondary macroparticles will be emitted, with a probability chosen so that on average S physical secondaries are emitted.

- `noCounting`

  This is the same as `emitCounting` except that at most 1 macroparticle is emitted (i.e., either 0 or 1 secondary macroparticles). This gives the wrong result if S exceeds the default number of physical particles per secondary macroparticle. However, it may be appropriate if that situation is rare and one wants to avoid emitting a glut of secondary macroparticles on those rare occasions.

- `vwCounting` (default for variable-weight secondaries)

  For variable-weight secondary species, emit a single macroparticle with weight set to represent S secondary physical particles.

`autoGenerateTag` (*boolean*, *optional*, *default = true*)
Whether a tag will be automatically generated for each secondary; if `false` then the `velocityAndTag` Expression must specify the tag.

`<Expression velocityAndTag> (block, required)`
An expression (see *Introduction to UserFuncs and Expressions*) that defines the velocity and (possibly) tag of the secondary particle. It must be named `velocityAndTag`, and can take arguments listed in *`functionVariables`* which depend on the particular primary particle. If *`autoGenerateTag`* is `true` then this must return 3 velocity components; if `false` then this must return 4 components, three velocities plus a tag for the secondary. The tag is used only if the secondary Species is tagged. The "velocity" components are actually the spatial four-velocity components in a coordinate system that depends on the emission surface and the primary particle. The first coordinate u_n is the component normal to the surface; the second is perpendicular to the first and in the plane of the primary particle's velocity. The third coordinate is then normal to the first two, and in right-hand-rule order. If the primary particle is incident along the normal, then the third coordinate is the cardinal direction (e.g., +x, +y, or +z in cartesian systems) most perpendicular to the normal; and the second is then chosen normal to the first and third in right-hand order. In 2D, the third coordinate is always in the direction of the unsimulated direction, with its sign chosen so that the primary incident velocity and the cross product of the third and first coordinates are in the same drection.

`<Expression internVars> (block, optional)`
An optional expression (see *Introduction to UserFuncs and Expressions*) that defines the final internal variables (e.g., velocity, tag, weight, etc.) of each secondary particle (this is an advanced feature that is rarely needed). It must be named `internVars`, and can take arguments listed in *`functionVariables`* as well as the recommended internal variables selected for the secondary particle, e.g., by `velocityAndTag` and *`ptclCountType`*. These are accessed by the variables `secInternVars_0`, `secInternVars_1`, etc. This expression allows ultimate flexibility in setting the final internal variables of the secondary particle, but one must be very careful that one understands exactly what those internal variables are.

`emitForPrimariesWithoutSurfaceNormal` (*boolean*, *default = false*)
Absorption of particles, especially at curved surfaces, can be complicated with finite precision computations. In pathological cases, particles may occasionally be absorbed in such a way that the surface normal (at absorption location) is unknown; that is usually because the particles are not absorbed at a surface. For example, a particle may be (intentionally or not) loaded inside an absorber, i.e., so that a particle may be absorbed without crossing the surface of the absorber. If this option is `true` then secondaries will still be emitted even for primaries for which no surface normal is known (in this case, the emission location is probably not the surface of the absorber, but may be deep in the bulk absorber). Typically the only reason for this to be `true` is if the failure to emit a secondary would result in a systematic conservation error.

`suppressEnergy` (*float*, *optional*, *default = -1.*)
If negative, this has no effect on emission. If this is zero, then emission does not occur if the electric force on the particle (at emission location) points back toward the surface, regardless of the secondary particle's energy. If

this is positive, then emission does not occur if the electric field forces the particle back toward the surface and the electric force is sufficiently strong. Specifically, the electric force on the particle, projected onto the surface normal, will be calculated; if the force is away from the surface, there is no effect (the secondary particle will be emitted). If the force is back toward the surface, then secondary emission is suppressed if the (projected) force times the cell diagonal (in Joules) exceeds (or equals) suppressEnergy. E.g., if suppressEnergy is zero, then emission will be suppressed whenever the electric force points toward the emission surface. If suppressEnergy is 1.6e-19, then emission will be suppressed (for an electron or singly-charged ion) if the electric field times the cell-diagonal-length is greater than 1 V; i.e., an emitted electron with less than 1 eV could not travel a cell-diagonal before being turned back toward the surface. If suppressEnergy is very large, then emission will not be suppressed, just as if suppressEnergy is negative. Note that the calculation of whether secondary emission is suppressed has nothing to do with the energy of primary or secondary particles, but depends only on the local electric field at the location of the primary particle. If non-negative, suppressEnergy is typically set to zero to emit only when the electric force points away from the surface, or it may set to a value on the order of typical secondary particle energy, ensuring that emission will be suppressed only if the electric force would prevent typical particles from traveling a cell-diagonal.

Example secondaryEmitter Blocks

The following secondaryEmitter emits a secondary particle with probability 1 (always), always normal to the surface with a tenth the speed of light (roughly: the spatial part of the 4-velocity is one tenth the speed of light). It also generates a tag for the secondary (applicable only if the secondary species is tagged) of 1.3.:

```
<ParticleSource coneSecEmitter>
  kind = secondaryEmitter
  recordParticleData = true
  gridBoundary = coneSurf
  ptclAbsorber = coneSurfSink
  functionVariables = []

  <Expression sey>
    kind = expression
    expression = 1.0
  </Expression>

  autoGenerateTag = false

  <Expression velocityAndTag>
    kind = expression
    expression = vector(0.1*299792458.0,0.,0.,1.3)
  </Expression>

  ptclCountType = emitCounting
  depositCurrentFromCorner = true
</ParticleSource>
```

The following secondaryEmitter emits a secondary particle, with 90 percent probability, that is specularly reflected with respect to the incident primary particle. This sets the tag of each secondary to -1. Here the `internVars` expression is used just as a demonstration; it doesn't change the secondary particle properties. (Note that the first velocity component is the 4-velocity of the primary particle projected onto the surface normal pointing into the absorber; this component represents the secondary's 4-velocity projected onto the surface normal pointing out of the absorber—i.e., the secondary's velocity is the primary's reflected about the surface.):

```
<ParticleSource specSecEmitter>
  kind = secondaryEmitter
  gridBoundary = specSurf
```

(continues on next page)

(continued from previous page)

```
  ptclAbsorber = specSurfSink
  functionVariables = ['gammaVelocity_0'  'gammaVelocity_1' \
           'gammaVelocity_2'  'surfaceNormalIntoAbsorber_0' \
           'surfaceNormalIntoAbsorber_1'  'surfaceNormalIntoAbsorber_2' \
           'surfaceTangent1_0'  'surfaceTangent1_1'  'surfaceTangent1_2' \
           'surfaceTangent2_0'  'surfaceTangent2_1'  'surfaceTangent2_2']

  <Expression sey>
    kind = expression
    expression = 0.9
  </Expression>

  autoGenerateTag = false

  <Expression velocityAndTag>
    kind = expression
    expression = vector( \
     (gammaVelocity_0*surfaceNormalIntoAbsorber_0+ \
      gammaVelocity_1*surfaceNormalIntoAbsorber_1+ \
      gammaVelocity_2*surfaceNormalIntoAbsorber_2), \
    (gammaVelocity_0*surfaceTangent1_0+ \
     gammaVelocity_1*surfaceTangent1_1+ \
     gammaVelocity_2*surfaceTangent1_2), \
    (gammaVelocity_0*surfaceTangent2_0+ \
     gammaVelocity_1*surfaceTangent2_1+ \
     gammaVelocity_2*surfaceTangent2_2),\
    -1.)
  </Expression>

  <Expression internVars>
    kind = expression
    expression = vector(\
      secInternVars_0,secInternVars_1,secInternVars_2,secPtclWeight)
  </Expression>

  ptclCountType = vwCounting
  depositCurrentFromCorner = true
</ParticleSource>
```

sputter

Sputtering is a method of depositing both thin metal films and insulators onto a substrate. Vorpal provides a method of simulating this behavior using the sputter kind of ParticleSource. Using sputter, you can simulate the emission of sputtered neutral atoms from solid materials when bombarded by energetic ions. If a beam of ions impacts a wall on which a sputter emitter is defined, these incident ions deposit energy into the wall, initiating neutral transport to the material surface and subsequent emission back into the simulation domain. The incident ions are absorbed by the wall. Yield and angular dependence of emitted atoms is based on the Yamamura[Yamamura1996] sputtering model.

Only ions impacting a wall are used to calculate the source for sputtered neutrals in TxPhysics; electrons cannot be used as impactors. If the incident particles are electrons then secondary electrons are the only emitted particles. The underlying physics models determine the number of sputtered neutrals emitted; there is no artificial limit to the maximum number of sputtered neutrals emitted. The sputtered neutral yield is determined primarily from the nuclear stopping power of the ion/target combination. See the *TxPhysics Manual* for a complete description of the models and algorithms used to determine the ion-induced neutral sputter yield.

This kind of particle source is available with a VSimPD license.

sputter Parameters

direction
: Direction for the source of the outward-facing normal; the algorithm uses this to determine the direction in which sputtered electrons are emitted. If this direction vector is incorrect, it may result in particles being emitted outside the simulation domain, leading to crashes. Sputtered neutrals are always emitted with a positive normal velocity component, i.e., moving away from the emitting surface.

ptclAbsorber
: Specifies the absorber that will remove particles generated by this ParticleSource.

sputterAtomType (*string*)
: Defines the material of which the source is composed. Material choices are:

- `carbon` (C)
- `aluminium` (Al)
- `silicon` (Si)
- `chromium` (Cr)
- `iron` (Fe)
- `nickel` (Ni)
- `copper` (Cu)
- `germanium` (Ge)
- `silver` (Ag)
- `osmium` (Os)
- `platinum` (Pt)
- `gold` (Au)
- `uranium` (U)

sputterAtomSpecies (*string*)
: Name of the neutral atom species that will be sputtered.

secVSig (*float*)
: value defining the velocity spread of the sputtered atoms.

ignoreProb (*float*)
: The fraction of electrons that does not get absorbed. Default value is 0.

Example sputter Block

```
# The sputter emitter
<ParticleSource rightSputterEmitter>
    kind = sputter
    minDim = 1
```

(continues on next page)

(continued from previous page)

```
    ptclAbsorber = rightAbsorber
    # The 'direction' vector should point along the
    # *outward-facing* normal...be sure to check this!
    direction = [1. 0. 0.]
    # The material that makes up the wall
    sputterAtomType = Cr
    # The species of neutral atoms that is being
    # sputtered off the wall
    sputterAtomSpecies = chromiumAtoms
    # The velocity spread of the sputtered atoms
    secVSig = CHROMIUMVSIG
</ParticleSource>
```

userDefinedSecElec

This particle source normally emits electrons based on the user defined secondary electron yield (SEY) data. The number of secondaries will be determined based on the SEY data file given by the user. The determination of energy spectrum and angular distribution of these secondary electrons can be found in *[FP02]*. The secondary electrons can be in the same species as the incoming, or a separate species.

This particle source is available with a VSimMD or VSimPD license.

userDefinedSecElec Parameters

ptclAbsorber (*string*)
The name of the absorber that collects the impacting particles.

yieldDataFile (*string*)
Name of the SEY data file. The first line of the data file should be an integer number of the data points given in the file. The subsequent lines consist of two columns, the first contains the incident electron energy values (eV). The second column contains the corresponding SEY values. The actual yield is interpolated with weights determined by interpolations from the incident energy.

secondarySpecies (*string*)
Name of the emitted secondary electrons.

direction (*double vector*)
Direction vector should point along the outward-facing normal of the particle absorber.

material (*string*)
The material being impacted by the incident particles. If the incident particle is an electron the material choices are:

- `copper`
- `stainless` (stainless steel)

When the incident particle is an ion, the choices are:

- `hydrogen`
- `helium`
- `carbon`
- `nitrogen`
- `oxygen`

- `sodium`
- `aluminum`
- `silicon`
- `phosphorus`
- `argon`
- `iron`
- `nickel`
- `copper`
- `silver`
- `gold`
- `uranium`
- `air`
- `water`
- `stainless` (stainless steel)

Example userDefinedSecElec Block

```
<ParticleSource rightSecondaryEmitter>
  kind = userDefinedSecElec
  yieldDataFile = userIonIndSEY.dat
  ptclAbsorber = rightAbsorber
  direction = [1. 0. 0.]
  material = copper
  secondarySpecies = electrons
</ParticleSource>
```

userFuncSecElec

A secondary electron emitter where the SEY (secondary electron yield) and emitted electron spectrum are defined by a pair of Expressions (see *Introduction to UserFuncs and Expressions*). The secondary electrons can be in the same species as the incoming electrons, or a separate species.

This particle source is available with a VSimMD or VSimPD license.

userFuncSecElec Parameters

`direction` (*vector*)
Direction vector should point along the outward-facing normal of the ParticleSource.

`ptclAbsorber` (*string*)
Name of the absorber that absorbs the electrons and from which the secondaries will be emitted.

`Expression` (*block, required*)
An expression (see *Introduction to UserFuncs and Expressions*) that defines SEY. It must be named `SEYFunc` It can take three arguments `eng` (incident energy in eV), `alpha` (incident angle in rad) and `t` (time in second). Not all the arguments need to be used but any argument must come from these three.

Expression (*block*, *required*)
An expression named `SpectrumFunc` (see *Introduction to UserFuncs and Expressions*) that defines secondary velocity distribution in terms of a three component vector (bn, bt, bp). The name `SpectrumFunc` and a three-component form of this vector are required. `SpectrumFunc` returns the velocity of a particle (bn, bt, bp) in units of c or c/γ, depending on how the flag emissionVelocityInUnitsOfC is set. The components (bn,bt,bp) are all velocities written in these units. bn is the component normal to the surface, i.e., bn is the velocity in the direction opposite to the outward surface normal (or opposite the direction attribute if no `gridBoundary` is specified). bt is the component in the direction of any motion the incident particle had tangential to the surface, i.e., bt is the velocity in the direction $t = v_{in} \times n$ (a cross product between the incoming particle velocity and the outward surface normal). bp is the velocity in the direction tangential to the surface in which the incident particle had no velocity component, i.e., bp is the velocity in the direction of $p = n \times t$. The particles are emitted opposite the surface outward normal.

Here, v is the 3-velocity; if SpectrumFunc specifies a value for v with magnitude larger than 1 (velocity greater than the speed of light), it will assume that for this particle the velocity is in units of c/γ. If this happens the user should really set emissionVelocityInUnitsOfC to false (0).

`SpectrumFunc` can take two arguments `eng` (incident energy in eV) and *t* (time in seconds). Not all the arguments need to be used but any argument must come from these two. The vector bases (bn, bt1, bt2) default to (1,0,0).

In the future, this will likely be changed to return the 4-velocity components instead of 3-velocity components.

emissionVelocityInUnitsOfC (*boolean*, *default = 0*)
Whether emission velocities are in units of c or in units of c/γ. For non-relativistic particles these options are almost the same. For relativistic particles the differences can be significant, and setting this to 0 allows for velocities with magnitude greater than 1.

emissionProb (*real*)
Specifies the probability for emission of secondaries from a material surface. For the constant probability model, a single electron is emitted with a given probability independent of all other factors, such as the incident energy, material and primary ion properties, etc.

ptclCountType (*string*, *default = emitCounting*)
Describes how the emission of multiple secondary electrons is handled. This parameter is optional, and defaults to emitCounting. Valid types are:

- `emitCounting` (default)

  Emit multiple macroparticles.

- `noCounting`

  Emit single particle.

- `vwCounting`

  Emit single variable-weight particle and increase its weight by the number of secondaries.

- `taggedVwCounting`

  Emit single variable-weight, tagged particle and increase its weight by the number of secondaries.

secondarySpecies (*string*, *required for ion primaries*)
Electron species to emit the secondaries to for ion primaries.

emittingSurface (*float*, *optional*)
Physical position of the emitting surface in the direction of emission. If this parameter is not specified, it is calculated to be the appropriate edge of the grid. It is ignored when emitting from a grid boundary.

suppressEnergy
By default, emission does not occur if the electric field has sign that would immediately force the emitted

particle back into the surface (default value is `suppressEnergy=0`). This parameter can be used to control this feature. If an emitted particle is desired, regardless of the sign of the electric field, then set this parameter to a very large number, e.g., `suppressEnergy=1.0e32`.

More specifically, emission occurs when the particle charge, times the local field strength, times a characteristic length based on the grid, e.g., q*E*dx, is less than the `suppressEnergy`. The local field strength is interpolated on the emission surface. In higher dimensions, the characteristic length is the diagonal across the cell. The units of the `suppressEnergy` are electronVolts.

`ignoreProb` (*float*, *default = 0*)
Float value representing probability to ignore an absorbed electron so it can be used for another process. This should be a value between `0` and `1`.

Example userFuncSecElec Block

```
<ParticleSource secondaryEmitter>
  kind = userFuncSecElec
  minDim = 1
  ptclAbsorber = primary.plateAbsorber
  gridBoundary = absPlate

  <Expression SEYFunc>
    expression = vaughanSEY(eng, alpha)
  </Expression>

  <Expression SpectrumFunc>
    $ betaGammaSqr = (2.0*eng*xDist()/ELECMASSEV)
    $ betaGamma = sqrt(betaGammaSqr)
    $ gamma = sqrt(1 + betaGammaSqr)
    $ beta = betaGamma/gamma
    expression = betaGamma * dirDist()
  </Expression>

</ParticleSource>
```

3.11.4 PositionGenerator Block

PositionGenerator

Block provides the number of *Particle emission attempts* and *Particle load attempts*, and the location where they attempt to be emitted or loaded. Whether or not the particle is actually loaded or emitted depends on the value of the **`relMacroDenFunc`** or **`relMacroFluxFunc`** function at that position and time.

Note: The use of **`relMacroDenFunc`** modifies *`ptclsPerCell`* in **`bitRevSlabPosGen`**.

All xvLoaderEmitter and boostLoader blocks must have a PositionGenerator block. A PositionGenerator block is contained within a ParticleSource block.

PositionGenerator is used by the xvLoaderEmitter and boostLoader particle sources to determine the positions of particles when loaded or emitted into the simulation. The position generator loads and/or emits particles based on slab objects, specifying the upper and lower bounds of a slab region. Particle *loading* occurs inside a volume (3D), while particle *emission* occurs from a surface (2D).

Loading particles: the PositionGenerator location is the starting location of the particle at the beginning of the time-step, and the particle experiences a normal dynamic push to get the location at the end of the time step.

Emitting particles: an additional uniform random number between 0 and 1 is generated, which provides the exact time of emission within the time step, after which a fractional dynamic push is performed to obtain the location at the end of the time step.

There are three possible kinds of PositionGenerator:

- *bitRevSlabPosGen*
- *cutCellPosGen*
- *gridPosGen*

See the documentation for each of these kinds for more details.

PositionGenerator Sub-Blocks

Slab (block)

`<Slab loadSlab>` Particle loading volume: Nested block that defines the physical volume into which particles will be loaded. loadSlab is defined by specifying the lowerBounds (lower-left-back corner of the region) and the upperBounds (upper-right-front corner of the region) in physical coordinates.

`<Slab emitSurface>` Particle emitting surface: Surface from which particles will be emitted. You can define the emitSurface only if particles are to be emitted into the simulation, starting with zero loaded particles. emitSurface is defined in a Slab block by specifying the lowerBounds (lower-left-back corner of the region) and the upperBounds (upper-right-front corner of the region) in physical coordinates. To define a surface, one of lowerBounds and upperBounds components must be equal in the emitSurface.

If you do not define a slab block for either a loading or emitting operation, Vorpal will not try to perform the operation that corresponds to the missing the slab.

Note: Use of the loadSlab or emitSurface parameter is mutually exclusive based on whether particles are initially loaded or emitted, respectively.

PositionGenerator Parameters

Note: Input parameters differ for *loading* and *emitting* particles. See below for details.

`kind`: PositionGenerator algorithm to be used is one of:

- ***bitRevSlabPosGen*:** *Loads* and *emits* particles based on a slab object within which bitRevSlabPosGen generates positions for particles based on the bit-reversed algorithm; can also emit particles from a surface that is normal to one of the coordinate axes. See the *bitRevSlabPosGen* description for details and parameters.
- ***cutCellPosGen*:** As an *emitter* it generates particle positions on the surface of a grid boundary using a triangulated approximation of that boundary and/or as a *loader* generates particles into a region defined by a grid boundary.

cutCellPosGen only works with the xvLoaderEmitter and in combination with a velocity generator. See the *cutCellPosGen* description for details and parameters. Use is enabled by VSim for Plasma Discharges and VSim for Microwave Devices licenses.

- ***gridPosGen*:** *Loads* and *emits* particles based on slab objects, specifying the upper and lower bounds of a slab region, laying the particles down on at the nodes of a uniform Cartesian grid different from the grid on which fields are located. See the *gridPosGen* description for details and parameters.

`ptclsPerCell`
For Loading Particles - Indicates nominal number of macroparticles the source assumes will be in cell.

`sweepRate` (*floating point*, *default = 0*)
For Emitting Particles - Determines the width of the region (measured from the **`emitSurface`** in the direction specified by **`emitSign`**) in which newly emitted particle positions are generated. The width of this region will be equal to **`sweepRate`** multiplied by the size of the time step. sweepRate, together with the area of the emission surface and the time step, defines the volume into which Vorpal emits or loads particles. sweepRate is usually set to the average speed of the particles being emitted. If this value is set and `nomMacroPtclsPerStep` or `nomMacroPtclsPerCellStep` (when using `bitRevSlabPosGen`) is set then it causes an exception and it asks the user to choose only one option.

sweepRate is used in the following way to determine how many particles are emitter per step:

- nomMacroPtclsPerAreaStep = sweepRate * dt * nominalDensity / numPtclsInMacro
- nomMacroPtclsPerStep = nomMacroPtclsPerAreaStep * emissionArea

`nomMacroPtclsPerStep` (*floating point*, *default = 0*)
For Emitting Particles - Specifies the nominal macro particles to emit per time step. If this value is set and `sweepRate` or `nomMacroPtclsPerCellStep` is set then it causes an exception. The user is advised to choose only one option.

`emitSign` (*integer*, *default = 1*)
For Emitting Particles - Specifies the direction from which the particles are to be emitted from the emitSurface; one of `+1` or `-1`.

`applyTimes` (*[float,float], default = [0,0]*)
Start and stop times for loading particles. Must specify loadAfterInit (see below).

`loadAfterInit` (*boolean*, *default = 0*)
Must be set with `applyTimes`. If set to 1, it enables loading after initialization.

`loadOnShift` (*boolean*, *default = 0*)
Applied for moving-window simulations.

`emitBasedOnLocalForce` (*boolean*, *default = 0*)
Set to true in order to prevent emission when the local electric field has sign such that the force returns particles to the surface.

`useCornerMove` (*boolean*, *default = 0*)
Set to true to specify corner move dynamics on the initial step, when emitting from cut cells. If false, uses the parallel move from 1 cell deep. Using true, e.g., corner move, is more robust in some circumstances. Ignored if not using a kind = `cutCellPosGen` type of <Position-Generator>.

Note: If the user considers an emission source and does not set `sweepRate` or `nomMacroPtclsPerStep` or `nomMacroPtclsPerCellStep`, then it causes an exception and the user is advised to specify one of these options.

PositionGenerator Kinds

bitRevSlabPosGen

Position generator that *loads* and *emits* particles based on a slab object. The slab object is defined by specifying the upper and lower bounds of a rectangular slab region. Within this slab region, bitRevSlabPosGen generates positions for particles based on the bit-reversed algorithm.

For *loading* particles into a volume (usually at initialization only), you must specify a slab block that has the name loadSlab.

bitRevSlabPosGen can also *emit* particles from a surface that is normal to one of the coordinate axes. To specify this surface, you must have a slab block with the name emitSurface and one (and only one) of the components of the upper and lower bounds vectors must be equal to each other. The component of the upper and lower bounds that is equal specifies the normal direction to the emitSurface, and the other components determine the area of the emitSurface.

To emit particles from the emitSurface, you must also specify the ptclsPerCell parameter for loading as well as the emitSign and the sweepRate parameters.

bitRevSlabPosGen Parameters when Loading

ptclsPerCell (*integer*, *default = 0*)
Determines the number of macroparticles that will be loaded per cell inside the loadSlab. Loading particles also requires that the user specify the *ptclsPerCell* parameter. The default (0) is that no particles are loaded.

numPhysPtclsPerStep (*float*, *default = 0.0*)
Determines the number of physical particles to load per time step. When this option is given it overrides the ptclsPerCell parameter.

Example bitRevSlabPosGen Block in Loading Mode

```
<PositionGenerator bitRevSlab>
  kind = bitRevSlabPosGen

  <Slab loadSlab>
    lowerBounds = [$XSTART +LX/2. - 1*DX$ $YSTART + LY/2. -DY$ $ZSTART + LZ/2. -DZ$]
    upperBounds = [$XSTART + LX/2.$      $YSTART + LY/2.$      $ZSTART + LZ/2.$]
   </Slab>

  ptclsPerCell = NOMPPC
</PositionGenerator>
```

bitRevSlabPosGen Parameters when Emitting

nomMacroPtclsPerCellStep (*floating point*, *default = 0*)
Emitting only - Specifies the nominal macro particles to emit per cell per time step. If this value is set and `sweepRate` or `nomMacroPtclsPerStep` set in `positionGenerator`, then it causes an exception and the user is advised to choose only one option.

Note: If the user considers an emission source and does not set `sweepRate` or `nomMacroPtclsPerStep` (in `positionGenerator`) or `nomMacroPtclsPerCellStep`, then it causes an exception and the user is advised to specify one of these options.

---

emitSign (*integer*, *default = 1*)
Determines the direction of emission from the `emitSurface`.

Example bitRevSlabPosGen Block in Emitting Mode

```
<PositionGenerator emitSlab>
  kind = bitRevSlabPosGen
  <Slab emitSurface>
    lowerBounds = [$XSTART$ $-NSIG*DY$ $-NSIG*DZ$]
    upperBounds = [$XSTART$ $NSIG*DY$  $NSIG*DZ$ ]
  </Slab>
  nomMacroPtclsPerStep = 10.
</PositionGenerator>
```

cutCellPosGen

As an emitter it generates particle positions on the surface of a grid boundary using a triangulated approximation of that boundary and/or as a loader generates particles into a region defined by a grid boundary.

cutCellPosGen only works with the *xvLoaderEmitter* and in combination with a velocity generator. cutCellPosGen will actually work on problems when no cut cells exist provided in the grid boundary block `calculateVolume = true` is used.

cutCellPosGen Parameters when Emitting

Use emit=true and load=false in the ParticleSource block

emitterBoundary (*string*, *required*)
defines the grid boundary that particles will be emitted from. Particles can be emitted from a subset of the emitterBoundary by the use of a mask. In order for a grid boundary to be used as an emission boundary the grid boundary must have `calculateVolume = true` within the grid boundary block, this allows for the computation of triangulated surfaces which are required for emission.

nomMacroPtclsPerStep (*float*, *required*)
Defines the number of macro particles to be emitted per step.

emitterMask (*string*, *optional*)
Used to mask out parts of the emitterBoundary. The emitterMask is a gridBoundary where emission regions are defined to be positive and non-emission regions are `0` or negative. The intersection of the positive regions of the emitterMask with the surface of the emitterBoundary defines the region where particles can be placed on the surface for emission. The emitterMask always points to a grid boundary and never to an *STFunc Block* or other structure.

mask (*block*, *optional*)
A mask for the emitterBoundary can be specified using an *STFunc Block* `<STFunc mask>` with the explicit name "mask" instead of the emitterMask. This allows the mask to be defined using an STFunc, which requires no geometry data. The emission region is computed once at the beginning of the simulation so time dependence of the STFunc is effectively ignored in the current implementation. Emission regions are defined to be positive and non-emission regions are `0` or negative.

emissionOffset (*float*, *optional*)
Defines the distance that particles are offset from the desired emission location on a grid boundary surface. The emissionOffset is measured as a fraction of the average cell edge length in each direction. So offset=emissionOffset*(dx+dy+dz)/3. The emission offset is often required to prevent particles from being immediately absorbed since emitters and absorber frequently exist on the same grid boundary. Furthermore, when the absCutCell absorber is used, the absorption boundary can be slightly ahead of the emission boundary resulting in patches of particles that are absorbed upon emission. emissionOffset should be a value between 0 and 1 with smaller values putting the particles closer to the true emission region. In many cases an emission offset of `0` will work.

positionFunction (*string*, *optional*, *default = random*)
Choose the type of loading, either bit reversed (quasi-uniform) or random, valid options `random`, `Random`, `bitReversed` or `bitreversed`.

Note: In serial runs, the position generator will dump exactly nomMacroPtclsPerStep macroparticles per step. In parallel runs, where the emitter is cut by block boundaries, only approximately this number of particles will be emitted.

Note: If both an STFunc mask and an emitterMask are defined, then the emission region becomes the intersection of the STFunc mask the emitterMask and the surface of the emitterBoundary.

Note: If you do not specify an emitterMask or <STFunc mask> then Vorpal emits particles from the entire emitterBoundary.

cutCellPosGen Parameters when Loading

Use emit=false and load=true in the ParticleSource block

gridBoundary (*string*, *required*)
Defines the grid boundary to be used for loading. Particles are loaded wherever this gridBoundary is positive.

ptclsPerCell (*int*, *required*)
Defines the number of particles to be distributed per cell on average.

positionFunction (*string*, *optional*, *default = random*)
Choose the type of loading, either bit reversed (quasi-uniform) or random, valid options `random`, `Random`, `bitReversed` or `bitreversed`.

When you use cutCellPosGen to load particles, Vorpal loads the particles into a region defined by a grid boundary. An alternative method to loading particles in a geometric region is by defining a loadSlab and then using <STFunc relMacroDenFunc> to reject particles outside the region defined by the <STFunc relMacroDenFunc>. However, this method is much less efficient since many particles may be rejected. In contrast, cutCellPosGen loads only particles within the specified region, rejecting a small number of particles.

It is also possible to use an <STFunc relMacroDenFunc> in conjunction with the cutCellPosGen. First define the loading region based on the gridBoundary and next use the relMacroDenFunc to either limit it further or give the loading a spatially varied shape.

Note: gridBoundary and emitterBoundary are actually the same variable, so if both `load = true` and `emit = true` only one of these grid boundaries needs to be defined as the other is redundant. Vorpal first checks for the existence of emitterBoundary, if emitterBoundary is not found then Vorpal looks for gridBoundary.

Example cutCellPosGen Block in Emitter Mode

Note: The names cathode and theMask refer to a pre-defined grid boundary.

```
<PositionGenerator thisGen>
    emit = true
    load = false
    kind = cutCellPosGen
    nomMacroPtclsPerStep = 20.0
    emitterBoundary = cathode
    emitterMask = theMask
    emissionOffset = 0.1
    <STFunc mask>
        kind=expression
        expression = x
    </STFunc>
 </PositionGenerator>
```

gridPosGen

Position generator that loads and emits particles based on Slab objects, just as is done with the *bitRevSlabPosGen*. In contrast to the bitRevSlabPosGen position generator, gridPosGen generates positions for particles at the nodes of a user-defined uniform grid.

All but one of the parameters used to define the bitRevSlabosGen are used to identically define the gridPosGen. The single difference is that the gridPosGen uses a *`macroPerDir`* vector instead of the `ptclsPerCell` parameter. macroPerDir is a vector of integers with components that default to 1. Each component of the macroPerDir vector determines the number of macroparticles generated per cell in that coordinate direction. A macroPerDir vector equal to `[2 3 1]`, in a 3-dimensional simulation, will generate positions for particles on a grid with 2 macroparticles per cell in the x-direction, 3 macroparticles per cell in the y-direction, and 1 macroparticle per cell in the z-direction. Therefore, the total number of macroparticles per cell will be equal to 2 multiplied by 3 multiplied by 1, or 6 macroparticles per cell.

gridPosGen Parameters

`macroPerDir` (*integer vector, default = [1 1 1]*)
Components of the macroPerDir vector each determine the number of macroparticles generated per cell in the corresponding coordinate direction. For example, a macroPerDir vector equal to `[2 3 1]`, in a 3-dimensional simulation, will generate positions for particles on a grid with 2 macroparticles per cell in the x-direction, 3 macroparticles per cell in the y-direction, and 1 macroparticle per cell in the z-direction. Therefore, the total number of macroparticles per cell will be equal to 2 multiplied by 3 multiplied by 1, or 6 macroparticles per cell.

`emitSign` (*integer, default = 1*)
Determines the direction of emission from the emitSurface.

Example gridPosGen Block

```
<PositionGenerator gridSlab>
  kind = gridPosGen
# The following gives the slab over which particles are loaded.
# If it is not present or has zero volume, no particles are loaded.
  <Slab loadSlab>
    lowerBounds = [0.0 0.0 0.0]
    upperBounds = [LX LY LZ]
  </Slab>
  macroPerDir = [1 1 1]
</PositionGenerator>
```

3.11.5 VelocityGenerator Block

VelocityGenerator

Used by *xvLoaderEmitter*, *gridLoader* and *boostLoader* particle sources to determine the velocities (and all internal variables) of particles when loaded or emitted into the simulation. Each xvLoaderEmitter or gridLoader or boostLoader block must have a VelocityGenerator block.

A VelocityGenerator block defines how the newly added (loaded or emitted particles) get their velocity. VelocityGenerator is used to generate **all** internal variables associated with the ParticleSource species, including particle weight for variable-weight species. Using VelocityGenerator features you could design a 3D simulation with variable-weight particles that might define a beamVelocityGen with 4 components for both vbar and vsig.

Note: Internal variables associated with particles, such as weights when using variably-weighted particles, are contained within the velocity vector. Therefore, the means (vbar) and the deviations (vsig) should also be specified for these other velocity components. Otherwise, the default value for unspecified velocity components is zero.

VelocityGenerator Parameters

kind

Type of VelocityGenerator; one of:

- ***funcVelGen*:** Determines the velocity components based on a user-specified space-time function for each component. That is, the velocity is determined by where and when the particle is placed.
- ***userFuncVelGen*:** Determines the velocity components based on a user-specified UserFunc; this allows correlations between components, as well as access to arbitrary random distributions.
- ***beamVelocityGen*:** Simplest kind of velocity generator; allows you to specify the mean (vbar) and standard deviation (vsig) of a Gaussian distribution, from which particle velocities will be selected at load/emission time.
- ***childLangmuirVelGen*:** Determines the weights of the particles (and therefore the current) based on the Child-Langmuir Law.
- ***kappaVelGen*:** Beam velocity generator loads a beam of particles.
- ***fieldEmitterVelGen*:** Velocity generator that fixes the component of the velocity based values of a given set of space-time.

- ***fieldScaleVelGen*:** Special velocity generator only for use with `relBorisVWScale` particles. The velocity generator correctly sets the tags for the particles as well as creating several particles at each location each with a different scale parameter that scales the electromagnetic fields the particle experience.

`velocityIsLocal`
Specifies the orientation of the component references in the block. When false, components refer to the overall coordinate system axes, and when true, components refer to local orientation of the emission surface, that is to say, component0 = normal, component1 = parallel_1, and component2 = parallel_2, where the choice of the two parallel directions are not available to the user. A setting of true is commonly used when emitting from a cut-cell surface, in order to ensure normal emission. In this case, the normal direction is inward, towards the surface, so that the sign of the component0 velocity is typically negative to ensure outward flow of particles.

`seed` (*integer*)
One can specify a `seed` or `randomSeed` input argument for Vorpal's random number generator, so as to get reproducible behavior.

`randomSeed` (*integer*)
One can specify a `seed` or `randomSeed` input argument for Vorpal's random number generator, so as to get reproducible behavior.

VelocityGenerator Kinds

beamVelocityGen

Gaussian distribution velocity generator.

beamVelocityGen Parameters

`vbar` (*vector*, *default = 0*)
Specifies the addition of a constant velocity to whatever velocity is returned from the specific kind of velocity generator. *vbar* can be specified for internal variables such as weight and tag, as well as the actual velocity components.

`vsig` (*vector*, *default = 0*)
Specifies the addition of a random velocity to whatever velocity is returned from the specific kind of velocity generator. The random velocity is sampled from a Gaussian with standard deviations for each component given by the value of vsig. The seed of the sampling can also be provided with the seed attribute. *vsig* can be specified for internal variables such as weight and tag, as well as the actual velocity components.

`emissionQuad` (*integer*, *default = 0*)
This defines the emission quadrant and thus the velocity directions based on which quadrant is selected. The default for this option is zero, in which case no emissionQuad check is performed on the particle velocities. The options are as follows:

- **First quadrant (`emissionQuad = 1`)** Velocities will be in the first quadrant, i.e. all positive.
- **Second quadrant (`emissionQuad = 2`)** Velocities will be in the second quadrant. The particle velocities in the x and y directions (V_0 and V_1) are negative and positive, respectively.
- **Third quadrant (`emissionQuad = 3`)** Velocities will be in the third quadrant, so V_0 will be negative and V_1 will also be negative.
- **Fourth quadrant (`emissionQuad = 4`)** Velocities will be in the fourth quadrant, so V_0 and V_1 will both be positive.

beamVelocityGen Example Block

```
<VelocityGenerator VelGen>
  kind = beamVelocityGen
  vbar = [VELOCITY 0.0 0.0]
  vsig = [0.0 0.0 0.0]
  emissionQuad = 1
</VelocityGenerator>
```

childLangmuirVelGen

Velocity generator that determines the weight of the particles (and therefore the current) based on the Child-Langmuir Law. The current is simply calculated to vary linearly with the normal E field. Ampere's law is solved, recognising that this represents the case where at the nearby virtual cathode surface $E = 0$, as this represents the potential minimum.

childLangmuirVelGen Parameters

reduction (*option*)
The factor by which emitted current is reduced. e.g. if reduction = 0.1 then you emit 1/10 the child-langmuir current. Reduction = 1.0 is the full child-langmuir current

STFunc (*block*, *option*)
STFunc Block with function name specifying component#, where '#' is the component number (0,1,2, . . .) for each velocity component you wish to specify.

The component# space-time function blocks are specified like any other Vorpal space-time function block, and the value of this function (depending on the position and time of the particle's placement at load or emission time) is used to determine the value of the specified component of the velocity.

childLangmuirVelGen Example Block

```
<VelocityGenerator emitVelGen>
  kind = childLangmuirVelGen
  reduction = 0.1 #1.0e-3

  # Optionally set individual components of the velocity with STFunc blocks
  <STFunc component0>
    kind = expression
    expression = -2.0e5
  </STFunc>
</VelocityGenerator>
```

fieldEmitterVelGen

Velocity generator that fixes the components of the velocity-based values of a given set of space-time functions.

fieldEmitterVelGen Parameters

velocityIsLocal (*boolean*)
Specifies the orientation of the component references in the block. When false, components refer to the overall coordinate system axes, and when true, components refer to local orientation of the emission surface. A setting of true is commonly used when emitting from a cut-cell surface, in order to insure normal emission. In this case, the normal direction is inward, towards the surface, so that the sign of the component0 velocity is typically negative to insure outward flow of particles.

work_function (*real*)
Work function for the metal.

alpha (*real*)
Normalized offset distance where `alpha = 1` corresponds to the length of the hypotenuse of one cell. If `alpha = 1` the electric field is sampled normal to the boundary a distance alpha*H from the emission point (where H is the hypotenuse). This factor needed since the field at the boundary is smaller than the field just inside the boundary. The field just inside the boundary is the correct field.

temperature (*real*)
Temperature of the metal.

multiplier (*real*)
Multiplication factor to enhance the current.

field_enhancement (*real*)
Multiplication factor to enhance the perceived electric field.

emitterType (*string*)
One of:

- *Richardson-Dushman*
- *Fowler-Nordheim*
- *txemit*

Richardson-Dushman Emitter

Emitter that evaluates the equation:

$$J = M\frac{4\pi me}{h^3}T^2\exp\left(-\frac{W}{T}\right), \tag{3.20}$$

where:

J is the resulting magnitude of the current density, M is the multiplier, m is the electron mass, e is the electron charge, h is Planck's constant, T is the temperature in appropriate units, and W is given by

$$W = W_0 - \frac{e}{4\pi\epsilon_0}|\mathbf{E}f|, \tag{3.21}$$

where:

$\mathbf{E}$ is the surface electric field, f is the enhancement factor, and W_0 is the work function.

Richardson-Dushman Emitter Parameters:

field_enhancement (*double*, *default = 1*)
Multiplies the measured electric field.

multiplier (*double*, *default = 1*)
Multiplies the resulting output current.

work_function (*double*, *default = 4.5*)
Minimum energy necessary to extract an electron from the surface; measured in electron volts.

temperature (*double*)
Temperature of the surface in Kelvin.

In the Richardson-Dushman emitter, the work function may be specified using an STFunc block named *workFunction*. The work function defined in the STFunc block overwrites the value of the work_function keyword.

Syntax to Specify the Work Function as an STFunc Block:

```
<VelocityGenerator emitVelGen>
  kind = fieldEmitterVelGen
  velocityIsLocal = false
  work_function = 4.5
  alpha = 0.1
  temperature = 2000.0
  field_enhancement = 1.0
  multiplier = 1.0
  emitterType = Richardson-Dushman
</VelocityGenerator>
```

Fowler-Nordheim Emitter

Field emission current density, J_{FN}, is determined by:

$$J_{FN} = \frac{A_{FN}\left(\beta_{FN}E_p\right)^2}{\phi_w}\exp\left(-\frac{B_{FN}v(y)\phi_w^{3/2}}{\beta_{FN}E_p}\right), \tag{3.22}$$

where:

E_p is the perpendicular component of the electric field in V/m, β_{FN} is the field enhancement factor, ϕ_w is the work function of the field-emitting material surface, $v(y)$ has the form

$$v(y) = 1.0 - C_v y^2, \tag{3.23}$$

where:

y is given by:

$$y = \frac{C_y E_y^{1/2}}{\phi_w}, \tag{3.24}$$

and:

A_{FN} , B_{FN} , C_v , and C_y are coefficients of the Fowler-Nordheim field emission model.

Fowler-Nordheim Parameters

beta_FN (*real*, *default = 1.0*)
Field enhancement factor β_{FN} of the Fowler-Nordheim field emission model.

A_FN (*real*, *default = 1.5414e-6*)
Coefficient A_{FN} of the Fowler-Nordheim field emission model.

B_FN (*real*, *default = 6.8308e9*)
Coefficient B_{FN} of the Fowler-Nordheim field emission model.

C_v_FN (*real*, *default = 0.0*)
Coefficient C_v of the Fowler-Nordheim field emission model.

C_y_FN (*real*, *default value = 3.79e-5*)
Coefficient C_y of the Fowler-Nordheim field emission model.

Photo-emission Emitter

Field emitter using the txphysics txemit field-thermal and photo emission emitters.

txemit Photo-emission Parameters:

mu (*real*, *default = 1.8*)
Fermi Level of material.

laser (*boolean*, *default = false*)
True or false. Whether to use a laser for photo emission.

beta_FN (*real*, *default = 1.0*)
Field enhancement factor.

txemit Photo-emission Required Extra Parameters for laser=true:

laserPulseStartTime (*float*, *default = 0.0*)
Starting time for calculating intensity of pulse at the cathode. Required for use with photo-emission Laser pulse.

laserPulseStopTime (*float*)
Ending time for calculating intensity of pulse at the cathode. Required for use with photo-emission Laser pulse.

laserPulseOrigin (*float vector*)
Origin from which the pulse is launched. The origin is defined at the pulse maximum value at the *startTime*. Required for use with photo-emission Laser pulse.

laserPulseIntensityEnvelopeWidths (*float vector*)
Specifies the laser pulse envelope widths. These are defined as twice the standard deviations of the laser pulse intensity envelopes with the first value assigned to the longitudinal envelope length. Required for use with photo-emission Laser pulse.

laserPulseIntensityMagnitude (*floating point*)
Specifies the magnitude of the laser intensity in W/cm2. Required for use with photo-emission Laser pulse.

laserPulsePropagationDirection (*float vector*)
Specifies the direction of propagation of the laser pulse. Make sure that the direction points into the surface of the cathode. By specifying this direction, the incidence angle can be varied. This vector does not have to be specified with a unit length. Required for use with photo-emission Laser pulse.

laserPulseWaveLength (*float*)
Specifies laser pulse wavelength in m. Required for use with photo-emission Laser pulse.

photonEnergy
Calculated variable to hold hbar*omega for the energy of photons in the laser pulse derived from laserPulseWavelength.

omega
Calculated frequency of laser light derived from laserPulseWavelength.

reflectivity (*default = 0.5*)
Material reflectivity; specifies reflectivity of surface.

scattering
Specifies scattering (F_lambda) for surface material.

fieldEmitterVelGen Example

```
<VelocityGenerator  emitVelGen>
    kind = fieldEmitterVelGen
    velocityIsLocal = true
    work_function = 4.5
    field_enhancement = 1.0
    multiplier = 1.0
    #
    # Offset for sampling the electric  field
    # alpha is the normalized offset distance where
    # alpha=1 corresponds to the length  of the hypotenuse
    # of one cell. If alpha=1 the electric field is
    # sampled normal to the boundary a  distance alpha*H
    # from the emission point
    # (where H is the hypotenuse). This factor is needed
    # since the field at
    # the boundary is smaller than the field just inside
    # the boundary... The field just inside the boundary
    # is the correct field. Try setting alpha=0 and
    # compare emission currents.
    #
        alpha = 0.1
        temperature = 300.0
    #
    # begin extra parameters needed for txemit emitterType
    #
        emitterType = txemit
        mu = 7.0
        laser = true
    #
    #  below are necessary subparameter is using laser for
    # photo-emission Laser pulse parameters. For now only a
    # pulse with Gaussian envelope shapes in all directions is
    # implemented. The laserPulseStartTime and
    # laserPulseStopTime are the times between which the
    # intensity of the pulse will be calculated at the cathode.
    #
            laserPulseStartTime = 0.0
            laserPulseStopTime = pulseDuration
        #
        # Set the origin from which the pulse is launched. The
        # origin is defined at the pulse maximum value at the
        # startTime.
        #
            laserPulseOrigin = [0.0 $threeLRMS*5.0$ 0.0]
        #
        # Specify the laser pulse envelope widths. These are
```

(continues on next page)

(continued from previous page)

```
        # defined as twice the standard deviations of the laser
        # pulse intensity envelopes with the first value assigned
        # to the longitudinal envelope length.
        #
            laserPulseIntensityEnvelopeWidths = [twoLRMS twoTRMS1 twoTRMS2]
        #
        # Specify the magnitude of the laser intensity in W/cm2.
        #
            laserPulseIntensityMagnitude = 1.0e+9

        #
        # Set the laser pulse wavelength in m.
        #
            laserPulseWavelength = LASER_PULSE_WAVELENGTH
        #


        # Specify the direction of propagation of the  laser pulse.
        # Make sure that the direction points into the surface of
        # the cathode. By specifying this direction, the incidence
        # angle can be varied. This vector does not have to
        # be specified with a unit length.
        #
            laserPulsePropagationDirection = [0.0 1.0 0.0]
     #
     # Specify Reflectivity of surface default set to 0.5
            reflectivity = 0.5

     # Specify Scattering (F_lambda) for surface material
     #
        scattering =  0.043
     #
     # end of extra parameters needed by txemit
     #
        beta_FN = 100.0 #field enhancment for txemit also
     #
        <STFunc  component0>
            kind = expression
            expression = -1.0e6
        </STFunc>
        </VelocityGenerator>
```

fieldScaleVelGen

Special velocity generator only for use with *relBorisVWScale* particles. The velocity generator correctly sets the tags for the particles as well as creating several particles at each location each with a different scale parameter that scales the electromagnetic fields the particle experience.

fieldScaleVelGen Parameters

fieldScaleEndPoints (*float vector*)
The start and end point for the scaling parameters. This must be used with *fieldScaleIncrement*.

fieldScaleIncrement (*float*)
The spacing between scaling parameters. This must be used with *fieldScaleEndPoints*.

fieldScaleValues (*float vector*)
Explicit values for the scaling parameters.

fieldScaleVelGen Example Block

```
<VelocityGenerator velGen>
  kind = fieldScaleVelGen
  # The field scale parameters set as a vector of values
  fieldScaleValues = [0.2 0.4 0.6]
</VelocityGenerator>
```

funcVelGen

Velocity generator that determines the velocity components based on a user-specified space-time function for each component. That is, the velocity is determined by where and when the particle is placed.

The funcVelGen VelocityGenerator sub-block of the ParticleSource block contains an *STFunc Block* block named component# (where '#' is the component number: 0, 1, . . .) for each velocity component you can specify.

funcVelGen Parameters

kind
Type of VelocityGenerator (e.g., beamVelocityGen).

STFunc (*block*)
STFunc Block with function name specifying component#, where '#' is the component number (0, 1,2, . . .) for each velocity component you wish to specify.

The component# space-time function blocks are specified like any other Vorpal space-time function block, and the value of this function (depending on the position and time of the particle's placement at load or emission time) is used to determine the value of the specified component of the velocity.

Note: For loading variable weight particles, a user may specify a component3 which will set the weight of the particles. The weight is an additional modifier to macroparticles applied in addition to the nominal value of the Number of Particles in a Macroparticle (NPIM). The NPIM is calculated according to the equation (nominal particle density)*(cell volume)/(Particles Per Cell). For constant weight particles, do not specify a component3.

For emitting variable weight particles, a currentDensityFunc may override a setting of component3 in the `funcVelGen`, or the component3 may be ignored entirely.

funcVelGen Example Block

```
<VelocityGenerator emitVelGen>
  kind = funcVelGen
  velocityIsLocal = true
  <STFunc component0>
    kind = expression
```

(continues on next page)

(continued from previous page)

```
    expression = -1.0e7
  </STFunc>

  <STFunc component3>
    kind = expression
    expression = 1.0
  </STFunc>
</VelocityGenerator>
```

kappaVelGen

Velocity generator to load a beam of particles.

kappaVelGen Parameters

`seed` (*integer*)
: Sets the seed for the random number generator.

`lowerLimit0` (*float*)
: Lower limit of velocity in the x-direction.

`upperLimit0` (*float*)
: Upper limit of velocity in the x-direction.

`lowerLimit1` (*float*)
: Lower limit of velocity in the y-direction.

`upperLimit1` (*float*)
: Upper limit of velocity in the y-direction.

`lowerLimit2` (*float*)
: Lower limit of velocity in the z-direction.

`upperLimit2` (*float*)
: Upper limit of velocity in the z-direction.

`vbar` (*vector, default = 0*)
: Specifies the addition of a constant velocity to whatever velocity is returned from the specific kind of velocity generator. *vbar* can be specified for internal variables such as weight and tag, as well as the actual velocity components.

`vsig` (*vector, default = 0*)
: Specifies the addition of a random velocity to whatever velocity is returned from the specific kind of velocity generator. The random velocity is sampled from a Gaussian with standard deviations for each component given by the value of vsig. The seed of the sampling can also be provided with the seed attribute. *vsig* can be specified for internal variables such as weight and tag, as well as the actual velocity components.

kappaVelGen Example Block

```
<VelocityGenerator positronVelGen>
  kind = kappaVelGen
  vbar = [VX_DRIFT  VY_DRIFT  VZ_DRIFT]
  vsig = [VX_RMS  VY_RMS  VZ_RMS]
```

(continues on next page)

(continued from previous page)

```
  seed = VY_SEED

  lowerLimit0 = $ -6. * VX_RMS $
  upperLimit0 = $  6. * VX_RMS $
  lowerLimit1 = $ -6. * VY_RMS $
  upperLimit1 = $  6. * VY_RMS $
  lowerLimit2 = $ -6. * VZ_RMS $
  upperLimit2 = $  6. * VZ_RMS $
</VelocityGenerator>
```

userFuncVelGen

Velocity generator that determines the velocity components based on a user-specified UserFunc (as function of particle position and time, as well as the surface normal for particle emission).

userFuncVelGen Parameters

velocityGenerator (*block*, *required*)

`<UserFunc velocityGenerator>` is a UserFunc that takes arguments (t, x, n) where t is a scalar (the time, in seconds), x is an NDIM-dimensional vector of floats (position), and n is also an NDIM-dimensional vector of floats (surface normal). The UserFunc should return as many floats as the <Species> has internal variables: generally, this is 3 (velocities) plus 1 (if tagged) plus 1 (if variable weight). The return value sets the particle's internal variables.

However, if the last internal variable is particle weight, that can be omitted, in which case, the weight will be determined via the PositionGenerator.

The surface normal is set to zero when a particle is "loaded", but is set to the surface normal when a particle is "emitted."

userFuncVelGen Example Block

```
<VelocityGenerator velgen>
   kind = userFuncVelGen
   <Expression velocityGenerator>
     kind = expression
     # don't forget the particle weight--the last element
     # the tag (penultimate element) will be overwritten by Species,
     #   as we requested
     expression = vector(\
       boostInX(randRelMaxwellSpeed() * rand3dUnitVec(), BETA_DRIFT), 0, 1)
   </Expression>
</VelocityGenerator>
```

3.12 Reactions

3.12.1 Reactions Introduction

The following pages of documentation provide details of setting up interactions between kinetically modeled particle species, background fluids/gases, or any combination of particles and fluids in the "Reactions" framework. For a list of supported interactions see *RxnProductGenerator Sub-Block*. A sample of a complete set of blocks that adds a electron attachment process to a simulation is included at the end of this page. For a user-centric description of this set of blocks see User Guide: Simulation Concepts: Collisions in the user guide.

Anisotropy

For some reactions the option is available to specify the anisotropy of the product distribution. This is specified in the range [-1,1] where 0 is isotropic, -1 results in back scatter, and 1 results in forward scatter. Since anisotropy is a collective behavior, each reaction will not necessarily satisfy this qualitative criteria, but overall the reactions will result in scattering angles as shown in Fig. 3.2.

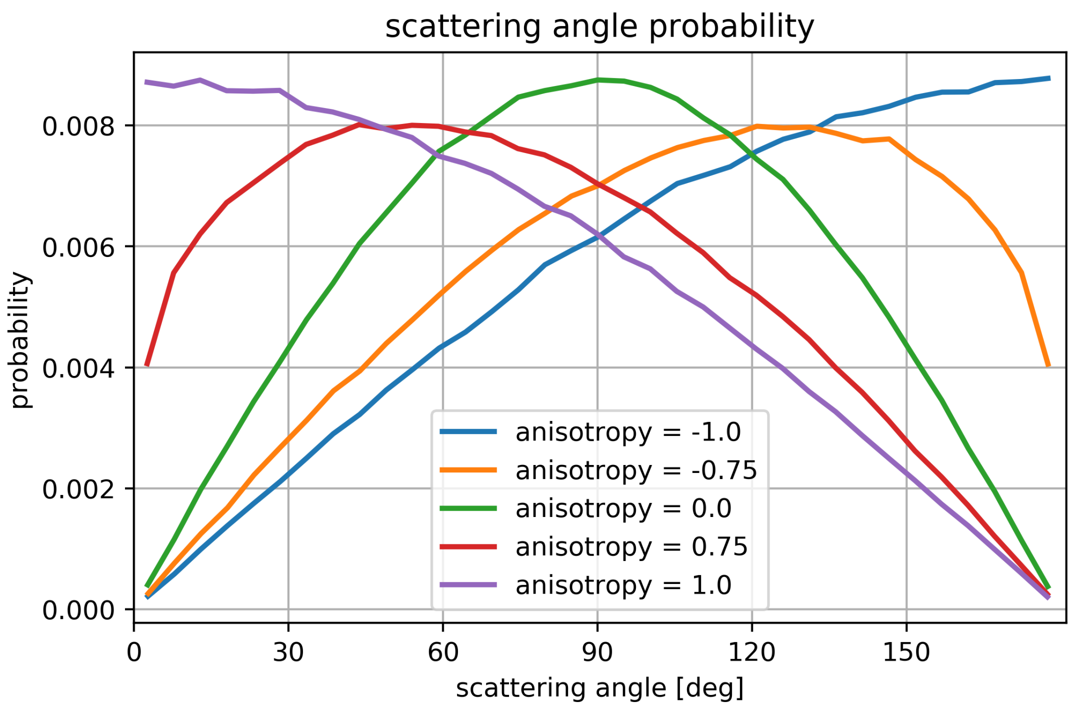


Fig. 3.2: Angular distribution for several choices of `anisotropy`

For a single reaction, the scattering angle is chosen using a random number $r \in [0, 1)$ and the following equation:

$$\theta = 2 \arccos(r^a)$$

where a is

$$a = \frac{(A/|A|)}{|A| - 2}$$

where A is the anisotropy value set by the user in a code block.

If a is negative, we use the absolute value to calculate theta but then reverse the center of momentum velocity of the particles during the calculation of the collision vector.

Reactions Algorithm

Consider a simulation cell with N particles of one species interacting with M particles of another. The total number of combinations of possible reactions is then

$$N_R = NM$$

A set of N_R reaction pairs could then be generated, resulting in a total number of reactions occurring given by

$$N_{occ} = N_R \langle P_{rxn} \rangle = NM \langle P_{rxn} \rangle$$

where P_{rxn} is the individual probability of each reaction pair colliding, which is likely different for each pair. Therefore, the average probability times the total possible reaction pairs gives the total number of reactions that will occur. This requires calculating the reaction probability of NM reaction pairs, which is computationally expensive.

Null Probability

It would be nice to reduce the number of reaction pairs chosen to increase computational efficiency, while maintaining accuracy. The null probability provides the answer here. It is calculated and used as a way to sub-sample the reaction pairs, while proportionally increasing their reaction probability. This should result in the same overall number of reactions as if the full, brute-force method were used. Null probability is defined as

$$P_{null} = \frac{\langle \sigma v \rangle_{max} W_{max} \Delta t}{V}$$

where σ is the cross section of the reaction process, v is the relative velocity of the particles involved, W is the weight of the particle (number of physical particles per macro particle), Δt is the time step, and V is the volume of the simulation cell. This denotes the maximum probability in the cell and is used to normalize the reaction probability such that the reaction pair with the highest probability will then have 100% chance to occur. The number of reaction pairs chosen to potentially collide, then is

$$N_{pairs} = N_R P_{null}$$

Thus, the reaction probability for each chosen reaction pair reaction is

$$P_{adj} = \frac{P_{rxn}}{P_{null}} = \frac{\langle \sigma v \rangle W \Delta t}{V P_{null}} = \frac{\langle \sigma v \rangle}{\langle \sigma v \rangle_{max}}$$

Using this probability along with the number of reaction pairs, we can calculate how many reactions occur.

$$N_{occ} = N_{pairs} \langle P_{adj} \rangle = NMP_{null} \frac{\langle P_{rxn} \rangle}{P_{null}} = NM \langle P_{rxn} \rangle$$

We see that, assuming the subset of particles we chose has the same average reaction probability as the full set, the number of reactions that occur matches exactly what should occur in the full method which is calculated in the first equation for N_{occ} above.

This has the desired effect, but there are some intricacies that need addressing.

Caveats

We have decided to only have each particle interact once per time step, so each particle can only interact with one other particle. This can be accomplished in one of two ways, though only one will give the correct number of reactions.

In the case where N_{pairs} is greater than the lesser of M and N, a single particle *must* appear more than once in the list of reaction pairs. The question then, is what to do in this case.

Remove Duplicates After Reactions

The original implementation involved generating a full list of NMP_{null} pairs, even if there a single particle appears multiple times in this list. Then we go through and determine which reaction pairs actually collide. We then remove any duplicates from the list of collision pairs that actually collide. The theoretical number of reactions that occur should then be correct with $N_{occ} = NM \langle P_{rxn} \rangle$. However, in the case where duplicates are removed, the number of occurring reactions is artificially lowered from this true reaction rate.

Never Include Duplicates

If we instead generate a list of reaction pairs such that there are no duplicates, the number of pairs is given by

$$N_{pairs} = min\left(NMP_{null}, min\left(N, M\right)\right)$$

We will assume that $N \leq M$ from this point on, reducing the above equation to

$$N_{pairs} = min\left(NMP_{null}, N\right)$$

This leaves two cases to explore. The case where $N_{pairs} = NMP_{null}$ has already been described in section 2 as the standard case for the use of null collision probability.

However, if $N_{pairs} = N$, we must then redefine the null probability to have the correct number of reactions occur. This is because the null probability is the ratio of the number of reaction pairs chosen to the total number of possible reaction pairs, as stated in the equation $N_{pairs} = N_R P_{null}$ (see equation above).

$$P_{null} = \frac{N_{pairs}}{N_R} = \frac{N}{NM} = M^{-1}$$

By adjusting the null probability to be the true number of reaction pairs chosen over the total possible, the number of reactions that occurs remains accurate:

$$N_{occ} = \frac{N_{pairs} \langle P_{rxn} \rangle}{P_{null}} = \frac{N \langle P_{rxn} \rangle}{M^{-1}} = NM \langle P_{rxn} \rangle$$

This second method of never including duplicates is thus more accurate than the first where duplicates are removed; however, both have the same limitation. When $N_{occ} > N$ (ie. more reactions should occur than we have particles in the cell), we find that $P_{rxn} > P_{null}$, which can lead to an underestimation of the reaction rate. For such a situation, the time step should be lowered to maintain $N_{occ} \leq N$.

Example of a "Fully-Assembled" Set of Reactions Code Blocks

```
<RxnProcessSettings RxnProcessSettings>
  updateOrder = random
  updatePeriod = [ 1 ]
</RxnProcessSettings>



<RxnProcess electronAttachmentParticlesRXN0>
  kind = collisionProcess
  reactants = [neutralArgon electrons]
  products = [ArMinus]
  rxnPhysics = electronAttachmentParticlesRXN0electronAttachment
  verbosity = 127
</RxnProcess>
```

(continues on next page)

(continued from previous page)

```
<RxnPhysics electronAttachmentParticlesRXN0electronAttachment>
  kind = generalCollision
  <RxnRate rxnRate>
    kind = twoColumnFile
    crossSectionVariable = velocity
    file = 2ColumnData.dat
  </RxnRate>

  <RxnProductGenerator productGenerator>
    kind = electronAttachment
    thresholdEnergy = 1.0
  </RxnProductGenerator>
</RxnPhysics>
```

3.12.2 Reactions Blocks

RxnProcessSettings Block

This optional block is placed before all RxnProcess and RxnPhysics blocks and sets the order the reactions are carried out and how frequently the reaction is updated by VSim. If this block is not included, the default values (indicated below) will be used.

RxnProcess Attributes

updateOrder (*string*, *optional*, *default* = *random*)
Sets the order in which the reactions will be carried out at each timestep.

- **random:** reactions are carried out at a different random order at each timestep.
- **constant:** reactions are always carried out in the order of the `RxnProcess Blocks` in the `.in` file.
- **rotate:** reactions are carried out in the order they are written at the first timestep, then cyclically permuted each subsequent timestep.

updatePeriod (*vector of integers*, *required*, *default: vector of 1's*)
This will set the interval (in timesteps) at which each interaction is updated. At the indicated interval, the reaction will be performed assuming all particles currently in each cell were in the cell since the previous update, using the entire time period since the previous update to determine the reaction probability as to ensure the total number of particle interactions remains constant regardless of the value chosen for the `updatePeriod`.

Advanced users may want to manually set this parameter when running simulations which have a low reaction rate compared to the timestep (due to a small PPC, low cross-section, etc); in such cases a lower sampling rate for a reaction process may be desired. Increasing the update period will result in a small performance boost.

Each position in the vector will match with the position of a *RxnProcess Block* in a `.pre` file.

The `RxnProcessSettings` block below comes from a simulation with 5 `RxnProcess` blocks. The interaction setup in the first, third, and fifth blocks will be carried out every timestep. The process set up in the second `RxnProcess` block will be performed every 5th timestep, and the interaction in the fourth block will be carried out every tenth timestep.

Example RxnProcessSettings Block

```
<RxnProcessSettings RxnProcessSettings>
  updateOrder = random
  updatePeriod = [ 1 5 1 10 1]
</RxnProcessSettings>
```

RxnProcess Block

The `RxnProcess` block sets the reactants and products for a particular reaction. The reactants and products can be any combination of *Fluids* or *Kinetically Modeled Particle Species*.

This block points to a *RxnPhysics Block* which will determine the physics of the reaction, i.e., whether the interaction is an ionization, scattering, dissociation, or an other type of process. See *RxnProductGenerator* for a list of supported processes.

Note: For some processes, the order that the reactants and products are written in the `Reactants` and `Products` lists below is important. Check the documentation for the `ProductGenerator` to determine whether the reactants and products need to be written in a particular order.

RxnProcess Attributes

`kind` (*string*, *required*)
The kind will always be `collisionProcess`

`Reactants` (*list of strings*, *required*)
A list, enclosed by square brackets, of the names of the reacting species. Can be any fluid or kinetically modeled species. Order is important for some reactions. Find the reaction in *RxnProductGenerator* for information on ordering.

`Products` (*list of strings*, *required*)
A list, enclosed by square brackets, of the names of the product species. Can be any fluid or kinetically modeled species. Order is important for some reactions. Find the reaction in *RxnProductGenerator* for information on ordering.

`rxnPhysics` (*string*, *required*)
This string points to a name of a *RxnPhysics Block*. The `RxnPhysics` block determines the physics properties of the `RxnProcess`.

`verbosity` (*int*, *optional*)
A verbosity setting determines the level of output messaging provided by the engine. If no value is provided the top level verbosity will be used (see *Global Variables*). If a value is provided in this block it will overwrite the top level verbosity. A verbosity setting of 127 or lower will show the number of reactions that occur. A setting of 128 or larger will suppress the number of reactions.

`randomSeed` (*int*, *optional*, *default: random int*)
Manually set the random seed used in the selection of reactant particles.

The default is to choose a random seed at run time. To produce identical simulations, advanced users may want to manually set a seed. It is recommended not to include this parameter so that a different random number will be generated for each run.

Example RxnProcess Block

```
<RxnProcess impactElasticPrimaryElectrons>
  kind = collisionProcess
  reactants = [electrons neutralArgon]
  products = [electrons neutralArgon]
  rxnPhysics = impactElasticPrimaryElectronsbinaryElastic
  #verbosity = 127
  #randomSeed = 423
</RxnProcess>
```

RxnPhysics Block

The RxnPhysics Block only contains a kind and two sub-blocks: *RxnRate Sub-Block* and *RxnProductGenerator Sub-Block*. The RxnPhysics block sets the physics of a particular interaction, and is pointed to in the rxnPhysics parameter of a *RxnProcess Block*

kind (*string*, *required*)
The kind will always be `generalCollision`

RxnRate Sub-Block

kind
This block will be used to determine the likelihood of a collision based on a reaction rate or a cross-section. There are several options for setting the reaction rate or cross-section:

- *Constant Cross-Section*
- *Power Law Cross-Section*
- *Exponential Polynomial Cross-Section*
- *Two-Column File Format*
- *Field Ionization DCADK*
- *Field Ionization Average ADK*
- *Decay Rate*

RxnProductGenerator Sub-Block

kind (*string*, *required*)
This will set the physical process. The options for `kind` are:

- *Binary Elastic*
- *Electron Scatter*
- *Binary Excitation*
- *Charge Exchange*
- *Binary Reaction*
- *Electron Attachment*
- *Negative Ion Detachment*

- *Impact Ionization*
- *Electron Ionization*
- *Dissociative Ionization*
- *Field Ionization*
- *Decay*
- *Binary Recombination*
- *Three Body Recombination*
- *Dissociative Recombination*
- *Electron Impact Dissociation*

Example

Below is an example of a complete `RxnPhysics` block

```
<RxnPhysics impactElasticPrimaryElectronsbinaryElastic>
  kind = generalCollision
  <RxnRate rxnRate>
    kind = twoColumnFile
    crossSectionVariable = energy
    file = sampleElasticCrossSection.dat
  </RxnRate>
  <RxnProductGenerator productGenerator>
    kind = binaryElastic
    #anisotropy = 0
    #randomSeed = 423
  </RxnProductGenerator>
</RxnPhysics>
```

3.12.3 Rates and Cross-Sections

Constant Cross-Section

To model a cross-section that does not depend on any other variables, use this `kind`.

constantCrossSection Attributes

`crossSection` (*float, required*)
A constant, in square meters, for the cross section.

Example constantCrossSection Block

```
<RxnRate rxnRate>
  kind = constantCrossSection
  crossSection = 1.25e-20
</RxnRate>
```

Power Law Cross-Section

Use `kind = powerLawCrossSection` to set a cross section according to a power law:

$$\sigma(x) = Ax^E$$

powerLawCrossSection Attributes

coefficient (*float*, *required*)
The coefficient, *A*, in the equation above.

exponent (*float*, *required*)
The exponent, *E*, in the equation above.

variable (*string*, *required*)
Set the dependent variable, x, in the equation above. Options are:

- **velocity** [m/s]
- **energy** [J]

This sets whether vorpal uses the velocity or energy of the reactant particles to calculate the probability of an interaction.

Example powerLawCrossSection Block

```
<RxnRate rxnRate>
  kind = powerLawCrossSection
  coefficient = 1.0e-15
  exponent = -1.0
  variable = velocity
</RxnRate>
```

Exponential Polynomial Cross-Section

This is a fitting formula presented by *[Kim92]* for electron-impact excitation and ionization interactions which fits well from threshold to high energies (<10 keV). For relativistic energies a relativistic formula should be used. Resonant structures are smoothed out. See *[Kim92]* for more details, including suggested values for the fitting constants, *A*, *B*, *C*, *D*, and *I* for some species. In the html version of the paper (https://www.ncbi.nlm.nih.gov/pmc/articles/PMC4909190/) there is a typo in table 1: C should be positive, which can be seen in the original pdf of the paper.

The cross-sections will take the functional from of the equation:

$$\sigma(y) = \frac{1}{y}\left(A\ln(y) + \frac{B\ln(y) + C*(y - "toZero")}{y + D}\right)$$

where

$$y = x/I$$

The `vanishAtThreshold` attribute (see below), sets whether the cross-section goes to a zero or non-zero value at the reaction threshold energy/velocity.

expPolyCrossSection Attributes

A (*float*, *required*)
The value for the fitting constant *A* in the formula above.

B (*float*, *required*)
The value for the fitting constant *B* in the formula above.

C (*float*, *required*)
The value for the fitting constant *C* in the formula above.

D (*float*, *required*)
The value for the fitting constant *D* in the formula above.

I (*float*, *required*)
The value for the fitting constant *I* in the formula above. A scaling factor for the independent variable, x, so must have the same units [m/s or eV]. For reactions with energy/velocity thresholds, this should be that threshold.

toZero (*int*, *required*)
Sets the value for the `toZero` parameter in the formula above. Enter either “1” to have the cross-section go to zero at threshold energy/velocity, or “0” to have the cross-section go to a non-zero value at this threshold.

variable (*string*, *required*)
Set the independent variable, x, in the equation above. Options are:

- **velocity** [m/s]
- **energy** [eV]

This sets whether vorpal uses the velocity or energy of the reactant particles to calculate the probability of an interaction.

Example expPolyCrossSection Block

```
<RxnRate rxnRate>
  kind = expPolyCrossSection
  A = 0.7576
  B = -5.521
  C = 5.867
  D = 2.948
  I = 13.61
  toZero = 1
  variable = energy
</RxnRate>
```

Two-Column File Format

Use this `kind` for the `RxnRate` sub-block to set the cross section using a two-column data file. The data file should have no header.

The units for the dependent variable (first column in the data file) must be either m/s (meters per second) or eV (electron volts). The user sets which with the `crossSectionVariable` parameter.

The units for the cross section (second column) **must be** in square meters (not barns). **Beware, that the eedl dataset is in units of MeV and barns** and so requires conversion: 1 MeV = 1.e6 eV, and 1 barn = 1.e-28 square meters.

The LxCat database (https://fr.lxcat.net/data/set_type.php) contains a wide range of cross sections which can be imported into VSim.

Note: headers must be removed, and units must be correct before importing into VSim.

twoColumnFile Attributes

`crossSectionVariable` (*string*, *required*)
The unit of the values in the first column of the two column data file. There are two options:

- **`energy`**
- **`velocity`**

`file` (*string*, *required*)
The name of the data file containing the cross section as a function of the `crossSectionVariable`

Example twoColumnFile Block

```
<RxnRate rxnRate>
  kind = twoColumnFile
  crossSectionVariable = energy
  file = ArArIonization.dat
</RxnRate>
```

Field Ionization DCADK

This cross-section models the tunneling ionization of a kinetically modeled species or background gas induced by an electric field. The "DCADK" should be used when the time step taken in the simulation resolves the oscillations of the fields.

The following formula for the ionization rate is used:

$$R_i = 4.13 \times 10^{16} \frac{Z^2}{2n_{\text{eff}}^2} \left(\frac{2e}{n_{\text{eff}}}\right)^{2n_{\text{eff}}} \frac{1}{2\pi n_{\text{eff}}} \left(2\frac{E_h}{E_L}\frac{Z^3}{n_{\text{eff}}^3}\right)^{2n_{\text{eff}}-1} \exp\left[-\frac{2}{3}\frac{E_h}{E_L}\frac{Z^3}{n_{\text{eff}}^3}\right] (\text{s}^{-1})$$

where Z is the charge state of the ionized particle, $n_{\text{eff}} = Z/\sqrt{U_{\text{ion}}/13.6[\text{eV}]}$, U_{ion} is the ionization potential in eV, $E_h = m_e^2 q_e^5/(4\pi\epsilon_0\hbar^4) = 5.13 \times 10^{11}\text{V/m}$, and E_L is the electric field strength at the particle position.

The modified ADK *[ADK86]* formula, updated in *[DK91]* and first adapted to PIC by Penetrante and Bardsley *[PB91]* and later corrected by Ilkov et al. *[IDC92]*, which gives the ionization rate for any type of atom, can also be used.

Subsequent validation work on the tunneling ionization models in VSim was conducted and is demonstrated in *[CSMZ06]*, *[CES+12]* and [chen2013numerica]. The model is valid up to approximately energy densities of $10^{23} - 10^{24}$ above which Barrier Suppression Ionization is likely to be the dominant effect, and one way also want to consider vacuum pair-production in the ionization cross-section.

The model assumes the background is a gas, that is to say that atoms are well separated with respect to their size. At very high number density, the model may break down. See references to the Mott Transition for further information.

Note: Must be paired with a *Field Ionization ProductGenerator* in a `RxnProcess` Block

fieldIonizationDCADK Attributes

kind (*string*, *required*)
Use the string `fieldIonizationDCADK`

ionizationEnergy (*float*, *optional*)
The ionization energy of the species to be ionized. The species that will be ionized by this process is determined by the `Reactants` parameter in the *RxnProcess Block*. By default, vorpal will take the ionization energy from the block defining the ionized species or fluid. The value provided in this block will overwrite the default value.

charge (*float*, *optional*)
The charge of the species to be ionized. The species that will be ionized by this process is determined by the `Reactants` parameter in the *RxnProcess Block*. By default, vorpal will take the `charge` from the block defining the ionized species or fluid. A value provided in this block will overwrite the default value.

Example fieldIonizationDCADK Block

```
<RxnRate rxnRate>
  kind = fieldIonizationDCADK
  #ionizationEnergy = 4.0
  #charge = 2.0
</RxnRate>
```

Field Ionization Average ADK

This cross-section models the tunneling ionization of a kinetically modeled species or background gas induced by an electric field. The “Average ADK” uses a time-averaged formula (see below) for the ionization rate, and should only be used when modeling a linearly polarized electric field and the time step is larger than the oscillation period (i.e., dt is much larger than the period of the field in question). This might occur, for example, when using an envelope model for the laser pulse.

$$R_i = 6.6 \times 10^{16} \frac{e}{\pi} \frac{Z^2}{n_{\rm eff}^{4.5}} \left(10.87 \times \frac{E_h}{E_L} \frac{Z^3}{n_{\rm eff}^4}\right)^{2n_{\rm eff}-1.5} \exp\left[-\frac{2}{3}\frac{E_h}{E_L}\frac{Z^3}{n_{\rm eff}^3}\right] ({\rm s}^{-1})$$

where Z is the charge state of the ionized particle, $n_{\rm eff} = Z/\sqrt{U_{\rm ion}/13.6[{\rm eV}]}$, $U_{\rm ion}$ is the ionization potential in eV, $E_h = m_e^2 q_e^5/(4\pi\epsilon_0\hbar^4) = 5.13 \times 10^{11}{\rm V/m}$, and E_L is the electric field strength at the particle position.

The modified ADK *[ADK86]* (Ammosov, Delone, and Krainov) formula, updated in *[DK91]* and first adapted to PIC by Penetrante and Bardsley *[PB91]* and later corrected by Ilkov et al. *[IDC92]*, which gives the ionization rate for any type of atom, can also be used.

Subsequent validation work on the tunneling ionization models in VSim was conducted and is demonstrated in *[CSMZ06]*, *[CES+12]* and *[CCMG+13]*. The model is valid up to approximately energy densities of $10^{23} - 10^{24}$ above which Barrier Suppression Ionization is likely to be the dominant effect, and one way also want to consider vacuum pair-production in the ionization cross-section.

The model assumes the background is a gas, that is to say that atoms are well separated with respect to their size. At very high number density, the model may break down. See references to the Mott Transition for further information.

Note: Must be paired with a *Field Ionization ProductGenerator* in a `RxnProcess` Block.

fieldIonizationAverageADK Attributes

ionizationEnergy (*float*, *optional*)
: The ionization energy of the species to be ionized. The species that will be ionized by this process is determined by the `Reactants` parameter in the *RxnProcess Block*. By default, vorpal will take the ionization energy from the block defining the ionized species or fluid. The value provided in this block will overwrite the default value.

charge (*float*, *optional*)
: The charge of the species to be ionized. The species that will be ionized by this process is determined by the `Reactants` parameter in the *RxnProcess Block*. By default, vorpal will take the `charge` from the block defining the ionized species or fluid. A value provided in this block will overwrite the default value.

Example fieldIonizationAverageADK Block

```
<RxnRate rxnRate>
  kind = fieldIonizationAverageADK
  #ionizationEnergy = 4.0
  #charge = 0.0
</RxnRate>
```

Decay Rate

The `RxnRate` block to be used to set the lifetime of a species under going a decay. Must be paired with a *Decay* ProductGenerator.

decayLifetime Attributes

lifetime (*float*, *required*)
: the lifetime of the species, in seconds

Example decayLifetime Block

```
<RxnRate rxnRate>
  kind = decayLifetime
  lifetime = 0.0
</RxnRate>
```

3.12.4 Interactions (productGenerators)

Binary Elastic

Works with a VSimPD license. Collision of the form

$$A + B \to A + B$$

Scattering of two particles with a variable angular distribution (set with the `anisotropy` parameter below). Momentum is conserved, and energy is conserved if the `energyLoss` parameter is zero.

The `binaryElastic` can be used to model inelastic collisions by setting the `energyLoss` parameter to be non-zero.

Any *RxnProcess Block* block which points to a productGenerator of kind = `binaryElastic` should have the `Reactants` and `Products` in the order:

```
reactants = [speciesA speciesB]
products = [speciesA speciesB]
```

Note: The order is arbitrary, but the must be consistent between the `Reactants` and `Products` blocks.

Note: The *Electron Scatter* interaction is optimized for an inelastic scattering process involving an electron.

binaryElastic Attributes

`energyLoss` (*float*, *optional*, *default = 0*)
The maximum energy, in eV, lost during a single inelastic collision. A choice of 0 will give an elastic collision.

`anisotropy` (*float*, *optional*, *default = 0*)
A value between -1 and 1 to set the degree of anisotropy. An anisotropy of -1 is full backscatter, 0 is isotropic, and +1 is full forward scatter.

`randomSeed` (*int*, *optional*, *default: random int*)
Manually set a random seed used to determine the final velocities of the product particles.

The default is to choose a random seed at run time. To produce identical simulations, advanced users may want to manually set a seed. It is recommended not to include this parameter so that a different random number will be generated for each run.

Example binaryElastic Block

```
<RxnProductGenerator productGenerator>
  kind = binaryElastic
  #energyLoss = .2
  #anisotropy = 0
  #randomSeed = 423
</RxnProductGenerator>
```

Electron Scatter

Works with a VSimPD license. Collision of the form

$$e + A \rightarrow e + A$$

A scattering process similar to `binaryElastic`, but optimized for the specific case where one of the scattering species is an electron. The `electronScatter` process is inelastic, for an elastic electron scattering process, use *Binary Elastic*.

A generator of 2 product particles to model electron scattering off of a neutral/ion. Can be isotropic or anisotropic following the Vahedi-Surendra algorithm *[VS95]*.

In the Vahedi-Surendra algorithm, the incident electron loses small percentage of its energy and is scattered after the collision. The scattered electron energy, ϵ_{sc}, is computed to be:

$$\epsilon_{sc} = \epsilon_{inc}\left(1 - \frac{2m_e M_0}{(m_e + M_0)^2}(1. - \cos\chi)\right)$$

where:

- ϵ_{inc} is the incident electron energy,
- m_e is the mass of the incident electron,
- M_0 is the mass of the "A" species in the equation above,
- χ is the scattering angle given by:

$$\cos\chi = \frac{2 + \epsilon_{inc} - 2\left(1 + \epsilon_{inc}\right)^{R_1}}{\epsilon_{inc}}$$

The azimuthal scattering angle is given by:

$$\varphi = 2\pi R_2$$

In the above equations R_1 and R_2 are two random numbers uniformly distributed between 0 and 1.

Any *RxnProcess Block* block which points to a productGenerator of kind = `electronScatter` should have the `Reactants` and `Products` in the order:

```
reactants = [electron speciesA]
products = [electron speciesA]
```

electronScatter Attributes

scatterType (*string*, *required*, *default = VS*)
The isotropy of the electron scatter. Can be `isotropic` or `VS` for a non-isotropic scattering following the algorithm presented by Vahedi and Surendra (see above).

randomSeed (*int*, *optional*, *default: random int*)
Manually set a random seed used to determine the final velocities of the product particles.

The default is to choose a random seed at run time. To produce identical simulations, advanced users may want to manually set a seed. It is recommended not to include this parameter so that a different random number will be generated for each run.

Example electronScatter Block

```
<RxnProductGenerator productGenerator>
  kind = electronScatter
  #scatterType = VS
  #randomSeed = 423
</RxnProductGenerator>
```

Binary Excitation

Works with a VSimPD license. Collision of the form:

$$A + B \rightarrow A^* + B$$

A generator of 2 product particles with an angular distribution set by the `anisotropy` attribute. Momentum is conserved and energy is conserved minus the `excitationEnergy` attribute.

Any *RxnProcess Block* block which points to a productGenerator of kind = `binaryExcitation` should have the `Reactants` and `Products` in the order:

```
reactants = [speciesA speciesB]
products = [excitedSpeciesA speciesB]
```

binaryExcitation Attributes

excitationEnergy (*float*, *required*)
The potential energy, in eV, gained by the excited species (a positive value for this attribute will result in an energy loss from the simulation). This is also a threshold energy. The reaction will not occur between reactant that have a center of mass energy less than this value.

anisotropy (*float*, *optional*, *default = 0*)
A value between -1 and 1 to set the degree of anisotropy. An anisotropy of -1 is full backscatter, 0 is isotropic, and +1 is full forward scatter.

randomSeed (*int*, *optional*, *default: random int*)
Manually set a random seed used to determine the final velocities of the product particles.

The default is to choose a random seed at run time. To produce identical simulations, advanced users may want to manually set a seed. It is recommended not to include this parameter so that a different random number will be generated for each run.

Example binaryExcitation Block

```
<RxnProductGenerator productGenerator>
  kind = binaryExcitation
  excitationEnergy = 1.1828
  #anisotropy = 0
  #randomSeed = 423
</RxnProductGenerator>
```

Charge Exchange

Works with a VSimPD license. Collision of the form

$$A + B^+ \to A^+ + B$$

This process models the charge exchange between two particles, so "A" and "B" in the formula above can be the same or different species. Momentum is conserved, and energy conserved less the `energyLoss` attribute which is set by the user.

Any *RxnProcess Block* block which points to a productGenerator of kind = `chargeExchange` should have the `Reactants` and `Products` in the order:

```
reactants = [Atom Ion]
products = [Ion Atom]
```

chargeExchange Attributes

energyLoss (*float*, *optional*, *default = 0*)
The maximum energy lost during a single inelastic collision in eV. A choice of 0 (default) will give an elastic collision.

anisotropy (*float*, *optional*, *default = 0*)
A value between -1 and 1 to set the degree of anisotropy. An anisotropy of -1 is full backscatter, 0 is isotropic, and +1 is full forward scatter.

randomSeed (*int*, *optional*, *default: random int*)
Manually set a random seed used to determine the final velocities of the product particles.

The default is to choose a random seed at run time. To produce identical simulations, advanced users may want to manually set a seed. It is recommended not to include this parameter so that a different random number will be generated for each run.

Example chargeExchange Block

```
<RxnProductGenerator productGenerator>
  kind = chargeExchange
  #energyLoss = 0.0
  #anisotropy = 0.0
  #randomSeed = 789
</RxnProductGenerator>
```

Binary Reaction

Works with a VSimPD license. Collision of the form

$$A + B \rightarrow C + D$$

This is a general reaction where the A, B, C, and D species can each be any fluid or particle species. For example, a user could set up a an inelastic scattering collision between argon ions of the same species:

$$Ar^{+} + Ar^{+} \rightarrow Ar^{+} + Ar^{+}$$

Or, a user could use this productGenerator to set up a collision between up to four different fluids and particles like the following reaction which creates an an exotic hydrogen molecule:

$$H_2 + H_2^{+} \rightarrow H_3^{+} + H$$

A generator of 2 product particles with an isotropic angular distribution, momentum conserved and energy conserved minus the `thresholdEnergy` parameter (which could be the energy required for the reaction).

binaryReaction Attributes

thresholdEnergy (*float*, *optional*, *default = 0*)
The threshold energy for the reaction in eV. So, a pair of reactants will need at least this much relative energy (i.e. energy in center of momentum frame) in order to react. If the reaction occurs then this much energy is lost from the products to potential energy. If this is zero, then energy will not be lost. If negative, then the products will gain kinetic energy.

anisotropy (*float*, *optional*, *default = 0*)
A value between -1 and 1 to set the degree of anisotropy. An anisotropy of -1 is full backscatter, 0 is isotropic, and +1 is full forward scatter.

randomSeed (*int*, *optional*, *default: random int*)
Manually set a random seed used to determine the final velocities of the product particles.

The default is to choose a random seed at run time. To produce identical simulations, advanced users may want to manually set a seed. It is recommended not to include this parameter so that a different random number will be generated for each run.

Example binaryReaction Block

```
<RxnProductGenerator productGenerator>
  kind = binaryReaction
  #thresholdEnergy = 1.0
  #anisotropy = 0.5
  #randomSeed = 324
</RxnProductGenerator>
```

Electron Attachment

Works with a VSimPD license. Collision of the form

$$A + e \rightarrow A^-$$

An electron attachment generator of one product negative ion. This process conserves momentum, but is inelastic (does not conserve energy).

Any *RxnProcess Block* block which points to a productGenerator of kind = `electronAttachment` should have the `Reactants` and `Products` in the order

```
reactants = [Atom electron]
products = [negIon]
```

electronAttachment Attributes

thresholdEnergy (*float*, *required*)
The threshold energy for the reaction in eV. So, a pair of reactants will need at least this much relative energy (ie energy in center of momentum frame) in order to react. If the reaction occurs then this much energy is lost.

randomSeed (*int*, *optional*, *default: random int*)
Manually set a random seed used to determine the final velocities of the product particles.

The default is to choose a random seed at run time. To produce identical simulations, advanced users may want to manually set a seed. It is recommended not to include this parameter so that a different random number will be generated for each run.

Example electronAttachment Block

```
<RxnProductGenerator productGenerator>
  kind = electronAttachment
  thresholdEnergy = 1.0
  #randomSeed = 753
</RxnProductGenerator>
```

Negative Ion Detachment

Works with a VSimPD license. Collision of the form

$$A^- + B \to A + B + e$$

Any *RxnProcess Block* block which points to a productGenerator of kind = `negativeIonDetachment` should have the `Reactants` and `Products` in the order

```
reactants = [speciesNegativeA speciesB]
products = [speciesA speciesB electron]
```

negativeIonDetachment Attributes

thresholdEnergy (*float*, *optional*, *default = 0*)
The threshold energy for the reaction in eV. So, a pair of reactants will need at least this much relative energy (ie energy in center of momentum frame) in order to react. If the reaction occurs then this much energy is lost.

anisotropy (*float*, *optional*, *default = 0*)
A value between -1 and 1 to set the degree of anisotropy. An anisotropy of -1 is full backscatter, 0 is isotropic, and +1 is full forward scatter.

randomSeed (*int*, *optional*, *default: random int*)
Manually set a random seed used to determine the final velocities of the product particles.

The default is to choose a random seed at run time. To produce identical simulations, advanced users may want to manually set a seed. It is recommended not to include this parameter so that a different random number will be generated for each run.

Example negativeIonDetachment Block

```
<RxnProductGenerator productGenerator>
  kind = negativeIonDetachment
  #thresholdEnergy = 1.0
  #anisotropy = 0.0
  #randomSeed = 477
</RxnProductGenerator>
```

Impact Ionization

Works with a VSimPD license. Collision of the form

$$A + B \to A^+ + B + e$$

Any *RxnProcess Block* block which points to a productGenerator of kind = `impactIonization` should have the `Reactants` and `Products` in the order

```
reactants = [speciesLosingElectron scatteredSpecies]
products = [ionizedSpecies scatteredSpecies Electron]
```

impactIonization Attributes

ionizationEnergy (*float*, *optional*)
The ionization energy of the neutral species. By default, the value of this parameter is taken from the species block of the first reactant in the `RxnProcess` block and does not need to be included.

anisotropy (*float*, *optional*, *default = 0*)
A value between -1 and 1 to set the degree of anisotropy. An anisotropy of -1 is full backscatter, 0 is isotropic, and +1 is full forward scatter.

randomSeed (*int*, *optional*, *default: random int*)
Manually set a random seed used to determine the final velocities of the product particles.

The default is to choose a random seed at run time. To produce identical simulations, advanced users may want to manually set a seed. It is recommended not to include this parameter so that a different random number will be generated for each run.

Example impactIonization Block

```
<RxnProductGenerator productGenerator>
  kind = impactIonization
  #ionizationEnergy = 1.45
  #anisotropy = 0
  #randomSeed = 423
</RxnProductGenerator>
```

Electron Ionization

Works with a VSimPD license. Collision of the form

$$A + e \rightarrow A^+ + e + e$$

An ionization process similar to *Impact Ionization*, but optimized for the specific case where an electron ionizes a second species.

In this collision, the energy of the scattered electron (that is, the final energy of the incident electron) is determined from:

$$\epsilon_{sc} = \epsilon_{inc} - \epsilon_{iz} - \epsilon_{se}$$

where:

- ϵ_{inc} the relative energy of the two reactants in center of mass frame
- ϵ_{iz} the ionization energy threshold
- ϵ_{se} the secondary electron energy which is given by:

$$\epsilon_{se} = \omega \tan\left[R_3 \tan^{-1}\left(\frac{\epsilon_{inc} - \epsilon_{iz}}{2\omega}\right)\right]$$

where ω is a constant with a value of 15 eV and R_3 is a random number with a uniform distribution between 0 and 1.

The trajectory of the scattered electron after the collision is determined by the scattering angle χ such that:

$$\cos\chi = \left(\frac{\epsilon_{sc}}{\epsilon_{inc} - \epsilon_{iz}}\right)^{0.5}$$

and the azimuthal angle $\varphi = 2\pi R_1$, where R_1 is a random number uniformly distributed between 0 and 1.

The scattering angle of the secondary electron (created in the interaction) is determined using:

$$\cos\chi_{sc} = \left(\frac{\epsilon_{se}}{\epsilon_{inc} - \epsilon_{iz}}\right)^{0.5}$$

and the azimuthal angle is $\varphi_{se} = 2\pi R_2$, where R_2 is a random number uniformly distributed between 0 and 1.

The Visual Setup will automatically use this `ProductGenerator` for an ionization interaction if an electron species is included in the reactants.

Any *RxnProcess Block* block which points to a productGenerator of kind = `electonIonization` should have the `Reactants` and `Products` in the order

```
reactants = [Atom Electron]
products = [Ion Electron secElectron]
```

electronIonization Attributes

`ionizationEnergy` (*float*, *optional*)
The ionization energy, in eV, of the neutral species. By default, the value of this parameter is taken from the species block of the first reactant in the `RxnProcess` block.

`randomSeed` (*int*, *optional*, *default: random int*)
Manually set a random seed used to determine the final velocities of the product particles.

The default is to choose a random seed at run time. To produce identical simulations, advanced users may want to manually set a seed. It is recommended not to include this parameter so that a different random number will be generated for each run.

Example electronIonization Block

```
<RxnProductGenerator productGenerator>
  kind = electronIonization
  #ionizationEnergy = 1.47
  #randomSeed = 789
</RxnProductGenerator>
```

Dissociative Ionization

Works with a VSimPD license. Collision of the form

$$AB + e \rightarrow A^+ + B + e + e$$

or

$$AB + e \rightarrow A^+ + B^+ + e + e + e$$

A generator of 4 product particles, two atoms that have been dissociated and two electrons: one that impacted to cause the ionization and other from the ionization caused the dissociation. Double ionization is achieved by including a third electron product and a second ionized atom in a the Reactants and Product vectors in the *RxnProcess Block*

Any *RxnProcess Block* block which points to a productGenerator of kind = `dissociativeIonization` should have the `Reactants` and `Products` in the order

```
reactants = [AB electron]
products = [ionizedA (ionized)B electron electron (electron)]
```

dissociativeIonization Attributes

`dissociationEnergy` (*float*, *required*)
The energy required to dissociate the molecule, in eV. This value also sets a threshold for whether or not the reaction will occur. So a pair of particle will need at least this much relative energy (i.e. energy in center of momentum frame) in order to react. This can be set as a negative value to have the products gain kinetic energy.

`randomSeed` (*int*, *optional*, *default: random int*)
Manually set a random seed used to determine the final velocities of the product particles.

The default is to choose a random seed at run time. To produce identical simulations, advanced users may want to manually set a seed. It is recommended not to include this parameter so that a different random number will be generated for each run.

Example dissociativeIonization Block

```
<RxnProductGenerator productGenerator>
  kind = dissociativeIonization
  dissociationEnergy = 1.0
  #randomSeed = 499
</RxnProductGenerator>
```

Field Ionization

Works with a VSimPA license. Interaction of the form

$$A + \vec{E} \rightarrow e + A^+$$

Any *RxnProcess Block* block which points to a productGenerator of kind = `fieldIonization` should have the `Reactants` and `Products` in the order:

```
reactants = [speciesToBeIonized multiFieldName.ElectricFieldName]
products = [Ion electron]
```

This product generator can be linked to a `RxnRate` block that determines the ionization rate for a field that time-resolved by the simulation time step, *Field Ionization DCADK*, or a block that determines the ionization rate when the fields are NOT resloved by the time step, *Field Ionization Average ADK*. This product generator should not be linked to any other type of `RxnRate` block.

Note: This process does not deplete EM field energy.

fieldIonization Attributes

randomSeed (*int*, *optional*, *default: random int*)
Manually set a random seed used to determine the final velocities of the product particles.

The default is to choose a random seed at run time. To produce identical simulations, advanced users may want to manually set a seed. It is recommended not to include this parameter so that a different random number will be generated for each run.

Example fieldIonization Block

```
<RxnProductGenerator productGenerator>
  kind = fieldIonization
  #randomSeed = 423
</RxnProductGenerator>
```

Decay

Works with a VSimPD license. Interaction of the form:

$$A \rightarrow B\ (+C + \gamma)$$

A decay of a single particle into one or more products. The daughter particles can be a single species, a single species and a photon, two species (without a photon), or two species and with a photon.

Should be paired with a *decayLifetime* `RxnPhysics` block in a `RxnProcess` Block.

Any *RxnProcess Block* block which points to a productGenerator of kind = `decay` should have the `Reactants` and `Products` in the order:

```
reactants = [speciesA]
products = [speciesB (speciesC)]
```

decay Attributes

untrackedPhoton (*boolean*, *required*)
Whether or not energy is lost to an untracked photon.

VAtheory (*boolean*, *optional*, *default = 0*)
If there is an untracked photon, this flag determines how the lost energy is calculated. Options are:

- **1** Use VA theory to determine energy loss
- **0**: The lost energy is uniformly distributed between none and all.

This parameter does nothing if `untrackedPhoton` = 0.

randomSeed (*int*, *optional*, *default: random int*)
Manually set a random seed used to determine the final velocities of the product particles.

The default is to choose a random seed at run time. To produce identical simulations, advanced users may want to manually set a seed. It is recommended not to include this parameter so that a different random number will be generated for each run.

Example decay Block

```
<RxnProductGenerator productGenerator>
  kind = decay
  untrackedPhoton = 1
  VAtheory = 1 #using VA theory to determine energy lost in decay
  #randomSeed = 423
</RxnProductGenerator>
```

Binary Recombination

Works with a VSimPD license. Collision of the form

$$A + B \rightarrow AB$$

This reaction will make one product particle out of two reactants. This process conserves momentum but is inelastic, with the energy loss going into an un-modeled photon. Supports either electron-ion recombination or something more exotic like

$$H^- + H^+ \rightarrow H_2$$

binaryRecombination Attributes

randomSeed (*int*, *optional*, *default: random int*)
Manually set a random seed used to determine the final velocities of the product particles.

The default is to choose a random seed at run time. To produce identical simulations, advanced users may want to manually set a seed. It is recommended not to include this parameter so that a different random number will be generated for each run.

Example binaryRecombination Block

```
<RxnProductGenerator productGenerator>
  kind = binaryRecombination
  #randomSeed = 456
</RxnProductGenerator>
```

Three Body Recombination

Works with a VSimPD license. Collision of the form

$$A^+ + e + B \rightarrow A + B$$

A three-body recombination generator with three reactants and two products.

Generally the bystander particle, B, in the formula above is an electron, but can be a particle of any mass or charge.

Any *RxnProcess Block* block which points to a productGenerator of kind = `threeBodyRecombination` should have the `Reactants` and `Products` in the order:

```
reactants = [ion electron elec/atom/ion]
products = [neutralSpecies elec/atom/ion]
```

threeBodyRecombination Attributes

recombinationEnergy (*float*, *optional*)
: The recombination energy in eV. By default, this value is taken from the `ionizationEnergy` attribute of the species block of the product species "A" in the reaction above, unless overwritten here. This energy will be given to the bystander species (B in the formula above).

randomSeed (*int*, *optional*, *default: random int*)
: Manually set a random seed used to determine the final velocities of the product particles.

  The default is to choose a random seed at run time. To produce identical simulations, advanced users may want to manually set a seed. It is recommended not to include this parameter so that a different random number will be generated for each run.

Example threeBodyRecombination Block

```
<RxnProductGenerator productGenerator>
  kind = threeBodyRecombination
  # recombinationEnergy = 2.33
  # randomSeed = 423
</RxnProductGenerator>
```

Dissociative Recombination

Works with a VSimPD license. Collision of the form

$$AB^+ + e \rightarrow A + B.$$

Any *RxnProcess Block* block which points to a productGenerator of kind = `dissociativeRecombination` should have the `Reactants` in the order:

```
reactants = [speciesABplus electron]
```

Note: The second reactant is not required to be an electron. The order of the products can be arbitrary.

dissociativeRecombination Attributes

thresholdEnergy (*float*, *required*)
: The threshold energy for the reaction in eV. So, a pair of reactants will need at least this much relative energy (i.e. energy in center of momentum frame) in order to react. If the reaction occurs then this much energy is lost from the products to potential energy. If this is zero, then energy will not be lost. If negative, then the products will gain kinetic energy.

anisotropy (*float*, *optional*, *default = 0*)
: A value between -1 and 1 to set the degree of anisotropy. An anisotropy of -1 is full backscatter, 0 is isotropic, and +1 is full forward scatter.

randomSeed (*int*, *optional*, *default: random int*)
Manually set a random seed used to determine the final velocities of the product particles.

The default is to choose a random seed at run time. To produce identical simulations, advanced users may want to manually set a seed. It is recommended not to include this parameter so that a different random number will be generated for each run.

Example dissociativeRecombination Block

```
<RxnProductGenerator productGenerator>
  kind = dissociativeRecombination
  thresholdEnergy = 1.0
  #anisotropy = 0
  #randomSeed = 423
</RxnProductGenerator>
```

Electron Impact Dissociation

Works with a VSimPD license. An interaction of the form

$$e + AB \rightarrow A + B + e$$

This reaction models an electron induced dissociation process. Three product particles are created.

The final speeds of each particle are determined by calculating the final total energy in the center of mass frame (i.e. total final energy = total initial energy - dissociationEnergy). Then a loss fraction is calculated, f = finalEnergy/initialEnergy. The energy of the AB particle is then multiplied by f and split between A and B. The initial energy of the electron is multiplied by f to get the final electron energy. Two random angles are chosen to determine the direction for the final momentum. One random angle sets the direction for the two atoms (the products' final momentum will be equal and opposite, i.e., back to back). A second random angle sets the direction of the product electron. Finally,the particles' velocities are transformed back to the simulation frame.

Any *RxnProcess Block* block which points to a productGenerator of kind = `electronImpactDissociation` should have the `Reactants` and `Products` in the order:

```
reactants = [electron molecule]
products = [atom1 atom2 electron]
```

electronImpactDissociation Attributes

dissociationEnergy (*float*, *required*)
The energy required to dissociate the molecule in eV.

randomSeed (*int*, *optional*, *default: random int*)
Manually set a random seed used to determine the final velocities of the product particles.

The default is to choose a random seed at run time. To produce identical simulations, advanced users may want to manually set a seed. It is recommended not to include this parameter so that a different random number will be generated for each run.

Example electronImpactDissociation Block

```
<RxnProductGenerator productGenerator>
  kind = electronImpactDissociation
  dissociationEnergy = 4.77
  #randomSeed = 754
</RxnProductGenerator>
```

3.13 ImpactCollider / ImpactCollision

3.13.1 ImpactCollider Block

ImpactCollider

> An ImpactCollider block is used to model charged particle collisions with fluid or background neutral gases, or fluid gas mixtures.

The ImpactCollider/ImpactCollisions feature handles electron and ion impact collisions with background fluid neutral gas or fluid gas mixtures. ImpactCollider/ImpactCollisions uses the Monte Carlo Collision (MCC) technique. It enables users to test a variety of charged particle impact collisions in plasma simulations with background gases or gas mixtures. Details of the collision models and the collision cross-section options available are described below.

This feature allows modeling of charged particle collisions with either one gas kind or multiple gas mixtures. Also, ImpactCollider allows users the flexibility to consider only one collision kind, more than one collision kind, or all collision kinds in their VSim simulations.

In the case of electron-impact collisions, both elastic scattering and inelastic collisions such as excitation or ionization processes are supported. For the ion-impact collisions case, both elastic scattering (momentum-exchange) and inelastic charge-exchange collision processes are supported. In addition, ImpactCollider enables users to select impact collision cross-section data from three different options.

Available options for impact cross-section selection data include:

- builtIn
- eedl
- userDefined

These cross-section options are described in detail in *ImpactCollision* blocks.

Users can use one or multiple ImpactCollider blocks in a VSim input file. Each ImpactCollider block can contain one or multiple ImpactCollision blocks to define different kinds of electron/ion impact collisions with the fluid neutral gases in the corresponding VSim input file.

Note: ImpactCollider blocks should be used only with background fluid gases (i.e., the neutral gas is assumed to be a fluid rather than a collection of kinetic neutral macroparticles). The gases listed under the ImpactCollider block's neutralGas parameter must be defined using the Fluid block, and the Fluid block's gasKind should match the gas names declared in the neutralGas parameter of the ImpactCollider block.

ImpactCollider Parameters

There is only one kind of ImpactCollider block, and therefore no kind attribute is needed.

The ImpactCollider block can contain one or more *ImpactCollision* blocks to specify which collisions will take place.

`neutralGas` (string vector) List of fluid gas or fluid gas mixture VSim objects that can be collided with the impacting particle species. VSim checks to see if a neutralGas vector of strings is provided. The strings correspond to the names of the ionizable objects. The names of the ionizable gas objects must be defined in the VSim input file using Fluid blocks.

`neutralGasTemp` (float vector, optional) Temperature values for the gases listed under the neutralGas. The default gas temperature value is 300 K. The neutralGasTemp parameter is used in determining the energies of ions produced in ionization reactions, and also in determining the ion energies during charge-exchange collisions.

`impactSpecies` (string vector) A list of particle species that can participate in impact collisions with the gas or gas mixtures listed under the neutralGas. VSim checks whether an impactSpecies vector of strings is provided. The strings should correspond to the names of the impact species objects, which must be defined in the VSim input file using Species blocks. Both fixed-weight and variable-weight particles are supported.

Example ImpactCollider Block

```
<ImpactCollider electronHeGasCollisions>

  neutralGas = [HeNeutralGas]
  impactSpecies = [electrons]
  neutralGasTemp = GAS_TEMP_IN_K
  <ImpactCollision ElecImpElastic>
    kind = impactElastic
    crossSectionDataType = [userDefined]
    crossSectionDataFile = [elasticCS.dat]
    scatteringType = uniform
  </ImpactCollision>

  <ImpactCollision ElecImpExcitation1>
    kind = impactExcitation
    crossSectionDataType = [userDefined]
    crossSectionDataFile = [excitation1CS.dat]
    scatteringType = uniform
  </ImpactCollision>

  <ImpactCollision ElecImpExcitation2>
    kind = impactExcitation
    crossSectionDataType = [userDefined]
    crossSectionDataFile = [excitation2CS.dat]
    scatteringType = uniform
  </ImpactCollision>

  <ImpactCollision elecImpIonization>
    kind = impactIonization
    crossSectionDataType = [userDefined]
    crossSectionDataFile = [ionizationCS.dat]
    ionizedElectronSpecies = [electrons]
    ionSpecies = [He1]
    depleteBackgroundGas = false
    scatteringType = uniform
  </ImpactCollision>

</ImpactCollider>
```

3.13.2 ImpactCollision Block

ImpactCollision

`ImpactCollision`:

ImpactCollider blocks require a block for each type of ImpactCollision. There are four possible kinds of ImpactCollision blocks:

- *ImpactElastic*
- *ImpactExcitation*
- *ImpactIonization*
- *ImpactIonCollision*

Note:

BuiltIn The built-in cross sections that exist in VSim vary with the type of collision/interaction. Please see below to learn which cross sections exist for each collision/interaction type.

EEDL Evaluated Electron Data Library (EEDL) data is available for elements from Z=1 to 100. These cross sections can be downloaded from the International Atomic Energy Agency Nuclear Data Services website. Users can download either the complete library or individual evaluations. VSim currently uses the eedl/endl format for the libraries, so please be sure to download the right one.

userDefined There are too many interaction types and cross sections for us to list sources for each one. Outside of the EEDL and LXcat databases, many reliable cross sections can be found through publications and other databases online. Using the userDefined type users may import their own cross sections or ionization rates.

ImpactElastic

`kind = impactElastic`

Enables users to set up electron-impact elastic scattering collisions with the background neutral gas or gas mixtures. In this collision event, the incident electron loses a small percentage of its energy and is scattered after the collision. The scattered electron energy is computed from the formula given in *[VS95]*. It is given by:

$$\varepsilon_{sc} = \varepsilon_{inc}[1 - \frac{2m_e}{M_0}cos(\chi)]$$

where

ε_{inc} is the incident electron energy

ε_{sc} is the incident electron energy

m_e is the mass of the electron

M_0 is the mass of the neutral gas atom

χ is the scattering angle given by:

$$cos(\chi) = \frac{2 + \varepsilon_{inc} - 2(1 + \varepsilon_{inc})^{R_1}}{\varepsilon_{inc}}$$

The azimuthal scattering angle is given by:

$$\phi = 2\pi R_2$$

where in the above equations, R_1 and R_2 are random numbers between 0.0 and 1.0.

ImpactElastic Parameters

- **`crossSectionDataType`** Collision cross-section data type to be used in the handling of electron impact elastic collisions with the background fluid gas or gas mixtures. The supported types are eedl and userDefined.
  - **`eedl`**: Using the eedl cross-section option, users can utilize electron impact elastic scattering cross-section EEDL data for gas elements from Z = 1 to Z = 100. EEDL data files compatible with VSim can be downloaded from the link above.
  - **`userDefined`**: Using this option, users may specify known elastic collision cross-section data for their simulations. Gas types that are supported in the userDefined cross-section options are:
    - monatomic gas elements from Z = 1 to Z = 100
    - molecular gases: H_2, CO_2, CO, O_2, N_2
- **`crossSectionDataFile`** A list which defines the path or locations of the EEDL data file or the user-defined cross section data files that contain the elastic scattering collision cross section data. For the case of userDefined, please see *userDefined crossSectionDataFile Format*.
- **`scatteringType`** Supported options are *VahediSurendra* (see above) or *uniform*.

ImpactExcitation

`kind = impactExcitation`

Sets up electron-impact excitation collisions with background neutral gas. In this kind of collision, the incident electron loses energy equal to the excitation threshold energy and is scattered after the collision.

The scattered electron energy is computed to be:

$$\varepsilon_{sc} = \varepsilon_{inc} - \varepsilon_{ex}$$

where:

ε_{inc} is the incident electron energy

ε_{ex} is the excitation threshold energy

VSim uses the excitation threshold limit from either the EEDL data set or the user-defined cross-section data file.

The trajectory for the scattered electron is given by the scattering angle χ and azimuthal angle, ϕ. These angles are determined as below:

$$cos(\chi) = \frac{2 + \varepsilon_{inc} - 2(1 + \varepsilon_{inc})^{R_1}}{\varepsilon_{inc}}$$

and

$$\phi = 2\pi R_2$$

where R_1 and R_2 are random numbers between 0.0 and 1.0.

ImpactExcitation Parameters

- **`crossSectionDataType`** Collision cross-section data type to be used in the handling of electron impact exitation collisions with the background fluid gas or gas mixtures. The supported types are eedl and userDefined.

  - **eedl**: Using the eedl cross-section option, users can utilize electron impact excitation cross-section EEDL data for gas elements from Z = 1 to Z = 100. EEDL data files compatible with VSim can be downloaded from the link above.
  - **userDefined**: Using this option, users may specify known collision cross-section data for their simulations. Gas types that are supported in the userDefined cross-section options are: monatomic gas elements from Z = 1 to 100; molecular gases H_2, CO_2, CO, O_2, N_2

- **crossSectionDataFile** A list which defines the path or locations of the EEDL data file or the user-defined cross-section data files that contain the electron impact excitation cross section data. In the case of userDefined, please see *userDefined crossSectionDataFile Format*.
- **scatteringType** Supported options are *VahediSurendra* (see above) or *uniform*.

ImpactIonization

`kind = impactIonization`

The impactIonization kind enables users to set up electron-impact (and proton-impact) ionization collisions with background neutral gas. In this kind of collision, the incident particle loses both ionization threshold energy and the energy value of the secondary electron created during this collision.

The final scattered electron energy is computed to be:

$$\varepsilon_{sc} = \varepsilon_{inc} - \varepsilon_{iz} - \varepsilon_{se}$$

where:

ε_{inc} is the incident electron energy

ε_{iz} is the excitation threshold energy

ε_{se} is the secondary electron energy

The secondary electron energy is determined by:

$$\varepsilon_{se} = \omega\ tan(R_3 atan[\frac{\varepsilon_{inc} - \varepsilon_{iz}}{2\omega}])$$

where:

ω is a known function with a value of 15 eV

R_3 is a random value between 0.0 and 1.0.

The trajectory for the scattered electron is given by:

$$cos(\chi) = (\frac{\varepsilon_{se}}{\varepsilon_{inc} - \varepsilon_{iz}})^{0.5}$$

where:

χ is the scattering angle

ϕ is the azimuthal angle given by:

$$\phi = 2\pi R_2$$

where R_2 is a random number that varies between 0.0 and 1.0.

The scattering angle for the new secondary electron is determined using:

$$cos\chi_{se} = (\frac{\varepsilon_{se}}{\varepsilon_{inc} - \varepsilon_{iz}})^{0.5}$$

where: $\chi_s e$ is the scattering angle

ϕ is the azimuthal angle given by:

$$\phi = 2\pi R_2$$

where R_2 is a random number that varies between 0.0 and 1.0.

The energy of the ion created during this reaction is determined from the neutral gas temperature. The background neutral gas density is decremented appropriately at the ionization location based on the ion produced during the reaction.

ImpactIonization Parameters

- **`crossSectionDataType`** Collision cross-section data types for handling electron-impact ionization or proton-impact ionization with a background fluid gas or gas mixture. The supported types are builtIn, eedl, and userDefined. The default type of crossSectionDataType is `builtIn`.
  - **`builtIn`** (default): This option uses VSim's existing internal collision cross-section data. Currently VSim has builtIn cross-sections for electron-impact or proton-impact ionization reactions for the following gas types:
    - Electron-impact Ionization Gas Types: H_2, He, CO_2, CO, O_2, N_2, Ar, Ne
    - Proton-impact Ionization Gas Types: H, H_2, He, CO_2, CO, O_2, N_2

    VSim determines whether to use electron or proton built-in cross-section data for handling impact ionization based on the ratio of charge to mass. If this ratio is not that of an electron, VSim assumes that the incident particle is a proton (regardless of whether or not this is actually the case). The default type of background gas for electron-impact ionization is H_2; the default for proton-impact ionization is H. If the user chooses a gas for ionization that is not on the lists above, VSim will use the default gas for that ionization type. For example, if users choose H for electron-impact ionization, VSim will use the default H_2. Similarly, if users choose Ar for protons, VSim will default to H.
  - **`eedl`**: Using the eedl cross-section option, users can utilize the electron impact ionization cross-section EEDL data for gas elements Z = 1 to Z = 100. EEDL data files compatible with VSim can be downloaded from the link above.
  - **`userDefined`**: Using this option, users may specify known collision cross-section data for their simulations. Gas types that are supported in the userDefined cross-section options are: monatomic gas elements from Z = 1 to Z = 100; molecular gases H_2, CO_2, CO, O_2, and N_2.
- **`crossSectionDataFile`** A list which defines the path or locations of the EEDL data file or the user-defined cross-section data files that contain the electron impact ionization cross-section data. In the case of userDefined, please see *userDefined crossSectionDataFile Format*.
- **`scatteringType`** Supported options are *VahediSurendra* (see above) or *uniform*.
- **`ionizedElectronSpecies`** A list of secondary electron species that can result from impact ionization collisions of impactSpecies (electrons or protons) with the gas or gas mixtures listed under the neutralGas attribute. VSim checks whether an ionizedElectronSpecies vector of strings is provided. The strings should correspond to the names of the secondary electron Species objects, which must be defined in the VSim input file using Species blocks.
- **`ionSpecies`** A list of ion Species that can result from impact ionization collisions of impactSpecies (electrons or protons) with the gas or gas mixtures listed under the neutralGas attribute. VSim checks whether an ionSpecies vector of strings is provided. The strings should correspond to the names of the ion species and these must be defined using Species blocks in the VSim input file.

- **depleteBackgroundGas** Boolean specifying whether or not to deplete the background gas density when ionization events occur.

ImpactIonCollision

kind = impactIonCollision

The impactIonCollision kind enables users to set up ion-impact collisions with the background neutral gas. In this kind of collision, the incident ion is scattered (referred to as momentum exchange) or else undergoes a charge-exchange collision with the background neutral gas. In the case of momentum exchange, the incident ion loses no energy. In the charge-exchange collision event, the resulting ion energy value is determined based on the background neutral gas temperature.

ImpactIonCollision Parameters

- **crossSectionDataType** A list of collision cross-section data types for handling impact collisions with the background fluid gas or gas mixtures. The supported types are builtIn and userDefined.
  - **builtIn**: This option uses VSim's existing internal collision cross-section data. The builtIn ion impact charge exchange cross-section data is based on the semi-empirical formula from Lindsay *[LS05]* for all gas types except for Xe and Ar. For xenon and argon background neutral gases, the built-in cross-section data comes from the data found in experiments by Miller *[MPL+02]*.
  - **userDefined**: Using this option, users may specify known collision cross-section data for their simulations. Gas types that are supported in the userDefined cross-section options are: monatomic gas elements from Z = 1 to Z = 100; molecular gases H_2, CO_2, CO, O_2, and N_2.
- **crossSectionDataFile** A list which defines the paths or locations of the userDefined cross-section data files that contain the ion impact collision cross-section data. In the case of userDefined, please see *userDefined crossSectionDataFile Format*.
- **ionCollType** Defines the type of ion-impact collision. Choices are:
  - MomentumExchange
  - ChargeExchange

userDefined crossSectionDataFile Format

When using userDefined crossSectionDataType functionality, the cross-section data file must contain the following (in ASCII plain text):

- 1st Line: Atomic number of the neutral gas (integer).
- 2nd Line: Collision name (string)
  - For impactElastic this is `ELASTIC`
  - For impactExcitation this is `EXCITATION`
  - For impactIonization this is `IONIZATION`
- 3rd Line: Collision threshold energy (float)
- 4th Line: Number of collision data points (integer).
- 5th Line and later: Collision cross-section information for the event. The data is supplied in two columns of floating point values, separated by white space. The first column contains the incident electron energy values in units of eV and the second column contains the respective electron-neutral cross-section values in units of m^2.

For impactIonCollision, the format is slightly different:

- 1st Line: Atomic number of the neutral gas (integer).
- 2nd Line: Collision name (string)
  - For impactIonCollision this is `ION_ELASTIC` (momentum exchange) or `CHARGE_EXCHANGE` (charge exchange)
- 3rd Line: Number of collision data points (integer).
- 4th Line and later: Collision cross section information for the ion-neutral collision event. The data is supplied in two columns of floating point values, separated by white space. The first column contains the incident ion energy values in units of eV and the second column contains the respective ion-neutral cross-section values in units of m^2.

```
18
ELASTIC
0.0
10
2.0 3.5e-19
5.0 5.e-19
10.0 6.e-19
15.0 5.5e-19
17.0 4e-19
20.0 3.e-19
50.0 2.e-19
100.0 1.e-20
250.0 1.e-21
1000.0 1.e-22
```

```
54
CHARGE_EXCHANGE
10
0.5 5.19e-19
1.0 4.57e-19
2.0 3.95e-19
5.0 3.13e-19
10.0 2.52e-19
20.0 1.9e-19
40.0 1.29e-19
60.0 9.26e-20
100.0 4.71e-20
150.0 1.10e-20
```

Example ImpactCollision Block inside ImpactCollider

```
<ImpactCollider electronHeGasCollisions>

  neutralGas = [HeNeutralGas]
  impactSpecies = [electrons]
  neutralGasTemp = GAS_TEMP_IN_K
  <ImpactCollision ElecImpElastic>
    kind = impactElastic
    crossSectionDataType = [userDefined]
    crossSectionDataFile = [elasticCS.dat]
    scatteringType = uniform
```

(continues on next page)

(continued from previous page)

```
  </ImpactCollision>

  <ImpactCollision ElecImpExcitation1>
    kind = impactExcitation
    crossSectionDataType = [userDefined]
    crossSectionDataFile = [excitation1CS.dat]
    scatteringType = uniform
  </ImpactCollision>

  <ImpactCollision ElecImpExcitation2>
    kind = impactExcitation
    crossSectionDataType = [userDefined]
    crossSectionDataFile = [excitation2CS.dat]
    scatteringType = uniform
  </ImpactCollision>

  <ImpactCollision elecImpIonization>
    kind = impactIonization
    crossSectionDataType = [userDefined]
    crossSectionDataFile = [ionizationCS.dat]
    ionizedElectronSpecies = [electrons]
    ionSpecies = [He1]
    depleteBackgroundGas = false
    scatteringType = uniform
  </ImpactCollision>

</ImpactCollider>
```

3.14 Monte Carlo Interactions (DEPRECATED)

Note: Collisions set up using the Monte Carlo Interactions frame work will still work in VSim 9 simulations, but we recommend converting simulations to the new Reactions Framework

3.14.1 MonteCarloInteractions Block

MonteCarloInteractions

Blocks enable random processes in VSim, such as the modeling of random interactions between real particles or the random production of particles due to processes like tunneling ionization. These interactions can be modeled in a fully-kinetic fashion (i.e., with macroparticles colliding with other macroparticles) or in a partially- or non-kinetic fashion with fluids.

The MonteCarloInteractions pages describe the kinds of *IncidentSelector* and the kinds of *Interaction* blocks that have been implemented.

Note: With **variable-weight** particles, once it is determined that a reaction has occurred, the smaller of the two initial-state particles weight*number-or-particles-in-macro-particle is used to compute the number of collision events that occur. This number of "real" particles is subtracted from the initial state macroparticle (i.e. the weight is reduced) and another particle of the lighter weight is produced. If the computed weight goes below 0, then it is deleted.

For example, e + H2 -> H2+ + 2e-

For e (of weight w_e) and H2 (of weight w_H2), after the reaction one has two electrons, each of weight w_e, one H2+ of weight w_e, and one H2 of weight (w_H2 - w_e).

Note: Caution should be exercised when using binaryChargeExchange and binaryElastic reactions in the Monte Carlo framework with variable-weight particles; the results may be unreliable. Consider using the newer Reactions framework instead.

MonteCarloInteractions Parameters

There is only one kind of MonteCarloInteractions block, and therefore no kind attribute is needed.

The MonteCarloInteractions block must contain one (and only one) *IncidentSelector* block, which determines the algorithm for selecting which random incidents in a given time step are performed and in which order.

The MonteCarloInteractions block can contain any number of NullInteraction and *Interaction* blocks. The NullInteraction blocks pertain to non-kinetic interactions, where no macroparticles interact (though macroparticles may be produced). The Interaction blocks pertain to full- or partially-kinetic interactions, where macroparticles interact with other macroparticles or fluids.

Note:

BuiltIn The built in cross sections that exist vary with the type of collision/interaction. Please see the chart below to learn which cross sections exist for each collision/interaction type.

EEDL Evaluated Electron Data Library (EEDL) data is available for Z=1 to 100 elements. These cross sections can be downloaded from the *International Atomic Energy Agency Nuclear Data Services website (https://www-nds.iaea.org/epdl97/). You can download either the complete library or individual evaluations. Vorpal currently uses the eedl/endl format for the libraries, so please be sure to download the right one. Please see :ref:'montecarlointeractions-interactions-eedl* for more information.

LXcat LXcat is an open-access website for collecting, displaying, and downloading electron and ion data. You can download the data from the LXcat Website (http://fr.lxcat.net/home/). Please see *LXcatFile OAFunc* for more information.

ADK The modified ADK (Ammosov, Delone, and Krainov) formula for field ionization, first reported by Penetrante and Bardsley [Penetrante] and later corrected by Ilkov et al. [Ilkov], gives the ionization rate for any type of atom. Both time averaged (averagedADK) and time resolved (DCADK) formula are implemented. Please see *fieldIonization* or *nullFieldIonization* for more information.

functionDefined There are too many interaction types and cross sections for us to list sources for each one. Outside of the EEDL and LXcat databases, many reliable cross sections can be found through publications and other databases online. Using the functionDefined (or userDefinedFunc for ionization) you may import your own cross section or ionization rate. Please see *OAFunc Cross-Section Interface* for more information.

Types of collisions

Cross Sections

Type of collision	Reaction	Type vvvv	crossSection				
			builtIn	eedl	functionDefined		
					interpolated	expression	LXcatFile
impactElastic	$A + e \rightarrow A + e$ (scatter)	fluid or species	Xe	×	×	×	×
impactExcitation	$A + e \rightarrow A + e$ (excited)	fluid or species	Xe	×	×	×	×
impactIonCollisions	$A + A^+ \rightarrow A + A^+$ (momentum exchange) $A + A^+ \rightarrow A^+ + A$ (charge exchange)	fluid	Ar, Xe *[MPL+02]* All others *[LS05]*		×	×	×
impactIonization	$A + e \rightarrow A^+ + 2e$ (electron impact ionization) $A + B^+ \rightarrow A^+ + B^+ + e$ (proton impact ionization) $e + Xe^+ \rightarrow Xe^{++} + 2e$ (electon impact double ionization)	fluid or species	electron impact: H2, He, CO2, O2, N2, Ar, Ne, Xe, Xe+ proton impact: H, H2, He, CO2, CO, O2, N2	×	×	×	×
negativeIonDetachment	$A + B^- \rightarrow A + B + e$	fluid	H, H2	×	×	×	×
electronAttachment	$A + e \rightarrow A^-$	fluid or species	All		×	×	
threeBodyRecombination	$e + A^+ + e \rightarrow A + e$ (scatter)	species	All		×	×	
chargeExchange	$A^+ + A \rightarrow A + A^+$	species	All		×	×	×
binaryChargeExchange	$A + A^+ \rightarrow A^+ + A$	species	All		×	×	×
binaryRecombination	$AB^+ + e \rightarrow A + B$	species	All		×	×	×
binaryIonization	$A + e \rightarrow A^+ + 2e$ (electron impact ionization)	species	All		×	×	×
binaryExcitation	$A + e \rightarrow A + e$ (excited)	species	Xe, Xe^+, Xe^{++}		×	×	×
binaryElastic	$A + e \rightarrow A + e$ (scatter) $A + A \rightarrow A + A$ (scatter)	species	Xe		×	×	×
binaryDissociation	$A + e \rightarrow B + C + e$ (excited)	species	none		×	×	×
nullBgAbsorber		fluid	H, He, C, N, O, Na, Al, Si, P, Ar, Fe, Ni, Cu, Ge, Ag, Ba, Os, Pt, Au, Pb, U, Air, Water, Stainless		×		

Ionization Rates

Type of collision	Background Type	ionizationKind				
		builtIn	averagedADK DCADK	userDefinedFunc		
				interpolatedFromFile	expression	LXcatFile
fieldIonization	species	H, He, Li, Na, Rb, Cs	×	×	×	
nullFieldIonization	fluid	H, He, Li, Na, Rb, Cs	×	×	×	

Self Interactions

Type of interaction	Interaction	Background Type	crossSection				
			builtIn	eedl	functionDefined		
					interpolatedFromFile	expression	LXcatFile
oneBodyDecay / VADecay		species	N/A	N/A	N/A	N/A	N/A
nullSelfCombination	a + a → A	species	N/A	N/A	N/A	N/A	N/A
nullSelfSplit	A → a + a	species	N/A	N/A	N/A	N/A	N/A

Example MonteCarloInteractions Block

```
<MonteCarloInteractions myInteractions>

  <IncidentSelector mySelector>
    kind=unbiasedSelector
  </IncidentSelector>

  <NullInteraction absorber>
    kind = nullBgAbsorber
    species = muons
    fluid = hydrogen
    # specify the stopping power with an OAFunc
    stoppingPower = mysp
    <OAFunc mysp>
      kind = interpolatedFromFile
      filename = h2StoppingPower.dat
      # set bounds of the function from min and max x values in the file.
      # option specific to kind = interpolatedFromFile
      # setMinMaxFromFile = 1
      # f has to be >0.
      fmin = 0.
    </OAFunc>
```

(continues on next page)

(continued from previous page)

```
    # charge of the particle for which the stopping power is given
    refPtclCharge = ELEMCHARGE
    # mass of the particle for which the stopping power is given
    refPtclMass = PROTMASS
    multipleScattering = none
    straggling = none
  </NullInteraction>

  <Interaction Decay>
    kind = oneBodyDecay
    unstableSpecies = muons
    productSpecies = electrons
    lifetime = $2.*DT$
  </Interaction>

</MonteCarloInteractions>
```

3.14.2 IncidentSelector Block

IncidentSelector

`IncidentSelector:` Determines how random incidents are selected. The IncidentSelector block is defined at the top level of the *MonteCarloInteractions* block.

IncidentSelector Parameters

Presently, the IncidentSelector can be of two different kinds:

`nullOnlySelector:`
Works with VSimPD and VSimPA licenses.

To be used if the MonteCarloInteractions‘ block contains only *Null Interaction* blocks, each interaction is performed consecutively at each time step. If any Interaction blocks are present, they are ignored.

`unbiasedSelector`
Works with VSimPD and VSimPA licenses.

Selects incidents in an unbiased manner according to their probability only. This selector allows each macroparticle to interact only once per time step.

3.14.3 Interaction/NullInteraction Blocks

Monte Carlo Interactions Introduction

`Interaction:` Any of the interactions handled by the **`MonteCarloInteractions`** block. These interactions can be separated into two distinct phases. First, each interaction is assumed to have a random probability for occurrence, which depends on the initial state of the interacting objects. Second, each interact is assumed to have a randomly determined final state, which also can depend on the initial state of the interacting objects. MonteCarloInteractions interactions are grouped into three main categories depending on the kind of initial state modeled by the interaction. The three main categories of interactions are:

***Null Interaction*:** any interaction that does NOT depend upon the full state (position and/or velocity) of any particle in the initial state. The interaction is forced to occur at every time step (i.e., have a probability of 1). For some null interactions, the initial state of the interaction does not depend upon kinetically modeled particles, such as field ionization of a gas modeled as a fluid. Alternately, other null interactions are random processes that always occur, such as the integrated effect of multiple scattering and energy loss of a particle traveling through a moderating gas.

***Unary Interaction*:** any interaction that depends upon the full state (position and velocity) of only one particle in the initial state. The interaction is randomly occurring, and whose probability and final state depend on only one kinetically modeled particle in the initial state. A common kind of unary interaction is one that involves the ionization of a gas, modeled as a fluid, by an incident particle.

***Binary Interaction*:** any interaction that depends upon the full state (position and velocity) of two distinct particles in the initial state. The interaction is randomly occurring, and whose probability and final state depend on two kinetically modeled particles in the initial state. The most common kind of binary interaction involves the collision of two kinetically modeled particles.

Some interactions model the collision of one fundamental particle with another (though not necessarily kinetically modeled). For these interactions, the probability for occurrence depends upon the cross section for the associated scattering event. These cross-section-dependent interactions include:

- unary `chargeExchange`
- all of the unary impact collisions
- all of the binary interactions

Impact collisions

`Impact collisions:` Models the interaction of incident ions or electrons with a background fluid neutral gas.

It also models the interaction of incident ions or electrons with kinetic neutral particles. In this setup, the kinetic neutral particles within each computational cell are assembled together to calculate the neutral particle density results which will be used in handling the collision algorithm. This model is completely different from the binary interactions model. In binary interactions, each pair of particles is checked for the possibility of collision and then the collision is handled. This approach is computationally expensive when there are many particles per cell in a simulation.

All impact collisions are cross section based. The user can either choose to use the `builtIn` cross section, the `eedl` cross section, in which case an EEDL data file name must be specified, or the `functionDefined` cross section, in which case the cross section is defined via an **`OAFunc`** function (examples below). The EEDL (Electron Evaluated Data Library *[PCS91]* contains electron collision cross section information for monatomic elements Z = 1 to 100 defined by the International Atomic Energy Agency (IAEA)'s Nuclear Data Services. Vorpal has interfaces, via the TxPhysics library, that can parse this EEDL data library and obtain the necessary collision cross section data for the collision handling.

Each impact collision interaction can be represented by an **`Interaction`** block, inside the **`MonteCarloInteractions`** block, in the Vorpal input file. There are different kinds of impact collisions:

- **`impactElastic`**
- **`impactExcitation`**
- **`impactIonCollisions`**
- **`impactIonization`**
- **`negativeIonDetachment`**
- **`threeBodyRecombination`**

Electrons/Ions interactions with background fluid neutral gas: In this setup, all impact collisions model interaction between a **`Species`** (**`sortSpecies`** or **`cellSpecies`**) and a **`Fluid`** representing the neutral gas.

These two blocks must be defined in the input file and `impactSpecies` and `neutralGas` parameter values must refer to the name of these two blocks respectively.

Electrons/Ions interactions with kinetic neutral gas: In this set up, all impact collisions model interaction between a incident electron/ion **`Species`** and a kinetic neutral particle species. The `inNeutrals` parameter value must refer to the name of the **`Species`** block for kinetic neutral particle.

Note: These interactions replace the ImpactCollider/ImpactCollision blocks previously present in Vorpal.

Interaction

Interaction blocks pertain to fully-kinetic (binary) or partially-kinetic (unary) interactions, where macroparticles interact with other macroparticles or fluids.

Unary interactions are defined as any interaction that depends upon the full state (position and velocity) of only one particle in the initial state.

For example, kinetic electron scatter on a fluid, where both the position and velocity of kinetic species matter.

Binary interactions are defined as any interaction that depends upon the full state (position and velocity) of two distinct particles in the initial state.

For example, two kinetic species collide.

Interaction Parameters

`OAFunc`**:** Block to define any cross-section function that the user wishes to use.

Note: The same species block can describe the two species participating in the interaction.

All binary interactions in Vorpal depend on a cross-section. The cross-section in binary interactions can be either `builtIn` or user defined using an **`OAFunc`** block. Binary interactions' work with both basic dynamic species (also referred to simply as species) and cellspecies and with both constant and variable weight particles.

Note: Although the binary interactions allow a mix of both constant and variable weight particles participating in the interaction, it is preferable that all species type be either constant weight or variable weight.

NullInteraction

Null interactions are defined as any interaction that does NOT depend upon the full state (position and/or velocity) of any particle in the initial state.

For example, in nullBgAbsorber, where only kinetic species energy (velocity) matters for the interaction.

NullInteraction Parameters

`OAFunc`

Block to define any cross-section function that the user wishes to use.

Interaction Kinds

Fully-Kinetic Interactions:

binaryChargeExchange

Works with VSimPD license.

Kind of MonteCarlo interaction that models the charge exchange of two particles of the same atomic element.

Note: This feature replaces the chargeExchange feature, which was previously implemented in the Vorpal Collision feature.

Note: Caution should be exercised when using binaryChargeExchange reactions in the Monte Carlo framework with variable-weight species kinds; the results may be unreliable. Consider using the newer Reactions framework instead.

binaryChargeExchange Parameters

`crossSection` (*string*)
: Cross section to be used in the interaction. Possible values are `builtIn` or `functionDefined`. See below for required parameters for each choice.

`inSpeciesA` (*string*)
: Name of the first input species participating in the charge exchange interaction.

`inSpeciesB` (*string*)
: Name of the second input species participating in the charge exchange interaction.

`outSpeciesA` (*string*)
: Name of the output species resulting from particle inSpeciesA exchanging charge with inSpeciesB.

`outSpeciesB` (*string*)
: Name of the output species resulting from particle inSpeciesB exchanging charge with inSpeciesA.

builtIn Parameters

Please see *Types of collisions* for the available `builtIn` gases.

The `builtIn` option is based on the semi-empirical formula from *[LS05]* for all gas types except Xe and Ar. For Xe and Ar background neutral gases the `builtIn` cross section data is used from the data found in experiments by *[MPL+02]*.

If the `builtIn` cross section type is used the following parameters must be set:

`inSpeciesAtomicWeight` (*real*)
: Atomic weight of the incident species.

`inSpeciesName` (*string*)
: Name of the element of the incident species.

functionDefined Parameters

If the `functionDefined` cross section type is used, the following parameters must be set:

`OAFunc` (*block*, *required*)
An OAFunc block of name `crossSectionFunc` must be used inside the **`Interaction`** block. The OAFunc block allows the user to define its own cross section for the interaction, either through a two-column data file or through an expression. The kinds of OAFunc available for this interaction are `interpolatedFromFile`, `LXcatFile`, or `expression`. The OAFunc must return the value of the cross section in m^2.

Please see *OAFunc Block* for more information on the OAFunc block.

`crossSectionVariable` (*string*, *optional*, *default* = *velocity*)
Used in the case when an OAFunc function is given for the cross section to specify whether the parameter of the function is either the:

- relative collision velocity magnitude of the incident particle in m^{-1}: `crossSectionVariable = velocity`
- kinetic energy of the incident particle in eV: `crossSectionVariable = energy`

Example binaryChargeExchange Block

```
<Interaction chargeXchange>
  kind = binaryChargeExchange
  inSpeciesA = oxygen0
  inSpeciesB = oxygen1
  outSpeciesA = oxygen1
  outSpeciesB = oxygen0
  inSpeciesAtomicWeight = 16.0
  speciesName = oxygen
  crossSection = builtIn
</Interaction>
```

binaryRecombination

Works with VSimPD license.

kind of MonteCarlo interaction that models the recombination of an electron with an ionized atom.

The `builtIn` recombination rate coefficient in Vorpal is:

$$C_i = \alpha_r + \alpha_d$$

with:

$$\alpha_r = A_{rad}(T/10^4)^{-X_{rad}}$$

$$\alpha_d = A_{di}T^{-3/2}\exp(-T_0/T) \times [1 + B_{di}\exp(-T_1/T)]$$

where $T = 0.5mv^2$ is the temperature in K, with v the relative velocity of the two particles interacting. The coefficients A_{rad}, X_{rad}, A_{di}, B_{di}, T_0 and T_1 are calculated in *[SVS82]*, and must be specified in the input file.

Note: This feature replaces the recombination feature, which was previously implemented in the Vorpal collisions.

binaryRecombination Parameters

crossSection (*string*)
: Cross section to be used in the interaction. Possible values are `builtIn` or `functionDefined`. See below for required parameters for each choice.

electrons (*string*)
: Name of the **species** describing the electrons recombining with the ionized atom.

inSpecies (*string*)
: Name of the **species** describing the ionized atom.

outSpecies (*string*)
: Name of the recombination product **species**.

intElecMass (*real, optional*)
: Mass of the interacting electron. Default value is the mass specified in the *electrons* **species** block.

builtIn Parameters

Please see *Types of collisions* for the available `builtIn` gases.

The `builtIn` option is based on the data from Shull and Van Steenberg *[SVS82]*.

If the `builtIn` cross section type is used the following parameters must be set:

Arad (*real*)
: Coefficient used in the calculation of the recombination rate. This parameter can be found in tables in the paper by *[SVS82]*.

Xrad (*real*)
: Coefficient used in the calculation of the recombination rate. This parameter can be found in tables in the paper by *[SVS82]*.

Adi (*real*)
: Coefficient used in the calculation of the recombination rate. This parameter can be found in tables in the paper by *[SVS82]*.

Bdi (*real*)
: Coefficient used in the calculation of the recombination rate. This parameter can be found in tables in the paper by *[SVS82]*.

T0 (*real*)
: Coefficient used in the calculation of the recombination rate. This parameter can be found in tables in the paper by *[SVS82]*.

T1 (*real*)
: Coefficient used in the calculation of the recombination rate. This parameter can be found in tables in the paper by *[SVS82]*.

functionDefined Parameters

If the `functionDefined` cross section type is used, the following parameters must be set:

`OAFunc` (*block*, *required*)

An OAFunc block of name `crossSectionFunc` must be used inside the **`Interaction`** block. The OAFunc block allows the user to define its own cross section for the interaction, either through a two-column data file or through an expression. The kinds of OAFunc available for this interaction are `interpolatedFromFile`, `LXcatFile`, or `expression`. The OAFunc must return the value of the cross section in m^2.

Please see *OAFunc Block* for more information on the OAFunc block.

`crossSectionVariable` (*string*, *optional*, *default* = *velocity*)

Used in the case when an OAFunc function is given for the cross section to specify whether the parameter of the function is either the:

- Relative collision velocity magnitude of the incident particle in m^{-1}: `crossSectionVariable = velocity`
- Kinetic energy of the incident particle in eV: `crossSectionVariable = energy`

Example binaryRecombination Block

```
<Interaction binary>
  kind = binaryRecombination
  electrons = electrons
  intElecMass = 9.1093896999999993e-31
  inSpecies = oxygen1
  outSpecies = oxygen0
  crossSection = builtIn
  # use parameters for crossSection calculation
  Arad = 1.24e-6
  Xrad = 6.78e-1
  Adi = 4440.0
  Bdi = 9.25e-2
  T0 = 1.75e5
  T1 = 1.45e5
</Interaction>
```

binaryIonization

Works with VSimPD and VSimPA licenses.

Models the electron-impact ionization of a kinetically neutral gas species. In the process, the incident electron is scattered and loses the energy of the secondary electron created.

The `builtIn` collisional ionization rate in Vorpal is *[SVS82]*:

$$C_i = A_{col} T^{1/2} \exp(-T_{col}/T)\,(1 + 0.1T/T_{col})^{-1}$$

where $T = 0.5mv^2$ is the temperature in K, with v the relative velocity of the two particles interacting. The coefficients A_{col} and T_{col} are calculated in *[SVS82]*, and must be defined in the input file.

Note: This feature replaces the ionization feature, which was previously implemented in the Collision feature.

binaryIonization Parameters

`crossSection` (*string*, *optional*, *default = builtIn*)
: Cross-section to be used in the interaction. Possible values are `builtIn` or `functionDefined`. See below for required parameters for each choice.

`electrons` (*string*)
: Name of the impact electron species.

`intElecMass` (*real*, *optional*)
: Mass of the interacting electron. Default value is the mass specified in the species block.

`inSpecies` (*string*)
: Name of the background gas species to be ionized.

`outSpecies` (*string*)
: Name of the ion species resulting from the ionization of *`inSpecies`*.

`speciesName` (*string*)
: Name of the atomic element involved in the ionization process.

`thresholdEnergy` (*real*)
: Ionization threshold energy used to calculate the energy of the secondary electron.

builtIn Parameters

Please see *Types of collisions* for the available `builtIn` gases.

The `builtIn` option is based on the data from *[SVS82]*. For Xe ionization, the cross-section data is based off *[AMMS00]*.

If the `builtIn` cross-section type is used the following parameters must be set:

`Acol` (*real*)
: Coefficient used in the calculation of the ionization rate. This parameter can be found in tables in the paper by *[SVS82]*.

`Tcol` (*real*)
: Coefficient used in the calculation of the ionization rate. This parameter can be found in tables in the paper by *[SVS82]*.

functionDefined Parameters

If the `functionDefined` cross-section type is used, the following parameters must be set:

`OAFunc` (*block*, *required*)
: An OAFunc block of name `crossSectionFunc` must be used inside the **`Interaction`** block. The OAFunc block allows the user to define its own cross-section for the interaction, either through a two-column data file or through an expression. The kinds of OAFunc available for this interaction are `interpolatedFromFile`, `LXcatFile`, or `expression`. The OAFunc must return the value of the cross-section in m^2.

Please see *OAFunc Block* for more information on the OAFunc block.

`crossSectionVariable` (*string*, *optional*, *default* = *velocity*)
Used in the case when an OAFunc function is given for the cross section to specify whether the parameter of the function is either the:

- Relative collision velocity magnitude of the incident particle in m^{-1}: `crossSectionVariable = velocity`
- Kinetic energy of the incident particle in eV: `crossSectionVariable = energy`

Example binaryIonization Block

```
<Interaction ionization>
  kind = binaryIonization
  electrons = electrons
  intElecMass = 9.1093896999999993e-31
  speciesName = oxygen
  inSpecies = oxygen0
  outSpecies= oxygen1
  crossSection = builtIn
  Acol = 0.000436
  Tcol = 1.58e5
</Interaction>
```

binaryExcitation

Works with VSimPD license.

Models the electron-impact of a kinetically modeled species of neutral gas or ions. In the process, the incident electron is scattered and loses the threshold energy. The scattered electron energy is given by:

$$\epsilon_{sc} = \epsilon_{inc} - \epsilon_{ex}$$

where ϵ_{inc} and ϵ_{ex} are the incident electron energy and the excitation threshold energy, respectively.

Note: Currently supports the electron-impact xenon neutral excitation, electron-impact singly charged xenon ion excitation, and electron-impact doubly charged xenon ion excitation interaction when using the `builtIn` cross-section type.

binaryExcitation Parameter

`crossSection` (*string*)
cross-section to be used in the interaction. Possible values are `builtIn`, `eedl` or `functionDefined`. See below for required parameters for each choice.

`inSpeciesA` (*string*, *required*)
Name of the impact electron species.

`inSpeciesB` (*string*, *required*)
Name of the impact neutral gas or ions.

`intElecMass` (*real*, *optional*)
: Mass of the impact electron species. If not specified its value is obtained from the mass specified in the `inSpecies` species block.

`thresholdEnergy` (*real*, *optional*)
: Excitation threshold energy. Default is set to 0.0.

builtIn Parameters

Please see *Types of collisions* for the available `builtIn` gases.

eedl Parameters

If the `eedl` cross-section type is used, the following parameters must be set:

`crossSectionDataFile` (*string*)
: This parameter is necessary when `crossSection = eedl` is specified and points to the file containing the EEDL data.

functionDefined Parameters

If the `functionDefined` cross-section type is used, the following parameters must be set:

`OAFunc` (*block*, *required*)
: An OAFunc block of name `crossSectionFunc` must be used inside the **`Interaction`** block. The OAFunc block allows the user to define its own cross-section for the interaction, either through a two-column data file or through an expression. The kinds of OAFunc available for this interaction are `interpolatedFromFile`, `LXcatFile`, or `expression`. The OAFunc must return the value of the cross-section in m^2.

  Please see *OAFunc Block* for more information on the OAFunc block.

`crossSectionVariable` (*string*, *optional*, *default = energy*)
: Used in the case when an OAFunc function is given for the cross section to specify whether the parameter of the function is either the:

  - Relative collision velocity magnitude of the incident particle in m^{-1}: `crossSectionVariable = velocity`
  - Kinetic energy of the incident particle in eV: `crossSectionVariable = energy`

Example binaryExcitation Block

```
<Interaction excitation>
  kind= binaryExcitation
  crossSection = builtIn
  thresholdEnergy = 19.10
  inSpeciesA = electrons
  inSpeciesB = xenon2
</Interaction>
```

binaryElastic

Works with VSimPD license.

Models the electron/ion/neutral impact elastic scattering collision with a kinetically modeled gas species. In the process, the incident particle is scattered and loses the threshold energy. The scattered particle energy is given by:

$$\epsilon_{sc} = \epsilon_{inc} - \epsilon_{el}$$

where ϵ_{inc} and ϵ_{el} are the incident particle energy and the elastic threshold energy, respectively.

Note: The builtIn model currently supports the electron-impact xenon neutral elastic scattering, ion-impact xenon neutral elastic scattering, and xenon neutral-neutral elastic scattering interaction types.

Note: Caution should be exercised when using binaryElastic reactions in the Monte Carlo framework with variable-weight species kinds; the results may be unreliable. Consider using the newer Reactions framework instead.

binaryElastic Parameters

crossSection (*optional*, *default = builtIn*)
Cross-section to be used in the interaction. Possible values are `builtIn` and `functionDefined`. See below for required parameters for each choice.

inSpeciesA (*string*, *required*)
Name of the impact species.

inSpeciesB (*string*, *required*)
Name of the impact neutral gas.

intElecMass (*real*, *optional*)
Mass of the impact electron species. If not specified its value is obtained from the mass specified in the *inSpeciesA* species block.

thresholdEnergy (*real*, *optional*)
Elastic threshold energy. Default is set to 0.0.

builtIn Parameters

Please see *Types of collisions* for the available `builtIn` gases.

functionDefined Parameters

If the `functionDefined` cross-section type is used, the following parameters must be set:

OAFunc (*block*, *required*)
An OAFunc block of name `crossSectionFunc` must be used inside the **Interaction** block. The OAFunc block allows the user to define its own cross-section for the interaction, either through a two-column data file or through an expression. The kinds of OAFunc available for this interaction are `interpolatedFromFile`, `LXcatFile`, or `expression`. The OAFunc must return the value of the cross-section in m^2.

Please see *OAFunc Block* for more information on the OAFunc block.

crossSectionVariable (*string*, *optional*, *default* = *energy*)
Used in the case when an OAFunc function is given for the cross section to specify whether the parameter of the function is either the:

- Relative collision velocity magnitude of the incident particle in m^{-1}: `crossSectionVariable = velocity`
- Kinetic energy of the incident particle in eV: `crossSectionVariable = energy`

Example binaryElastic Block

```
<Interaction elastic>
  kind= binaryElastic
  inSpeciesA = electrons
  inSpeciesB = xenon
</Interaction>
```

binaryDissociation

Works with VSimPD license.

Kind of MonteCarlo interaction that models the electron-impact dissociation of a kinetically modeled gas species.

e + A -> B + C + e

The outgoing species assume isotropic scatter. The outgoing species energy is set using the outSpeciesEnergyRatio parameter that defines the ratio of the energies between the two output species.

binaryDissociation Parameters

electrons (*string*, *required*)
Name of the impact electron **species**.

inSpecies (*string*, *required*)
Name of the background **species** to be dissociated.

intElecMass (*string*)
Mass of the interacting electron. Default value is the mass specified in the *electrons* **species** block.

intSpecMass (*string*)
Mass of the interacting neutral species. Default value is the mass specified in the *inSpecies* **species** block.

outSpecies1 (*string*, *required*)
Name of the species product 1 of the dissociation on inSpecies.

outSpecies2 (*string*, *required*)
Name of the species product 2 of the dissociation on inSpecies.

outSpec1Mass (*string*)
Mass of the outgoing species 1. Default value is the mass specified in the *outSpecies1* **species** block.

outSpec2Mass (*string*)
Mass of the outgoing species 2. Default value is the mass specified in the *outSpecies2* **species** block.

speciesName (*string*, *required*)
Name of the molecular element involved in the dissociation process.

thresoldEnergy (*float*)
Dissociation threshold energy used to calculate the energy of the secondary electron.

outSpeciesEnergyRatio (*float*)
Ratio of energies between output species.

crossSection (*string*)
cross-section to be used in the interaction. Possible values are `functionDefined`. See below for required parameters.

functionDefined Parameters

If the `functionDefined` cross-section type is used, the following parameters must be set:

OAFunc (*block*, *required*)
An OAFunc block of name `crossSectionFunc` must be used inside the **Interaction** block. The OAFunc block allows the user to define its own cross-section for the interaction, either through a two-column data file or through an expression. The kinds of OAFunc available for this interaction are `interpolatedFromFile`, `LXcatFile`, or `expression`. The OAFunc must return the value of the cross-section in m^2.

Please see *OAFunc Block* for more information on the OAFunc block.

crossSectionVariable (*string*, *optional*, *default* = *velocity*)
Used in the case when an OAFunc function is given for the cross section to specify whether the parameter of the function is either the:

- Relative collision velocity magnitude of the incident particle in m^{-1}: `crossSectionVariable = velocity`
- Kinetic energy of the incident particle in eV: `crossSectionVariable = energy`

Example binaryDissociation Block

```
<MonteCarloInteractions main>
  <IncidentSelector mySelector>
    kind=unbiasedSelector
  </IncidentSelector>
  <Interaction ionization>
    kind = binaryDissociation
    electrons = electrons
    intElecMass = 9.1093896999999993e-31
    intSpecMass = 1.e-27
    outSpec1Mass = 0.2e-27
    outSpec2Mass = 0.8e-27
    speciesName = oxygen
    inSpecies = oxygen0
    outSpecies1 = oxygen1
    outSpecies2 = oxygen2
    outSpeciesEnergyRatio = 1.
    crossSection = functionDefined
    crossSectionVariable = velocity
    <OAFunc crossSectionFunc>
      kind = interpolatedFromFile
      filename = dissXSecVel.dat
    </OAFunc>
```

(continues on next page)

(continued from previous page)

```
    thresholdEnergy = 15.76
  </Interaction>
</MonteCarloInteractions>
```

Partially-Kinetic Interactions:

chargeExchange

Works with VSimPD license.

Kind of MonteCarlo interaction that models ionic charge exchange of an ion with a background neutral gas, based on the method described by Spicer, Wang, and Brieda *[SWB06]*. The method assumes that the neutral atoms are slow enough that their velocity is negligible (for the purposes of computing the relative ion-neutral velocity), and that the ions are moving quickly through the background neutrals. The method also assumes the charge-exchange is between a neutral atom and a singly-ionized atom.

Note: This feature replaces the cellChargeExchange kind previously implemented in the Vorpal collisions.

chargeExchange Parameters

`crossSection` (*string*)
: cross-section to be used in the interaction. Possible values are `builtIn` or `functionDefined`. See below for required parameters for each choice.

builtIn Parameters

Please see *Types of collisions* for the available `builtIn` gases.

If the `builtIn` cross-section type is used, the following parameters must be set:

`inIons` (*string*)
: Specifies the initial state ionized **`Species`**.

`inNeutrals` (*string*)
: Specifies the initial state neutral background species.

`outNeutrals` (*string*)
: Specifies the neutral ion species produced by the interaction.

`outIons` (*string*)
: Specifies the ionized ion species produced by the interaction.

`vthermNeutral` (*real*)
: This parameter specifies the thermal velocity, in meters per second, of the initial state neutral background species. For the model to be valid, this value should be significantly less than the average velocity of the initial state ion species.

functionDefined Parameters

If the `functionDefined` cross-section type is used, the following parameters must be set:

`OAFunc` (*block*, *required*)

An OAFunc block of name `crossSectionFunc` must be used inside the **`Interaction`** block. The OAFunc block allows the user to define its own cross-section for the interaction, either through a two-column data file or through an expression. The kinds of OAFunc available for this interaction are `interpolatedFromFile`, `LXcatFile`, or `expression`. The OAFunc must return the value of the cross-section in m^2.

Please see *OAFunc Block* for more information on the OAFunc block.

`crossSectionVariable` (*string*, *optional*, *default* = *velocity*)

Used in the case when an OAFunc function is given for the cross section to specify whether the parameter of the function is either the:

- Relative collision velocity magnitude of the incident particle in m^{-1}: `crossSectionVariable = velocity`
- Kinetic energy of the incident particle in eV: `crossSectionVariable = energy`

Example chargeExchange Block

In this example, the builtIn cross-section is used by default. You can refer to **`impactCollisions`** interaction examples to see how to use the cross-section with an OAFunc function.

```
<Interaction chargeXchange>
  kind = chargeExchange
  vthermNeutrals=16.0
  inIons=oxygen1
  inNeutrals=oxygen0
  outNeutrals=oxygen0
  outIons=oxygen1
  crossSection = builtIn
</Interaction>
```

oneBodyDecay/oneBodyVADecay

`oneBodyDecay` / `oneBodyVADecay`: Works with VSimPD license.

Kinds of MonteCarlo interactions that model the decay of an unstable particle into a single final-state particle.

oneBodyVADecay choses the energy and momentum of the final-state particle is based on the V-A theory. This is valid, for example, for describing the decay of muons into electrons, where the final-state neutrinos are not kinetically modeled. The momentum of the final-state particle however takes into account that part of the momentum of the unstable particle is carried away by one or several neutrinos.

Note: This feature replaces the **`oneBodyVADecay`** and **`oneBodyVADecayVW`** features previously implemented in Vorpal collisions.

oneBodyDecay/oneBodyVADecay Parameters

unstableSpecies (*string*)
: This specifies the name of the unstable (decaying) species.

lifetime (*real*)
: This specifies the unstable species lifetime in seconds.

productSpecies (*string*)
: This specifies the **Species** product of the decay of the unstable species.

isDissociation (*boolean*)
: If true, the unstable species decays into two resultant **productSpecies** particles.

Example oneBodyDecay Block

```
<Interaction Decay>
  kind = oneBodyDecay
  unstableSpecies = muons
  productSpecies = electrons
  lifetime = 2.2e-6
<Interaction>
```

fieldIonization

Works with VSimPA license.

Kind of MonteCarlo interaction that models the tunneling ionization of a kinetically modeled background pre-ionized gas. This process is similar to the **nullFieldIonization**, except that the background gas is described by a **Species** block, i.e. the background medium can be ionized.

Built-in models for field ionization of H, He, Li, Na, Rb and Cs exist, using Tech-X's proprietary txphysics library. This uses the analytical expressions derived by *[K+65]*.

An OAFunc can be used to get the rate of this interaction. It must provide the effective ionization rate from the initial state to the final state as a function of electric field E (V/m). This OAFunc can either be an analytical expression or import from a two column data file. See *OAFunc Block* for details.

The modified ADK *[ADK86]* (Ammosov, Delone, and Krainov) formula, updated in *[DK91]* and first adapted to PIC by Penetrante and Bardsley *[PB91]* and later corrected by Ilkov *et al.* *[IDC92]*, which gives the ionization rate for any type of atom, can also be used. The user must specify the ionization potential of each atom by using the *energy* keyword. Both time averaged and time resolved formula are implemented. The time averaged formula (`ionizationKind = averagedADK`) should be used when modeling a linearly polarized electric field and the time step is larger than the oscillation period (i.e., `dt` is much larger than the period of the field in question). This might occur, for example, when using an envelope model for the laser pulse. The time resolved formula (`ionizationKind = DCADK`) is the correct choice in most cases, i.e., any time the total electric field is fully time resolved.

The ionization rate for a time resolved field is given by:

$$R_i = 4.13 \times 10^{16} \frac{Z^2}{2n_{\text{eff}}^2} \left(\frac{2e}{n_{\text{eff}}} \right)^{2n_{\text{eff}}} \frac{1}{2\pi n_{\text{eff}}} \left(2 \frac{E_h}{E_L} \frac{Z^3}{n_{\text{eff}}^3} \right)^{2n_{\text{eff}}-1} \exp \left[-\frac{2}{3} \frac{E_h}{E_L} \frac{Z^3}{n_{\text{eff}}^3} \right] (\text{s}^{-1})$$

where Z is the charge state of the ionized particle, $n_{\text{eff}} = Z/\sqrt{U_{\text{ion}}/13.6[\text{eV}]}$, U_{ion} is the ionization potential in eV, $E_h = m_e^2 q_e^5/(4\pi\epsilon_0\hbar^4) = 5.13 \times 10^{11}\text{V/m}$, and E_L is the electric field strength at the particle position.

The time averaged modified ADK formula is given by:

$$R_i = 6.6 \times 10^{16} \frac{e}{\pi} \frac{Z^2}{n_{\text{eff}}^{4.5}} \left(10.87 \times \frac{E_h}{E_L} \frac{Z^3}{n_{\text{eff}}^4}\right)^{2n_{\text{eff}}-1.5} \exp\left[-\frac{2}{3} \frac{E_h}{E_L} \frac{Z^3}{n_{\text{eff}}^3}\right] (\text{s}^{-1})$$

Subsequent validation work on the tunneling ionization models in VSim was conducted and is demonstrated in *[CSMZ06]*, *[CES+12]* and *[CCMG+13]*. The model is valid up to approximately energy densities of $10^{23} - 10^{24}$ above which Barrier Suppression Ionization is likely to be the dominant effect, and one way also want to consider vacuum pair-production in the ionization cross-section.

The model assumes the background is a gas, that is to say that atoms are well separated with respect to their size. At very high number density, the model may break down. See references to the Mott Transition for further information.

fieldIonization Parameters

`ionizationKind` (*optional*, *default = builtIn*)
: The ionization rate to be used in the interaction. Possible values are `builtIn`, `averagedADK`, `DCADK` and `userDefinedFunc`. See below for required parameters for each choice.

`input` (*string*)
: The **`species`** that undergoes the tunneling ionization process.

`charge` (*integer*)
: Charge of the input **`species`** (in unit of |e|).

`atomicName` (*string*)
: Atomic name of the gas corresponding to the ionized **`species`**. Possible names are the fluid names listed in the **`nullFieldIonization`** interaction.

`electrons` (*string*)
: Name of the **`species`** representing the electrons product of the ionization process.

`ions` (*string*)
: Name of the **`species`** representing the ionized particles after the ionization process has occurred.

`polarizationFlag` (*integer*)
: When the electric field magnitude is approximately constant in a Vorpal time step, 1 is the correct choice; i.e. any time the total electric field is fully time-resolved. 0 is used for the case where you are modeling a linearly polarized electric field, but the time step is larger than the oscillation period. This might occur, for example, when using an envelope model for the laser pulse. This should be applied when the dt of Vorpal is much larger than the period of the field in question.

`frequency` (*real*)
: Used in the initialization process; the frequency of a laser pulse if there is such a pulse (the default scenario when using field ionization). If a static field is used, this parameter should be set to 0.

builtIn Parameters

Please see *Types of collisions* for the available `builtIn` gases.

userDefinedFunc Parameters

If the `userDefinedFunc` ionizationKind type is used, the following parameters must be set:

OAFunc (*block*, *required*)
Effective field ionization rate as a function of electric field E (V/m). If not provided, built in rates are used. The kinds of OAFunc available for this interaction are `interpolatedFromFile`, or `expression`.

Please see *OAFunc Block* for more information on the OAFunc block.

averagedADK or DCADK Parameters

If the `averagedADK or DCADK` ionizationKind type is used, the following parameters must be set:

energy (*real*, *required*)
The ionization potential in eV.

fieldIonization Block

```
<Interaction FieldIonizationCs1>
  kind = fieldIonization
  input = Cs1
  charge = 1
  atomicName = Cs
  electrons = electrons
  ions = Cs2
  polarizationFlag = 1
  frequency = 1.e15
</Interaction>
```

threeBodyRecombination

Works with VSimPD license.

A unary interaction that models three-body recombination of electron-electron-ion collisions. In this one electron is treated in kinetic fashion and the second electron is assumed to be the background electron density and the ion also assumed as a background ion density.

threeBodyRecombination Parameters

impactSpecies (*string*, *required*)
Name of the impact species.

inIons (*string*, *required*)
Name of the species representing the input ions.

inElectrons (*string*, *required*)
Name of the species representing the input electrons.

outNeutrals (*string*, *required*)
Name of the neutral output species.

vthermNeutrals (*float*, *required*)
The thermal velocity of the neutral species.

alpha (*float*, *optional*)
Parameter used to calculate the three-body recombination probability.

tempExpFactor (*string*)
    Parameter used to calculate the three-body recombination probability.

crossSection (*optional*, *default = builtIn*)
    Cross section to be used in the interaction. Possible values are `builtIn` and `functionDefined`. See below for required parameters for each choice.

builtIn Parameters

Please see *Types of collisions* for the available `builtIn` gases.

functionDefined Parameters

If the `functionDefined` cross section type is used, the following parameters must be set:

OAFunc (*block*, *required*)
    An OAFunc block of name `crossSectionFunc` must be used inside the **Interaction** block. The OAFunc block allows the user to define its own cross section for the interaction, either through a two-column data file or through an expression. The kinds of OAFunc available for this interaction are `interpolatedFromFile`, `LXcatFile`, or `expression`. The OAFunc must return the value of the cross section in m^2.

    Please see *OAFunc Block* for more information on the OAFunc block.

crossSectionVariable (*string*, *optional*, *default = energy*)
    Used in the case when an OAFunc function is given for the cross section to specify whether the parameter of the function is either the:

- Relative collision velocity magnitude of the incident particle in m^{-1}: `crossSectionVariable = velocity`
- Kinetic energy of the incident particle in eV: `crossSectionVariable = energy`

threeBodyRecombination Example Block

```
<Interaction chargeXchange>
  kind=threeBodyRecombination
  vthermNeutrals=16.0
  inIons=XePlus
  inElectrons = electrons
  impactSpecies = electrons
  outNeutrals=Xe0
  alpha = 1.125e-39
  tempExpFactor = -4.5
  crossSection=builtIn
</Interaction>
```

Partially-Kinetic Interactions working with a neutral Fluid block:

impactElastic

    Works with VSimPD license.

Kind of MonteCarlo interaction that models the elastic scattering collision of an electron with a background neutral gas or species. In this kind of collision, the incident electron loses small percentage of its energy and is scattered after the collision. The scattered electron energy, ϵ_{sc}, is computed to be:

$$\epsilon_{sc} = \epsilon_{inc}\left(1 - \frac{2m_e M_0}{(m_e + M_0)^2}(1. - \cos\chi)\right)$$

where:

ϵ_{inc} is the incident electron energy,

m_e is the mass of the incident electron,

M_0 is the mass of the neutral gas atom,

χ is the scattering angle given by:

$$\cos\chi = \frac{2 + \epsilon_{inc} - 2\,(1 + \epsilon_{inc})^{R_1}}{\epsilon_{inc}}$$

The azimuthal scattering angle is given by:

$\varphi = 2\pi R_2$

In the above equations R_1 and R_2 are two random numbers uniformly distributed between 0 and 1.

Note: This feature replaces the impactElastic feature previously implemented in Vorpal **`ionization`**.

impactElastic Parameters

`crossSection` (*string*, *required*)
Cross section to be used in the interaction. Possible values are `builtIn`, `eedl`, and `functionDefined`. See below for required parameters for each choice.

`impactSpecies` (*string*, *required*)
Name of the incident particles described by a species block in the input file.

`neutralGas` (*string*, *required*)
Name of the Fluid block describing the background neutral gas. For more information on the kinds of allowed neutral Gases, please see *Working with neutralGas Fluids and the gasKind Parameter*.

`neutralGasTemp` (*real*, *optional*, *default = 300.*)
Temperature of the background neutral gas in Kelvin. This parameter is used to determine the incident ion velocity relative to the neutral gas atoms and the final ion velocity.

`isNeutralGasFluid` (*string*, *optional*, *default = true*)
Defines whether the neutral gas tracked is a fluid background or not. The default is true, i.e. the neutral gas is fluid. When this option is false, the kinetic neutral species and its gas type must be defined using `inNeutrals` and `inNeutralsGasType`.

`inNeutrals` (string)

Name of the impact kinetic neutral particle described by a species block in the input file. This attribute is required when `isNeutralGasFluid` is set to false.

`inNeutralsGasType` (string)

Type of the impact kinetic neutral gas particle. This attribute is required when `isNeutralGasFluid` is set to false.

depleteBackgroundGas (*string*, *optional*, *default = true*)
A flag to modify the background fluid gas density when performing the interaction.

leaveIncidentUnchanged (*bool*, *optional*, *default = false*)
When enabled, this parameter leaves the impact (incident) particle unchanged after the collision is processed. This is helpful when the scattering effect of the incident particle is treated as a bulk effect, such as in the **nullBgAbsorber**.

force1D (*integer*, *optional*, *default = 0*)
Force the interaction to occur in 1D.

builtIn Parameters

Please see *Types of collisions* for the available `builtIn` gases.

eedl Parameters

If the `eedl` cross section type is used, the following parameters must be set:

crossSectionDataFile (*string*)
Points to the file containing the EEDL data.

functionDefined Parameters

If the `functionDefined` cross section type is used, the following parameters must be set:

OAFunc (*block*, *required*)
An OAFunc block of name `crossSectionFunc` must be used inside the **Interaction** block. The OAFunc block allows the user to define its own cross section for the interaction, either through a two-column data file or through an expression. The kinds of OAFunc available for this interaction are `interpolatedFromFile`, `LXcatFile`, or `expression`. The OAFunc must return the value of the cross section in m^2.

Please see *OAFunc Block* for more information on the OAFunc block.

crossSectionVariable (*string*, *optional*, *default = energy*)
Used in the case when an OAFunc function is given for the cross section to specify whether the parameter of the function is either the:

- Relative collision velocity magnitude of the incident particle in m^{-1}: `crossSectionVariable = velocity`
- Kinetic energy of the incident particle in eV: `crossSectionVariable = energy`

impactElastic Block

```
<Interaction ElecImpElastic>
  kind = impactElastic
  neutralGas = ArNeutralGas
  impactSpecies = electrons
  crossSection = eedl
  crossSectionDataFile = eedlAr.dat
</Interaction>
```

impactExcitation

Works with VSimPD license.

This kind MonteCarlo interaction models electron-impact excitation collisions with a background neutral gas. In this collision, the incident electron first loses the energy corresponding to the excitation energy threshold and is scattered after the collision. The scattered electron energy is given by:

$$\epsilon_{sc} = \epsilon_{inc} - \epsilon_{ex}$$

where ϵ_{inc} and ϵ_{ex} are the incident electron energy and the excitation threshold energy, respectively. For a user defined cross section with an **`OAFunc`** function, the excitation threshold energy must be explicitly specified in the input file through the `excitationThreshold` parameter, otherwise Vorpal uses the threshold limit from the EEDL data set.

The final direction of the incident electron is given by the scattering angle X defined as:

$$\cos \chi = \frac{2 + \epsilon_{sc} - 2(1 + \epsilon_{sc})^{R_1}}{\epsilon_{sc}}$$

and the azimuthal angle $\varphi = 2\pi R_2$. In the above equations R_1 and R_2 are two random numbers uniformly distributed between 0 and 1.

Note: This feature replaces the impactExcitation feature previously implemented in Vorpal ionization.

impactExcitation Parameters

`crossSection` (*string*, *required*)
Cross section to be used in the interaction. Possible values are `builtIn`, `eedl`, and `functionDefined`. See below for required parameters for each choice.

`impactSpecies` (*string*, *required*)
Name of the incident particles described by a species block in the input file.

`neutralGas` (*string*, *required*)
Name of the Fluid block describing the background neutral gas. For more information on the kinds of allowed neutral Gases, please see *Working with neutralGas Fluids and the gasKind Parameter*.

`neutralGasTemp` (*real*, *optional*, *default = 300.*)
Temperature of the background neutral gas in Kelvin. This parameter is used to determine the incident ion velocity relative to the neutral gas atoms and the final ion velocity.

`isNeutralGasFluid` (*string*, *optional*, *default = true*)
Defines whether the neutral gas tracked is a fluid background or not. The default is true, i.e. the neutral gas is fluid. When this option is false, the kinetic neutral species and its gas type must be defined using `inNeutrals` and `inNeutralsGasType`.

`inNeutrals` (string)

Name of the impact kinetic neutral particle described by a species block in the input file. This attribute is required when `isNeutralGasFluid` is set to false.

`inNeutralsGasType` (string)

Type of the impact kinetic neutral gas particle. This attribute is required when `isNeutralGasFluid` is set to false.

depleteBackgroundGas (*string*, *optional*, *default = true*)
A flag to modify the background fluid gas density when performing the interaction.

leaveIncidentUnchanged (*bool*, *optional*, *default = false*)
When enabled, this parameter leaves the impact (incident) particle unchanged after the collision is processed. This is helpful when the scattering effect of the incident particle is treated as a bulk effect, such as in the **nullBgAbsorber**.

force1D (*integer*, *optional*, *default = 0*)
Force the interaction to occur in 1D, i.e. the scattered angle of the electron after the interaction is always 0.

excitationThreshold (*real*)
The excitation threshold energy (in eV). To be set in the case the cross section is user defined with an OAFunc block. This value is used to determine the scattered electron energy in the excitation collision.

excAtomSpecies (*string*)
Name of the excited atom species, product of the excitation of an atom of the background neutral gas. To track the excited atom as separate species, this string must be included in the Interaction block. By ignoring this parameter makes the interaction not to track the excited atom in simulation.

builtIn Parameters

Please see *Types of collisions* for the available `builtIn` gases.

eedl Parameters

If the `eedl` cross section type is used, the following parameters must be set:

crossSectionDataFile (*string*)
Points to the file containing the EEDL data.

functionDefined Parameters

If the `functionDefined` cross section type is used, the following parameters must be set:

OAFunc (*block*, *required*)
An OAFunc block of name `crossSectionFunc` must be used inside the **Interaction** block. The OAFunc block allows the user to define its own cross section for the interaction, either through a two-column data file or through an expression. The kinds of OAFunc available for this interaction are `interpolatedFromFile`, `LXcatFile`, or `expression`. The OAFunc must return the value of the cross section in m^2.

Please see *OAFunc Block* for more information on the OAFunc block.

crossSectionVariable (*string*, *optional*, *default = energy*)
Used in the case when an OAFunc function is given for the cross section to specify whether the parameter of the function is either the:

- Relative collision velocity magnitude of the incident particle in m^{-1}: `crossSectionVariable = velocity`
- Kinetic energy of the incident particle in eV: `crossSectionVariable = energy`

Example impactExcitation Block

```
<Interaction CuExcitation>
  kind = impactExcitation
  neutralGas = CuNeutralGas
  impactSpecies = electrons
  crossSection = eedl
  crossSectionDataFile = eedlCu.dat
</Interaction>
```

impactIonCollisions

Works with VSimPD license.

This kind of MonteCarlo interaction models ion impact collisions with a background neutral gas. In this kind of collision the incident ion is scattered (referred to as momentum exchange) or else undergoes charge exchange collision with the background neutral gas. In the momentum exchange collision, the incident ion does not lose energy. In the charge exchange interaction, the resulting ion energy is determined based on the background neutral gas temperature.

Note: This feature replaces the impactIonCollisions feature previously implemented in Vorpal ionization.

impactIonCollisions Parameters

`crossSection` (*string*, *required*)
Cross section to be used in the interaction. Possible values are `builtIn`, and `functionDefined`. See below for required parameters for each choice.

`ionCollType` (*string*, *required*)
Type of ion impact collision, one of:

- `MomentumExchange`
- `ChargeExchange`.

`impactSpecies` (*string*, *required*)
Name of the incident particles described by a species block in the input file.

`neutralGas` (*string*, *required*)
Name of the Fluid block describing the background neutral gas. For more information on the kinds of allowed neutral Gases, please see *Working with neutralGas Fluids and the gasKind Parameter*.

`neutralGasTemp` (*real*, *optional*, *default = 300.*)
Temperature of the background neutral gas in Kelvin. This parameter is used to determine the incident ion velocity relative to the neutral gas atoms and the final ion velocity.

`depleteBackgroundGas` (*string*, *optional*, *default = true*)
A flag to modify the background fluid gas density when performing the interaction.

`leaveIncidentUnchanged` (*bool*, *optional*, *default = false*)
When enabled, this parameter leaves the impact (incident) particle unchanged after the collision is processed. This is helpful when the scattering effect of the incident particle is treated as a bulk effect, such as in the **`nullBgAbsorber`**.

builtIn Parameters

The `builtIn` option is based on the semi-empirical formula from Lindsay *[LS05]* for all gas types except Xe and Ar. For Xe and Ar background neutral gases the `builtIn` cross section data is used from the data found in experiments by Miller *[MPL+02]*.

Please see *Types of collisions* for the available `builtIn` gases.

functionDefined Parameters

If the `functionDefined` cross section type is used, the following parameters must be set:

OAFunc (*block*, *required*)

An OAFunc block of name `crossSectionFunc` must be used inside the **Interaction** block. The OAFunc block allows the user to define its own cross section for the interaction, either through a two-column data file or through an expression. The kinds of OAFunc available for this interaction are `interpolatedFromFile`, `LXcatFile`, or `expression`. The OAFunc must return the value of the cross section in m^2.

Please see *OAFunc Block* for more information on the OAFunc block.

crossSectionVariable (*string*, *optional*, *default* = *energy*)

Used in the case when an OAFunc function is given for the cross section to specify whether the parameter of the function is either the:

- Relative collision velocity magnitude of the incident particle in m^{-1}: `crossSectionVariable = velocity`
- Kinetic energy of the incident particle in eV: `crossSectionVariable = energy`

Example impactIonCollisions Block

```
<MonteCarloInteraction ArIonCollider>

  <IncidentSelector mySelector>
    kind=unbiasedSelector
  </IncidentSelector>


  <Interaction ArIonElasticM>
    kind = impactIonCollisions
    neutralGas = ArNeutralGas
    impactSpecies = Ar1
    neutralGasTemp = 300
    ionCollType = MomentumExchange
    crossSection = builtIn
  </Interaction>

  <Interaction ArIonCEX>
    kind = impactIonCollisions
    neutralGas = ArNeutralGas
    impactSpecies = Ar1
    neutralGasTemp = 300
    ionCollType = ChargeExchange
    crossSection = builtIn
```

(continues on next page)

(continued from previous page)

```
  </Interaction>

</MonteCarloInteractions>
```

impactIonization

Works with VSimPD and VSimPA licenses.

This kind of MonteCarlo interaction models electron or proton impact ionization of a background neutral gas. In this kind of collision, the incident electron loses both ionization threshold energy and the energy value of the secondary electron created during this collision. The scattered particle energy is then given by:

$$\epsilon_{sc} = \epsilon_{inc} - \epsilon_{iz} - \epsilon_{se}$$

where:

ϵ_{inc} the incident particle energy

ϵ_{iz} the ionization energy threshold

ϵ_{se} the secondary electron energy which is given by:

$$\epsilon_{se} = \omega \tan \left[R_3 \tan^{-1} \left(\frac{\epsilon_{inc} - \epsilon_{iz}}{2\omega} \right) \right]$$

where ω is a know function with a value of 15 eV and R_3 is a random number with a uniform distribution between 0 and 1.

In the case of a `functionDefined` cross-section using an **`OAFunc`** function, the ionization energy threshold must be explicitly specified in the input file using the `ionizationThreshold` parameter. Otherwise, the ionization threshold is either determined from the TxPhysics database or from the EEDL dataset.

The trajectory of the incident particle after the collision is determined by the scattering angle χ such that:

$$\cos \chi = \left(\frac{\epsilon_{sc}}{\epsilon_{inc} - \epsilon_{iz}} \right)^{0.5}$$

and the azimuthal angle $\varphi = 2\pi R_2$, where R_2 is a random number uniformly distributed between 0 and 1.

The scattering angle of the secondary electron is determined using:

$$\cos \chi_{sc} = \left(\frac{\epsilon_{se}}{\epsilon_{inc} - \epsilon_{iz}} \right)^{0.5}$$

and the azimuthal angle is $\varphi_{se} = 2\pi R_2$, where R_2 is a random number uniformly distributed between 0 and 1.

The energy of the created ion during this interaction is determined based on the neutral gas temperature. The neutral gas density is decremented appropriately at the ionization location based on the ion produced during the reaction.

Note: This feature replaces the impactIonization feature previously implemented in Vorpal ionization.

impactIonization Parameters

crossSection (*string*, *required*)
: cross-section to be used in the interaction. Possible values are `builtIn`, `eedl`, and `functionDefined`. See below for required parameters for each choice.

impactSpecies (*string*, *required*)
: Name of the incident particles described by a species block in the input file.

neutralGas (*string*, *required*)
: Name of the Fluid block describing the background neutral gas. For more information on the kinds of allowed neutral Gases, please see *Working with neutralGas Fluids and the gasKind Parameter*.

neutralGasTemp (*real*, *optional*, *default = 300.*)
: Temperature of the background neutral gas in Kelvin. This parameter is used to determine the incident ion velocity relative to the neutral gas atoms and the final ion velocity.

isNeutralGasFluid (*string*, *optional*, *default = true*)
: Defines whether the neutral gas tracked is a fluid background or not. The default is true, i.e. the neutral gas is fluid. When this option is false, the kinetic neutral species and its gas type must be defined using `inNeutrals` and `inNeutralsGasType`.

  inNeutrals (*string*)
  : Name of the impact kinetic neutral particle described by a species block in the input file. This attribute is required when `isNeutralGasFluid` is set to false.

  inNeutralsGasType (*string*)
  : Type of the impact kinetic neutral gas particle. This attribute is required when `isNeutralGasFluid` is set to false.

depleteBackgroundGas (*string*, *optional*, *default = true*)
: A flag to modify the background fluid gas density when performing the interaction.

leaveIncidentUnchanged (*bool*, *optional*, *default = false*)
: When enabled, this parameter leaves the impact (incident) particle unchanged after the collision is processed. This is helpful when the scattering effect of the incident particle is treated as a bulk effect, such as in the **nullBgAbsorber**.

force1D (*integer*, *optional*, *default = 0*)
: Forces the interaction to occur in 1D.

ionizedElectronSpecies (*string*)
: Name of the secondary electron species, product of the ionization.

ionSpecies (*string*)
: Name of the ion species, product of the ionization of an atom of the background neutral gas.

builtIn Parameters

The `builtIn` cross-section uses data from the TxPhysics library, based on the formulas given in *[Rei08]*.

Please see *Types of collisions* for the available `builtIn` gases.

Vorpal determines whether to use electron or proton built-in cross-section data for handling impact ionization based on the ratio of charge to mass. If the ratio is not that of an electron, Vorpal assumes that the incident particle is a proton (regardless of whether or not this is actually the case).

The default type of background gas for electron-impact ionization is H2; the default for proton-impact ionization is H. If you choose a gas for ionization that is not on the correct list, Vorpal will use the default gas for that ionization type.

For example, if you choose H for electron impact ionization, Vorpal will use the default H2. Similarly, if you choose Ar for protons, Vorpal will default to H.

eedl Parameters

If the `eedl` cross-section type is used, the following parameters must be set:

`crossSectionDataFile` (*string*)
: Points to the file containing the EEDL data.

functionDefined Parameters

If the `functionDefined` cross-section type is used, the following parameters must be set:

`ionizationThreshold` (*real*)
: The ionization threshold energy (in eV). Must be set if the cross-section is used defined with an OAFunc block.

`OAFunc` (*block*, *required*)
: An OAFunc block of name `crossSectionFunc` must be used inside the **`Interaction`** block. The OAFunc block allows the user to define its own cross-section for the interaction, either through a two-column data file or through an expression. The kinds of OAFunc available for this interaction are `interpolatedFromFile`, `LXcatFile`, or `expression`. The OAFunc must return the value of the cross-section in m^2.

  Please see *OAFunc Block* for more information on the OAFunc block.

`crossSectionVariable` (*string*, *optional*, *default* = *energy*)
: Used in the case when an OAFunc function is given for the cross section to specify whether the parameter of the function is either the:

  - Relative collision velocity magnitude of the incident particle in m^{-1}: `crossSectionVariable = velocity`
  - Kinetic energy of the incident particle in eV: `crossSectionVariable = energy`

Example impactIonization Block

```
<Interaction ArIonizer>
  kind = impactIonization
  neutralGas = ArNeutralGas
  impactSpecies = PrimaryElectrons
  crossSection = functionDefined
  <OAFunc crossSectionFunc>
    kind = interpolatedFromFile
    filename = electronCSection.dat
  </OAFunc>
  ionizedElectronSpecies = SecondaryElectrons
  ionSpecies = Ar1
  ionizationThreshold = 15.76
</Interaction>
```

negativeIonDetachment

Works with VSimPD license.

This kind of MonteCarlo interaction models electron detachment from a negative ion impact collision with a background neutral gas. In this kind of collision, the incident negative ion loses the electron and becomes a neutral particle. The detached electron's energy is calculated with two options. Option 1 (the default) uses the relation given in Reiser's book *[Rei08]*:

$$T_e = \frac{2.}{3.0}\sqrt{\frac{T_{inc}T_{thr}}{m_{ratio}}}$$

where:

T_e the detached electron energy in eV

T_{inc} the incident negative ion energy in eV

T_{thr} the threshold energy in eV (~15.0 eV rough estimate)

m_{ratio} the mass ratio of incident ion and electron which is given by:

$$m_{ratio} = \frac{m_{ion}}{m_e}$$

In option 2, the detached electron energy is given by the energy specified by the user in the input block. This option is enabled when the attribute `useThermalDist` is set to 1 in the *Interaction* block. The detached electron velocity is set based on the mean velocity *vbar* and the width of the electron thermal velocity distribution *vsig*.

negativeIonDetachment Parameters

`crossSection` (*string*, *required*)
cross-section to be used in the interaction. Possible values are `builtIn`, and `functionDefined`. See below for required parameters for each choice.

`impactSpecies` (*string*, *required*)
Name of the incident particles described by a species block in the input file.

`neutralGas` (*string*, *required*)
Name of the Fluid block describing the background neutral gas. For more information on the kinds of allowed neutral Gases, please see *Working with neutralGas Fluids and the gasKind Parameter*.

`neutralGasTemp` (*real*, *optional*, *default = 300.*)
Temperature of the background neutral gas in Kelvin. This parameter is used to determine the incident ion velocity relative to the neutral gas atoms and the final ion velocity.

`isNeutralGasFluid` (*string*, *optional*, *default = true*)
Defines whether the neutral gas tracked is a fluid background or not. The default is true, i.e. the neutral gas is fluid. When this option is false, the kinetic neutral species and its gas type must be defined using `inNeutrals` and `inNeutralsGasType`.

`inNeutrals` (string)

Name of the impact kinetic neutral particle described by a species block in the input file. This attribute is required when `isNeutralGasFluid` is set to false.

`inNeutralsGasType` (string)

Type of the impact kinetic neutral gas particle. This attribute is required when `isNeutralGasFluid` is set to false.

`depleteBackgroundGas` (*string*, *optional*, *default = true*)
A flag to modify the background fluid gas density when performing the interaction.

leaveIncidentUnchanged (*bool*, *optional*, *default = false*)
When enabled, this parameter leaves the impact (incident) particle unchanged after the collision is processed. This is helpful when the scattering effect of the incident particle is treated as a bulk effect, such as in the **nullBgAbsorber**.

detachmentThreshold (*real*, *optional*, *default = 0.0*)
Negative ion threshold energy (in eV) for detachment reaction.

detachedElectronSpecies (*string*, *required*)
Name of the detached electron species, product of the ion detachment reaction.

detachedNeutralSpecies (*string*, *optional*)
Name of the neutral species. If this is not set then the neutral species resulting from electron detachment is not tracked.

crossSecScaleFactor (*real*, *optional*, *default = 1.0*)
A factor to scale the cross-section values.

useThermalDist (*integer*, *optional*, *default = 0*)
A flag to turn-on thermal distribution energy for detached electrons. If this flag is not set, then the electron energy value is based on the expression given by Reiser's book.

`vbar` (real, required)

Mean value of the electron thermal-distribution energy. This parameter is required when `useThermalDist` flag is set to 1.

`vsig` (real, required)

Widths of detached electron thermal energy. This parameter is required when `useThermalDist` flag is set to 1.

builtIn Parameters

The cross-section data for these interactions are based on ORNL's Atomic Data for Fusion *[BHF+90]*.

Please see *Types of collisions* for the available `builtIn` gases.

functionDefined Parameters

If the `functionDefined` cross-section type is used, the following parameters must be set:

OAFunc (*block*, *required*)
An OAFunc block of name `crossSectionFunc` must be used inside the **Interaction** block. The OAFunc block allows the user to define its own cross-section for the interaction, either through a two-column data file or through an expression. The kinds of OAFunc available for this interaction are `interpolatedFromFile`, `LXcatFile`, or `expression`. The OAFunc must return the value of the cross-section in m^2.

Please see *OAFunc Block* for more information on the OAFunc block.

crossSectionVariable (*string*, *optional*, *default = energy*)
Used in the case when an OAFunc function is given for the cross section to specify whether the parameter of the function is either the:

- Relative collision velocity magnitude of the incident particle in m^{-1}: `crossSectionVariable = velocity`
- Kinetic energy of the incident particle in eV: `crossSectionVariable = energy`

Example negativeIonDetachment Block

```
<Interaction HminusIonDetachment>
  kind = negativeIonDetachment
  neutralGas = H2NeutralGas
  impactSpecies = Hminus
  crossSection = builtIn
  detachedElectronSpecies = electrons
  detachedNeutralSpecies = Hneutral
  detachmentThreshold = 0.75
</Interaction>
```

electronAttachment

works with VSimPD license.

A MonteCarlo interaction that models electron impact of a background neutral gas to cause attachment.

electronAttachment Parameters

neutralGas (*string*, *required*)
Name of the block describing the background neutral gas. For more information on the kinds of allowed neutral Gases, please see *Working with neutralGas Fluids and the gasKind Parameter*.

impactSpecies (*string*, *required*)
Name of the impact species

ionSpecies (*string*, *required*)
Name of the ion species. Product of the ionization of an atom of the background gas.

force1D (*integer*, *option*, *default = 0*)
Flag to force the interaction to occur in 1D.

crossSection (*optional*, *default = functionDefined*)
Cross section to be used in the interaction. Possible value is only `functionDefined`. See below for required parameters.

functionDefined Parameters

If the `functionDefined` cross section type is used, the following parameters must be set:

OAFunc (*block*, *required*)
An OAFunc block of name `crossSectionFunc` must be used inside the **Interaction** block. The OAFunc block allows the user to define its own cross section for the interaction, either through a two-column data file or through an expression. The kinds of OAFunc available for this interaction are `interpolatedFromFile`, or `expression`. The OAFunc must return the value of the cross section in m^2.

Please see *OAFunc Block* for more information on the OAFunc block.

crossSectionVariable (*string*, *optional*, *default = energy*)
Used in the case when an OAFunc function is given for the cross section to specify whether the parameter of the function is either the:

- Relative collision velocity magnitude of the incident particle in m^{-1}: `crossSectionVariable = velocity`
- Kinetic energy of the incident particle in eV: `crossSectionVariable = energy`

electronAttachment Example Block

```
<Interaction CO2Attachment>
  kind = electronAttachment
  neutralGas = CO2NeutralGas
  impactSpecies = electrons
  # use an OAFunc to define the cross section
  crossSection = functionDefined
   <OAFunc crossSectionFunc>
     # read data from a file
     kind = interpolatedFromFile
     filename = electronAttachmentCS.dat
   </OAFunc>
  ionSpecies = CO2minus
</Interaction>
```

NullInteraction Kinds

NullInteractions:

nullSelfCombination

Works with VSimPD license.

Kind of MonteCarlo interaction that combines macroparticles of a single species together into one single macroparticle. This interaction is designed to use variable-weight species.

Note: This feature replaces the **`selfCombCollision`** previously implemented in Vorpal **`collision`**.

nullSelfCombination Parameters

`species` (*string*)
This specifies the name of the **`Species`** block representing the particles to be combined.

`threshold` (*integer*)
This specifies the number of particles that must exist in a cell before self-combination.

`weight_max` (*real*)
This specifies the maximum variable-weight to be given to a particle when self-combining variable-weight species.

`algorithm_kind` (*integer*)
This specifies the kind of algorithm to be used when combining macroparticles. The possible values are 1, 2, 3, 4 or 5. Option 1 indicates a simple pair-wise combination of macroparticles, deleting one and adding the number of real particles (weight) to the other (decimation). Option 2 indicates a slightly more accurate pair-wise approach, deleting one macroparticle and assigning the other the mean position and velocity of the pair as

well as the total weight of the pair (inelastic). Option 3 is an elastic combination approach, where quartets of macroparticles are combined into pairs that conserve energy and momentum (elastic). Option 4 is a pair-wise combining approach similar to Option 2 except that it allows users to avoid combining particles that are above the user specified limit. Option 5 is a pairwise approach in which the particles within a cell are grouped in velocity phase-space bins and then particles in each velocity phase-space bin are combined pair-wise. In this approach energy and momentum are conserved.

force1D (*integer*)
: This option only applies to *algorithm_kind* 3 (elastic). If non-zero (true), this flag forces the velocities of the new particles to be in the 0 direction, rather than in a random 3D direction.

noCombThresholdKE (*float*, *optional*, *default = 10000.*)
: This option only applies to *algorithm_kind* 4. User specified kinetic energy limit in eV. This enables users to set certain energetic particles not to participate in self-combination. Only energy values below this limit are selected for combination.

numPSBins (*integer vector, default = [2 2 2]*)
: This option only applies to *algorithm_kind* 5. This enables users to set up the number of velocity phase-space bins to consider in grouping particles in velocity phase-space. If the number of phase space bins large, then the simulation will become slower. A reasonable choice would be using [4 4 4].

Example nullSelfCombination Block

```
<NullInteraction elecCollision>
  kind = nullSelfCombination
  species = electrons
  algorithm_kind = 1
  weight_max = 1000.
  threshold = 2
</NullInteraction>
```

nullSelfSplit

Works with VSimPD license.

Kind of MonteCarlo interaction that splits macroparticles of a single species into two or more macroparticles. This interaction is designed to use variable-weight species. This interaction is applied at each time step.

nullSelfSplit Parameters

species (*string*, *required*)
: This specifies the name of the **Species** block representing the particles to be split.

thresholdFunc (*block*, *required*)
: This specifies the maximum number of macroparticles existing in a cell for the self-split operation to be applied. The thresholdFunc STFunc can be a space-time dependent function. This threshold is evaluated at each cell and time step. Splitting will not take place in any cell with a greater number of macroparticles than this threshold.

minWeight (*real*, *optional*)
: This specifies the minimum variable-weight to be given to a particle when self-split variable-weight species. The default value is 1e-5. If the original particle weight is below the *minWeight* then the split is not performed.

algorithmType (*integer*, *optional*)
This specifies the type of algorithm to be used when splitting macroparticles. The possible values are 1 and 2. Algorithm 1 (default) performs a simple, fast split operation, whereas option 2 is more accurate but more computationally intensive. In algorithm 1, the weight of the macro particle is simply split to one half and a new macroparticle gets added at the same location with one half weight value. In algorithm 2, the split operation is done such that the number of macro particles within a cell meets the requirement of the threshold value specified by the user.

splitInterval (*integer*, *optional*)
This option specifies the number of time steps interval between two successive split operations. The default value is 1, i.e. the split operation is done at every time step.

Example nullSelfSplit Block

```
<NullInteraction electronSelfSplit>
  kind = nullSelfSplit
  species = electrons
  algorithmType = 1
  splitInterval = 1
  minWeight = 1e-3
  <STFunc thresholdFunc>
    kind = expression
    expression = 100.0
  </STFunc>
</NullInteraction>
```

NullInteractions working with a neutralGas Fluid block:

nullBgAbsorber

Works with VSimPD license.

Kind of MonteCarlo interaction that models the loss of energy and scattering of a particle interacting with a background gas. nullBgAbsorber is used in conjunction with a **Fluid** block and a **Species** block.

nullBgAbsorber Parameters

species (*string*, *required*)
This is the name of the species interacting with the background gas. The species must be defined in the input file using a **Species** block. The background absorber works with both `sortSpecies` and `cellSpecies` in Vorpal. The species can be constant weight or variable weight.

fluid (*string*, *required*)
This is the name of the background gas, defined as a **Fluid** block in the input file. To use the `builtIn` models of energy loss, multiple scattering, and energy straggling, the Fluid must be of kind `neutralGas`. For more information on the kinds of allowed neutral Gases, please see *Working with neutralGas Fluids and the gasKind Parameter*.

stoppingPower (*string*, *required*)
This attribute specifies the stopping power function to be used for energy loss in the gas. Possible values are `builtIn` or `functionDefined`. See below for required parameters for each choice.

materialMass (*string*, *required*)
The material scatterer mass in kg

fluidTemperature (*float*, *non-negative*, *optional*, *default = 0.0*)
This parameter specifies the electron temperature of the fluid, in eV. If the temperature is 0.0, then all of the electronic stopping comes from electrons that are bound to the fluid atoms. If the temperature is non-zero, then some electronic stopping is from bound electrons, and some is from free electrons.

multipleScattering (*string*, *optional*, *default = none*)
This string specifies which model to use for multiple scattering of the incident particle. The accepted values are `none` (turns multiple scattering off), `ThomasFermi` and `LenzJensen` (uses TxPhysics models based on the Thomas-Fermi and Lenz-Jensen screening potentials), and `gaussian` (a simple gaussian model for the mean scattering angle). When using the `gaussian` option, the parameter `radiationLength` is required, in kg/m^2, to determine the RMS angular spread experienced by particles traveling through the material.

radiationLength (*float*, *optional*)
The radiation length for the gaussian multiple scattering option

straggling (*string*, *optional*, *default = none*)
This string specifies which model to use for energy straggling of the incident particle. The accepted values are `none` (turns energy straggling off) and `builtIn` (uses a TxPhysics model to calculate the Vavilov distribution).

atomicRatio (*float*, *optional*)
The ratio of the scatterer's atomic number to atomic mass

thicknessFunc (*block*, *optional*)
Use an STFunc block of name `thicknessFunc` to specify a thickness function for the material. See below for parameters related to the STFunc block.

builtIn Parameters

If the `builtIn` stopping power type is used, the following parameters must be set:

speciesAtomicNumber (*real*)
This parameter is required when using the `builtIn` stopping power. It defines the effective atomic number (Z) of the incident species.

functionDefined Parameters

If the `functionDefined` stopping power type is used, the following parameters must be set:

OAFunc (*block*, *required*)
An OAFunc block of name `stoppingPowerFunc` must be used inside the **NullInteraction** block. The OAFunc block allows the user to define its own stopping power for the interaction, either through a two-column data file or through an expression. The kinds of OAFunc available for this interaction are `interpolatedFromFile`, or `expression`. The OAFunc must return the value of the stopping power as a function of the kinetic energy in MeV and return a value in MeV-cm^2/g .

Please see *OAFunc Block* for more information on the OAFunc block.

refPtclMass (*real*)
This parameter is required when using the `OAFunc` stopping power. This defines the mass of the incident particle (in kg) for which the stopping power data is given in the `stoppingPower` OAFunc function.

refPtclCharge (*real*)
This parameter is required when using the `OAFunc` stopping power. This defines the charge number of the incident particle for which the stopping power data is given in the stoppingPower OAFunc function.

thicknessFunc Parameters

Use an STFunc block to specify a thickness function for the material. The following parameters are possible when using the STFunc block of name `thicknessFunc`:

`gridBoundary` (*string*, *required*)
The name of the grid boundary for the location of the absorbing material.

`materialName` (*string*, *required*)
The name of the material absorbing the particles

`materialDensity` (*string*, *required*, *default = 0.0*)
The material scatterer density in m^-3

`removeStoppedParticles` (*string*, *option*, *default = 0*)
Whether to delete particles once they get stopped in the gas. 0 for false, 1 for true

Example nullBgAbsorber Block

This is an example of using an OAFunc function block to define the stopping power. In this case the stopping power function is defined through a two-column data file (`h2StoppingPower.dat`), which gives the proton stopping power in hydrogen. Blocks `<Species refMuon>` and `<Fluid hydrogen>` are also present in the input file.

```
<NullInteraction absorber>
  kind = nullBgAbsorber
  species = refMuon
  fluid = hydrogen
  # specify the stopping power with an OAFunc
  stoppingPower = functionDefined
  materialMass = 1.66e-27
  <OAFunc stoppingPowerFunc>
    kind = interpolatedFromFile
    filename = h2StoppingPower.dat
    # set bounds of the function from min and max x values in the file.
    # option specific to kind = interpolatedFromFile
    # setMinMaxFromFile = 1
    # f has to be >0.
    fmin = 0.
  </OAFunc>
  # charge of the particle for which the stopping power is given
  refPtclCharge = ELEMCHARGE
  # mass of the particle for which the stopping power is given
  refPtclMass = PROTMASS
  multipleScattering = none
  straggling = none
</NullInteraction>
```

Using the builtIn stopping power (requires TxPhysics):

```
<NullInteraction absorber>
  kind = nullBgAbsorber
  species = muons
  fluid = hydrogen
  stoppingPower = builtIn
  speciesAtomicNumber = 1.
  multipleScattering = LenzJensen
```

(continues on next page)

(continued from previous page)

```
  straggling = builtIn
</NullInteraction>
```

nullFieldIonization

Works with VSimPD and VSimPA licenses.

Kind of MonteCarlo interaction that models tunneling ionization of a background neutral gas, the quantum mechanical phenomena where by applying an electric field, an electron can escape the potential barrier of an atom represented by a background neutral gas. The background gas is described by a `Fluid` block in the input file. If you require a background gas represented by a `Species` block see *fieldIonization*. During the interaction, an ion and an electron are created; these must be defined by two Species block in the input file. Vorpal uses the field associated with the ion species to determine if the background gas is ionized. When ions are created due to tunneling ionization, the density of the background gas is reduced accordingly at the position of the ionization event.

Built-in models for field ionization of H, He, Li, Na, Rb and Cs exist, using Tech-X's proprietary txphysics library. This uses the analytical expressions derived by *[K+65]*.

Alternatively an OAFunc can be used to provide the rate of this interaction. It must provide the effective ionization rate from the initial state to the final state as a function of electric field E (V/m). This OAFunc can either be an analytical expression or import from a two-column data file. See *OAFunc Block* for details.

The modified ADK (Ammosov, Delone, and Krainov) *[ADK86]* formula, updated in *[DK91]* and first adapted to PIC by Penetrante and Bardsley *[PB91]* and later corrected by Ilkov *et al.* *[IDC92]*, which gives the ionization rate for any type of atom, can also be used. The user must specify the ionization potential of each atom by using the `energy` keyword. Both time averaged and time resolved formula are implemented. The time averaged formula (`ionizationKind = averagedADK`) should be used when modeling a linearly polarized electric field and the time step is larger than the oscillation period (i.e., `dt` is much larger than the period of the field in question). This might occur, for example, when using an envelope model for the laser pulse. The time resolved formula (`ionizationKind = DCADK`) is the correct choice in most cases, i.e., any time the total electric field is fully time resolved.

The ionization rate for a time resolved field is given by:

$$R_i = 4.13 \times 10^{16} \frac{Z^2}{2n_{\text{eff}}^2} \left(\frac{2e}{n_{\text{eff}}}\right)^{2n_{\text{eff}}} \frac{1}{2\pi n_{\text{eff}}} \left(2\frac{E_h}{E_L}\frac{Z^3}{n_{\text{eff}}^3}\right)^{2n_{\text{eff}}-1} \exp\left[-\frac{2}{3}\frac{E_h}{E_L}\frac{Z^3}{n_{\text{eff}}^3}\right] (\text{s}^{-1})$$

where Z is the charge state of the ionized particle, $n_{\text{eff}} = Z/\sqrt{U_{\text{ion}}/13.6[\text{eV}]}$, U_{ion} is the ionization potential in eV, $E_h = m_e^2 q_e^5/(4\pi\epsilon_0\hbar^4) = 5.13 \times 10^{11}\text{V/m}$, and E_L is the electric field strength at the particle position.

The time averaged modified ADK formula is given by:

$$R_i = 6.6 \times 10^{16} \frac{e}{\pi} \frac{Z^2}{n_{\text{eff}}^{4.5}} \left(10.87 \times \frac{E_h}{E_L}\frac{Z^3}{n_{\text{eff}}^4}\right)^{2n_{\text{eff}}-1.5} \exp\left[-\frac{2}{3}\frac{E_h}{E_L}\frac{Z^3}{n_{\text{eff}}^3}\right] (\text{s}^{-1})$$

Subsequent validation work on the tunneling ionization models in VSim was conducted and is demonstrated in .. [ChenJAP06], *[CES+12]* and *[CCMG+13]*. The model is valid up to approximately energy densities of $10^{23} - 10^{24}$ above which Barrier Suppression Ionization is likely to be the dominant effect, and one way also want to consider vacuum pair-production in the ionization cross-section.

The model assumes the background is a gas, that is to say that atoms are well separated with respect to their size. At very high number density, $10^{26}/m^3$ the model may break down. See references to the Mott Transition for further information.

nullFieldIonization Parameters

ionizationKind (*optional*, *default = builtIn*)
: The ionization rate to be used in the interaction. Possible values are `builtIn`, `averagedADK`, `DCADK` and `userDefinedFunc`. See below for required parameters for each choice.

input (*string*)
: This specifies the name of the **Fluid** block used to described the neutral gas. For more information on the kinds of allowed neutral Gases, please see *Working with neutralGas Fluids and the gasKind Parameter*.

electrons (*string*)
: This specifies the name of the **Species** block representing the electron product of the ionization.

ions (*string*)
: This specifies the name of the **Species** block representing the ion product of the ionization.

polarizationFlag (*integer*)
: When the electric field magnitude is approximately constant in a Vorpal time step, `1` is the correct choice; i.e. any time the total electric field is fully time-resolved. Use `0` for the case where you are modeling a linearly polarized electric field, but the time step is larger than the oscillation period. This might occur, for example, when using an envelope model for the laser pulse. This should be applied when the dt of Vorpal is much larger than the period of the field in question.

frequency (*real*)
: This parameter is used in the initialization process; the frequency of a laser pulse if there is such a pulse (the default scenario when using field ionization). If a static field is used, this parameter should be set to 0.

builtIn Parameters

Please see *Types of collisions* for the available `builtIn` gases.

userDefinedFunc Parameters

If the `userDefinedFunc` ionizationKind type is used, the following parameters must be set:

OAFunc (*required*)
: Effective field ionization rate as a function of electric field E (V/m). If not provided, built in rates are used.

  Please see *OAFunc Block* for more information on the OAFunc block.

averagedADK or DCADK Parameters

If the `averagedADK or DCADK` ionizationKind type is used, the following parameters must be set:

energy (*real*, *required*)
: The ionization potential in eV.

Example nullFieldIonization Block

```
<NullInteraction FluidIonizationCs>
  kind = nullFieldIonization
  input = CsNeutralGas
  electrons = electrons
```

(continues on next page)

(continued from previous page)

```
  ions = Cs1
  polarizationFlag = 1
  frequency = 1.e15
</NullInteraction>
```

3.14.4 Using Cross Section Data

Working with neutralGas Fluids and the gasKind Parameter

A Fluid block of `kind=neutralGas` provides a static background gas density. The block must also have the parameter `gasKind` and an InitialCondition of component 0 to give the gas density. The `gasKind` is required in all cases, otherwise an exception is thrown.

How this `gasKind` effects cross-section calculations depends on which interaction is being used. They are summarized in following:

`impactElastic`

If crossSection = builtIn, the only value of `gasKind` available is Xe. For all other `gasKind`, the cross-section is 0.

If crossSection = eedl, `gasKind` and the user supplied EEDL data file should match. If gasKind and EEDL data file do not match, Vorpal still runs, but the results are unphysical.

If crossSection = functionDefined, the user provides an OAFunc that defines the cross-section. In this case, `gasKind` does not set the mass, charge, ionization threshold, or other relevant properties, but it is still validated by the Composer. Thus, it must at least be set to a value that is normally accepted as a `gasKind`.

`impactExcitation``

If crossSection = builtIn, the only value of `gasKind` available is Xe. For all other `gasKind`, cross-section is 0.

If crossSection = eedl, `gasKind` and the user supplied EEDL data file should match. If gasKind and EEDL data file do not match, Vorpal still runs, but the results are unphysical.

If crossSection = functionDefined, the user provides an OAFunc that defines the cross-section. In this case, `gasKind` does not set the mass, charge, ionization threshold, or other relevant properties, but it is still validated by the Composer. Thus, it must at least be set to a value that is normally accepted as a `gasKind`.

`impactIonization`

If crossSection = builtIn, available values for `gasKind` for electron impact are H2, He, CO2, O2, N2, Ar, Ne, Xe, XePlus and for proton impact are H, H2, He, CO2, CO, O2, N2.

If crossSection = eedl, `gasKind` and the user supplied EEDL data file should match. If gasKind and EEDL data file do not match, Vorpal still runs, but the results are unphysical.

If crossSection = functionDefined, the user provides an OAFunc that defines the cross-section. In this case, `gasKind` does not set the mass, charge, ionization threshold, or other relevant properties, but it is still validated by the Composer. Thus, it must at least be set to a value that is normally accepted as a `gasKind`.

`impactIonCollisions`

If crossSection = builtIn

> If gasKind is Xe or Ar, cross-section is based on *[MPL+02]*.
>
> For all other gasKind, cross-section is based on *[LS05]*.

If crossSection = functionDefined, the user provides an OAFunc that defines the cross-section. In this case, `gasKind` does not set the mass, charge, ionization threshold, or other relevant properties, but it is still validated by the Composer. Thus, it must at least be set to a value that is normally accepted as a `gasKind`.

No EEDL for ion.

negativeIonDetachment
If crossSection = builtIn, the only available `gasKind` are H, H2. All other gasKind leads to cross-section = 0.

If crossSection = functionDefined, the user provides an OAFunc that defines the cross-section. In this case, `gasKind` does not set the mass, charge, ionization threshold, or other relevant properties, but it is still validated by the Composer. Thus, it must at least be set to a value that is normally accepted as a `gasKind`.

electronAttachment
If crossSection = functionDefined, the user provides an OAFunc that defines the cross-section. In this case, `gasKind` does not set the mass, charge, ionization threshold, or other relevant properties, but it is still validated by the Composer. Thus, it must at least be set to a value that is normally accepted as a `gasKind`.

nullBgAbsorber
If stoppingPower = builtIn, rate values are obtained from TxPhysics. `gasKind` must be one of following, otherwise an exception is thrown: H, He, C, N, O, Na, Al, Si, P, Ar, Fe, Ni, Cu, Ge, Ag, Ba, Os, Pt, Au, Pb, U, Air, Water, Stainless.

nullFieldIonization
A neutralGas Fluid block provides a background gas density field. If crossSection = builtIn, the only available `gasKind` are H, He, Li, Na, Rb, and Cs. All other gasKind leads to cross-section = 0.

OAFunc Cross-Section Interface

OAFunc: Particle collision is an important physical process in simulations of various plasma applications, such as plasma discharges, magnetron sputtering and low temperature plasma processing. Vorpal allows user defined cross-section in the Monte Carlo Interactions Package. This section describes how a user may include thier own, potentially proprietary, cross-section data directly for Monte Carlo interactions in Vorpal.

Import Scattering Cross-Section from an External Text Data File

Vorpal will assume you have arranged your data into two columns. It assumes there is no header information. (This is different to the syntax that was required for some cross-sections in Vorpal 4.2 before the introduction of the new flexible Monte-Carlo infrastructure to the software). The text file should have two columns, separated by a space. The first column would contain either the electron or ion energy in eV, or the electron or ion velocity in ms^{-1}, depending on your choice of `crossSectionVariable = energy` or `crossSectionVariable = velocity` in the parent Interaction block. The second column would contain the cross-section in m^2.

Example of Cross-Section Import from an External Text Data File

This is an example of using an OAFunc function block to define the cross-section of a dissociation reaction. The data is provided through a two-column data file (`dissXSecVel.dat`), which gives the cross-section as a function of velocity.

```
<Interaction ionization>
  kind = binaryDissociation
  electrons = electrons
  intElecMass=9.1093896999999993e-31
  intSpecMass=1.e-27
```

(continues on next page)

(continued from previous page)

```
  outSpec1Mass = 0.2e-27
  outSpec2Mass = 0.8e-27
  speciesName = oxygen
  inSpecies = oxygen0
  outSpecies1 = oxygen1
  outSpecies2 = oxygen2
  outSpeciesEnergyRatio = 1.
  crossSection = functionDefined
  crossSectionVariable = velocity
  <OAFunc crossSectionFunc>
    kind = interpolatedFromFile
    filename = dissXSecVel.dat
  </OAFunc>
  thresholdEnergy = 15.76
</Interaction>
```

Example Text Data File

In the example above, the text data file might look like this, where as described earlier, the first column is velocity in units of ms^{-1}and the second column is the cross-section in units of m^2.

```
0 0
2.29542e+06 0
2.35285e+06 3.455e-12
2.3707e+06 3.45522e-12
2.44366e+06 2.10494e-11
2.5145e+06 4.18906e-11
2.65052e+06 8.67665e-11
2.71597e+06 1.08962e-10
...
1.51103e+07 1.17765e-10
1.56807e+07 1.11636e-10
1.6231e+07 1.06171e-10
1.67633e+07 1.01266e-10
1.72793e+07 9.68342e-11
1.77802e+07 9.28099e-11
1.82674e+07 8.91374e-11
1.8742e+07 8.57709e-11
```

The text file should have two columns, separated by a space. The first column is the cross-section variable, which can be either the kinetic energy of the incident particle in eV or the relative collision velocity magnitude of the incident particle in ms^{-1}. This variable should be consistent with the choice of `crossSectionVariable = energy` or `crossSectionVariable = velocity` in the parent Interaction block. The second column is the cross-section in m^2.

One can generate this data format from any other existing data file. Just keep the two column data and remove all other information. The resulting file can be used directly in above method.

Specifying the Cross-Section as a Function

Alternatively, it may be possible for the user to specify the function that the cross-section obeys. Note: the example below is not a physically correct model, it is simply an example of the input file syntax.

```
<Interaction ionization>
  kind = binaryDissociation
  electrons = electrons
  intElecMass=9.1093896999999993e-31
  intSpecMass=1.e-27
  outSpec1Mass = 0.2e-27
  outSpec2Mass = 0.8e-27
  speciesName = oxygen
  inSpecies = oxygen0
  outSpecies1 = oxygen1
  outSpecies2 = oxygen2
  outSpeciesEnergyRatio = 1.
  crossSection = functionDefined
  crossSectionVariable = energy
  <OAFunc crossSectionFunc>
    kind = expression
    expression = H(x-15.6)*x*1.2e-13
  </OAFunc>
  thresholdEnergy = 15.76
</Interaction>
```

LXcatFile OAFunc

LXcat is an open-access website for collecting, displaying, and downloading electron ans ion scattering cross-sections required for modeling low temperature plasmas. The available databases include electron-neutral and ion-neutral scattering cross-sections for various kinds of gases. There are fourteen databases that include around one hundred different materials, such as Hg, Kr, Mg, BF3, C2OH6 and CF4. For each material, cross-sections of ionization, excitation, as well as many other collisional processes are available at the LXcat website and can be used in PIC simulations or Boltzmann solvers. These data can be either plotted or downloaded in text format. LXcat database can be accessed from the LXcat Website (http://fr.lxcat.net/home/).

Particle collision is an important physical process in simulations of various plasma applications, such as plasma discharges, magnetron sputtering and low temperature plasma processing. Vorpal allows user defined cross-section in the Monte Carlo Interactions Package. In a simulation of a particular material, user can download the cross-section data for this material from LXcat and use it directly for Monte Carlo interactions in Vorpal.

Import scattering cross-section from LXcat data file

To import scattering cross-section from LXcat data file, one needs to download the data for corresponding material. The cross-section data can be found at http://fr.lxcat.net/data/set_type.php. The LXcat website is under constant development and its contents could change without notice. Please refer to the "how to use" section of LXcat website for more details about obtaining the cross-section database.

After the LXcat data for a particular material is downloaded, it can be directly read in any Vorpal Monte Carlo binary interactions that require a cross-section. The LXcat cross-section data is imported via an OAFunc of kind **`LXcatFile`**, specifying the file name and type of collision. In order to use this OAFunc in the **`Interaction`** block, the **`crossSection`** should be set as `functionDefined`, the OAFunc of kind **`LXcatFile`** must be named as `crossSectionFunc`, and the option **`crossSectionVariable`** should be set as `energy`.

LXcatFile OAFunc Parameters

`filename` (*string*)
   Name of the file that contains downloaded LXcat cross-section data.

typeOfCollision (*string*)
The type of the collision. This must exactly match the first line of the LXcat defining block. In the example here, it is **ELASTIC**

```
-------

ELASTIC
Xe
 4.200000e-6
SPECIES: e / Xe
PROCESS: E + Xe -> E + Xe, Elastic
PARAM.:  m/M = 0.0000042, complete set
COMMENT: elastic momentum transfer.
UPDATED: 2012-04-13 18:41:58
COLUMNS: Energy (eV) | cross-section (m2)
-----------------------------
```

Process (*string*)
Name of the target particle species and interaction process. This must exactly match the second line of the LXcat defining block. In the example here, it is **Xe**

```
-------

ELASTIC
Xe
 4.200000e-6
SPECIES: e / Xe
PROCESS: E + Xe -> E + Xe, Elastic
PARAM.:  m/M = 0.0000042, complete set
COMMENT: elastic momentum transfer.
UPDATED: 2012-04-13 18:41:58
COLUMNS: Energy (eV) | cross-section (m2)
-----------------------------
```

Option (*string*, *optional*, *no default value*)
This is an optional parameter and if it is used, it must exactly match the third line of the LXcat defining block. In the example here, it is **4.200000e-6**

```
-------

ELASTIC
Xe
 4.200000e-6
SPECIES: e / Xe
PROCESS: E + Xe -> E + Xe, Elastic
PARAM.:  m/M = 0.0000042, complete set
COMMENT: elastic momentum transfer.
UPDATED: 2012-04-13 18:41:58
COLUMNS: Energy (eV) | cross-section (m2)
-----------------------------
```

Comments (*string*, *optional*, *no default value*)
This is an optional parameter and if it is used, it must exactly match the *COMMENT* line of the LXcat defining block, i.e. user comments and reference information of the LXcat data. In the example here, it is **elastic momentum tranfer.**

```
-------

ELASTIC
Xe
 4.200000e-6
SPECIES: e / Xe
PROCESS: E + Xe -> E + Xe, Elastic
PARAM.:  m/M = 0.0000042, complete set
COMMENT: elastic momentum transfer.
UPDATED: 2012-04-13 18:41:58
COLUMNS: Energy (eV) | cross-section (m2)
-----------------------------
```

`outerBoundsValue` (*string*, *optional*, *default "except"*)
This controls how the OAFunc interpolates output data when the input value is out of bounds of the data file. "except" throws an exception and the code stops running. "null" uses 0.0 whenever the input value is out of bounds. "constant" uses the data at lower or upper bound when the input value is smaller or larger than the lower and upper bounds. If it sets as " " (an empty string), a linear interpolation based on two data points at the boundary will be used to obtain the outbound value, which may generate inaccurate or nonphysical cross-section value.

Example interaction Block using LXcat cross-section

```
<Interaction CO2Excitation>
  kind = impactExcitation
  neutralGas = CO2Gas
  impactSpecies = Electrons
  crossSectionVariable = energy
  crossSection = functionDefined
  <OAFunc crossSectionFunc>
    kind = LXcatFile
    filename = lxcat.dat
    typeOfCollision = EXCITATION
    Process = "CO2 -> CO2*(7.0eV)"
    Option = "7.000e+0 / threshold energy"
    Comments = "COMMENT: Electronic Excitation. ENERGY LOSS =7eV"
    outBoundsValue = "null"
  </OAFunc>
</Interaction>
```

LXcatFile Data Format

For the LXcat data file to work, it must have the proper format. Unfortunately, the LXcat headers and format have been modified since it was implemented in Vorpal. Therefore, a recent download of LXcat data may not work with Vorpal without some minor modifications first.

Vorpal uses the string `-----` (5 dashes in a row, no spaces) to locate the headers and cross-section data. There must be a minimum of 5 dashes starting in the left most column of the file in order for the header to be found.

Upon finding the first instance of `-----`, Vorpal will read the header data. The header data in the .dat file must exactly match the parameters listed in the .pre file.

If headers do not match, you may run across the error:

```
Can not find the required data in OAFunc of kind "LXcatFile"..
```

After locating the header that matches the block in your .pre file, Vorpal will continue its search for the string -----. It will then start reading in the *Energy* and *Cross-Section** data until it reaches another string -----.

Multiple cross-sections may be included in a single .dat file.

An example of data with the proper format is as follows. There must be a set of dashes ----- before the first header. Currently, the LXcat downloads do not include this.

```
-----------------------------


EXCITATION
Xe -> Xe(1s5)
 8.310000e+0
SPECIES: e / Xe
PROCESS: E + Xe -> E + Xe(1s5), Excitation
PARAM.:  E = 8.31 eV, complete set
COMMENT: Excitation XE+e__e + XE*(1s5).
UPDATED: 2009-09-09 21:02:24
COLUMNS: Energy (eV) | cross-section (m2)
-----------------------------
 8.310000e+0  0.000000e+0
 8.470000e+0  2.500000e-22
 8.840000e+0  6.300000e-22
 9.036000e+0  1.027000e-21
 9.380000e+0  8.700000e-22
 9.520000e+0  1.800000e-21
 9.730000e+0  1.026000e-21
 1.049000e+1  1.158000e-21
 1.097000e+1  1.180000e-21
 1.165000e+1  1.160000e-21
 1.220000e+1  1.126000e-21
 1.498000e+1  8.570000e-22
 1.746000e+1  6.360000e-22
 2.005000e+1  4.640000e-22
 2.368000e+1  3.110000e-22
 2.746000e+1  2.170000e-22
 3.336000e+1  1.180000e-22
 3.510000e+1  1.040000e-22
 5.000000e+1  0.000000e+0
-----------------------------


EXCITATION
Xe -> Xe(1s4)
 8.440000e+0
SPECIES: e / Xe
PROCESS: E + Xe -> E + Xe(1s4), Excitation
PARAM.:  E = 8.44 eV, complete set
COMMENT: Excitation XE+e__e + XE*(1s4).
UPDATED: 2009-09-09 21:02:47
COLUMNS: Energy (eV) | cross-section (m2)
-----------------------------
 8.440000e+0  0.000000e+0
 1.509000e+1  3.420000e-21
 1.718000e+1  4.210000e-21
 1.904000e+1  4.770000e-21
 2.185000e+1  5.330000e-21
```

(continues on next page)

(continued from previous page)

```
 2.506000e+1  5.710000e-21
 2.857000e+1  6.000000e-21
 3.374000e+1  6.120000e-21
 4.130000e+1  6.056000e-21
 6.388000e+1  5.390000e-21
 8.546000e+1  4.738000e-21
 1.000000e+2  4.420000e-21
 3.050000e+2  0.000000e+0
-----------------------------

EXCITATION
Xe -> Xe(1s3-1s2-2p)
 9.690000e+0
SPECIES: e / Xe
PROCESS: E + Xe -> E + Xe(1s3-1s2-2p), Excitation
PARAM.:  E = 9.69 eV, complete set
COMMENT: XE+e__e + XE*(1s3+1s2+2p)   extrapolated after 40 eV using ln(energy)/energy.
UPDATED: 2009-09-09 21:03:14
COLUMNS: Energy (eV) | cross-section (m2)
-----------------------------
 9.690000e+0  0.000000e+0
 1.004000e+1  1.057000e-21
 1.060000e+1  2.127000e-21
 1.132000e+1  3.194000e-21
 1.227000e+1  4.304000e-21
 1.318000e+1  4.840000e-21
 1.440000e+1  5.320000e-21
 1.574000e+1  5.560000e-21
 1.674000e+1  5.660000e-21
 1.836000e+1  5.690000e-21
 2.081000e+1  5.630000e-21
 2.577000e+1  5.360000e-21
 3.327000e+1  4.890000e-21
 3.965000e+1  4.550000e-21
 5.000000e+1  3.800000e-21
 1.000000e+2  2.260000e-21
 2.000000e+2  1.300000e-21
 5.000000e+2  6.100000e-22
 1.000000e+3  3.340000e-22
-----------------------------
```

Available materials in LXcat database

There are about one hundred different materials available at LXcat. A short list of some materials that you may encounter in simulations are in the following. Please refer to the LXcat website for a complete list.

- Ar
- BF3
- C, C2
- C2H2, C2H2+, C2H4, C2H6, C2OH6
- C3, C3H4, C3H6, C3H8, C3N
- CCl2F2, CCl4

- CF, CF2, CF4
- CH, CH+, CH2, CH3, CH4
- CHF3, CNH
- CO, CO2, CO2+
- CONH3, COS, CS
- CaF, CaF+
- Cl2
- Cu
- D2
- F2, F2O
- H, H2, H2+
- H2O, H2S, H4C, HBr, HCHO, HCN, HCP, HCl
- He
- Hg
- Kr
- Mg
- N, N2, N2O, NH3, NO, NO2
- Na
- Ne
- O, O2, O3, O-
- PH3
- SF6, SO2
- Si(CH3)4, Si2H6, SiF2, SiH4, SiO
- Xe

EEDL interface

`EEDL`: Particle collision is an important physical process in simulations of various plasma applications, such as plasma discharges, magnetron sputtering and low temperature plasma processing. Vsim allows user defined cross-section in the Monte Carlo Interactions Package. The Evaluated Electron Data Library [article{perkins1991tables], includes data to describe the transport of electrons, as well as the initial generation of secondary particles, such as the primary photon due to bremmsstrahlung, as well as the primary electron due to inelastic scattering and electroionization.

These cross-sections can be downloaded from the International Atomic Energy Agency Nuclear Data Services website. You can download either the complete library or individual evaluations. **Vorpal currently uses the eedl/endl format for the libraries, so please be sure to download the right one.**

The EEDL cross-section database supports minimum energy value from 10 eV (or from the threshold value in inelastic interactions) and maximum up to 100 GeV. In VSim, when used an elastic interaction with the “eedl” option, a constant cross-section value is taken for energy values below 10 eV, i.e. it uses the cross-section value of 10 eV for low energy particles. The EEDL datasets are available for Z=1 to 100 elements.

They work with the following collision types: *impactElastic*, *impactExcitation impactIonization* and *negativeIonDetachment* as described in *MonteCarloInteractions*.

Import scattering cross-section from EEDL data file

```
<Interaction ElecImpElastic>
  kind = impactElastic
  neutralGas     = ArNeutralGas
  impactSpecies = electrons
  crossSection = eedl
  crossSectionDataFile = eedlAr.dat
</Interaction>
```

After the EEDL data for a particular material is downloaded, it can be directly read in any Vsim Monte Carlo interactions of the kinds specified in the interactions table that require a cross-section. In order to use an EEDL cross-section in the **`Interaction`** block, the **`crossSection`** should be set as `eedl`, then the `crossSectionDataFile` should be set to the name of the file you intend to use.

EEDL Data Format

An example set of data is given below in the EEDL/ENDL format.

Note: It is very important to make sure you are using the proper format. Vorpal currently uses the eedl/endl format for the libraries, so please be sure to download the right one.

```
18000  9  0  0.00000e+00 8912015 2  0.00000e+00  0.00000e+00  0.00000e+00
7  0  0  0.00000e+00  0.00000e+00  0.00000e+00  0.00000e+00  0.00000e+00
1.00000e-05 2.49353e+09
1.25893e-05 2.02388e+09
1.58489e-05 1.64269e+09
1.99526e-05 1.33330e+09
2.51189e-05 1.08218e+09
3.16228e-05 8.78354e+08
1.99526e+04 3.02448e-06
2.51189e+04 1.93916e-06
3.16228e+04 1.24298e-06
3.98107e+04 7.96534e-07
5.01187e+04 5.10324e-07
6.30957e+04 3.26874e-07
7.94328e+04 2.09326e-07
.
.
.
1.00000e+05 1.34020e-07
                                                                       1
18000  9  0  0.00000e+00 8912015 2  0.00000e+00  0.00000e+00  0.00000e+00
8  0  0  0.00000e+00  0.00000e+00  0.00000e+00  0.00000e+00  0.00000e+00
1.00000e-05 2.49353e+09
1.25893e-05 2.15264e+09
1.58489e-05 1.85836e+09
1.99526e-05 1.60430e+09
2.51189e-05 1.38498e+09
```

(continues on next page)

(continued from previous page)

```
3.16228e-05 1.19564e+09
3.98107e-05 1.03266e+09
5.01187e-05 8.91897e+08
6.30957e-05 7.70321e+08
7.94328e-05 6.65469e+08
1.00000e-04 5.75282e+08
1.25893e-04 4.97317e+08
1.58489e-04 4.29918e+08
1.99526e-04 3.71654e+08
2.51189e-04 3.21434e+08
3.16228e-04 2.78264e+08
3.98107e-04 2.40892e+08
5.01187e-04 2.08539e+08
6.30957e-04 1.80556e+08
```

If you wish to write your own cross-section data files in EEDL/ENDL format, please refer to *[PC02]* which describes the format. All EEDL data distributed with VSim is freely available to all users.

3.15 History

3.15.1 History

Top-level block for recording data from a Vorpal simulation. You can use History blocks for Vorpal to calculate and record data about fields and particles in a simulation.

History uses the kind parameter to determine which specific data to collect.

History data is useful for diagnostics and for generating particle position data, as well as determining the basis for performing current and field parameter adjustment during the simulation.

Throughout a simulation run, Vorpal writes the History data to a separate HDF5 file. The History data file has the file name suffix _History.h5

When a simulation is started or restarted, Vorpal looks for an existing History file. If Vorpal finds a History file, Vorpal appends data to contents of that existing file. If Vorpal does not find a History file, Vorpal creates a new file.

You can create tagged particles that can then be used to generate particle position data by using a History block. Once you have created a tagged species, you can use History blocks to follow particle trajectories based on tag values. Learn how to tag particle species in the *speciesTrackTag*.

You can use Feedback History blocks to adjust parameters based on measured time dependent quantities. For example, you could automatically adjust the current in a simulation to generate a desired voltage. Learn how to use Feedback History blocks in *historySTFunc*.

All Tensor Histories can also be used for feedback, in conjuction with UserFuncs of kind=historyFunc (see *historyFunc*).

When Histories Collect Data

Generally, histories collect data at the end of each time step, after all other Vorpal objects have been updated.

However, histories can also collect data after the initialization of Vorpal objects (just before the first time step). Some histories do not do this; all *Tensor Histories* do collect data just after initialization by default. To avoid collecting data after initialization, simply add the following to the <History> block:

```
<Expression applySteps>
  expression = (n > 0)
</Expression>
```

History Parameters

History block parameters might include:

kind (*string or string vector*)
: Determines type of Field or Species data to be collected in the History data file during the simulation run. The available field and particle species kinds are detailed in descriptions in the following pages. In addition to specifying the kind, for each History block you must also specify one of the field, fields, or species keywords (see below).

keyword (*string or string vector*)
: Indicates species or field(s) for use with specified `History` kind.

field (*string*)
: Indicates the field for a History kind that reports data for a single field. Specify a field reference in a History block by using the name of the **EmField** followed by a dot (.) then the name of the field subobject. Example: `field = multiField.elecField`

fields (*string vector*)
: Indicates the fields for a History kind that reports data for multiple fields. Specify references to fields in a History block by using the name of the EmField followed by a dot (.) then the name of the field subobject. Example: `fields = [multiField.elecField multiField.magField]`

species (*string vector*)
: Indicates particle species type for a History kind that reports data for particle species. Example: `species = [electrons]`

Field History

- *acceleratingVoltage*
- *bLoop*
- *EMfieldEnergy*
- *fieldAtCoords*
- *fieldAverage*
- *fieldAtIndices*
- *fieldEnergy*
- *fieldOnSemiCircle*
- *fieldOnMovingPlane*
- *fieldOnLine*
- *fieldPoyn*
- *integratedSurfaceFieldSquared*
- *kirchhoffSurfaceIntegral*
- *peakSurfaceFieldMagnitude*

- *pseudoPotential*
- *timeAverage*

Particle History

- *speciesAbsPtclData*
- *speciesCurrAbs*
- *speciesCurrEmit*
- *speciesDiag*
- *speciesEnergy*
- *speciesEngyAbs*
- *speciesMomen*
- *speciesNumberOf*
- *speciesNumPhysical*
- *speciesRmsDistToAxis*
- *speciesRmsMomen*
- *speciesRmsPosition*
- *speciesTrackTag*
- *speciesDataOnPlane*

History Operation Histories

- *binaryOperation*
- *unaryOperation*

Feedback Histories

- *feedbackDesired*
- *feedbackMeasured*

Scalar Histories

- *scalarValue*

Tensor Histories

- *fieldArray*
- *cellFuncHist*
- *functionOfTime*
- *speciesAbsPtclData2*

- *speciesBinning*

Tensor Histories are somewhat different from other Histories. They comprise select Histories that store records as a tensor (or multi-dimensional array). A record is a single tensor (array), and the tensor history is a 1D array of records. A record could by the value of a field at a certain time, or a sub-array of a field at a certain time, or the properties of a particle absorbed, etc.

The values stored in Tensor Histories can be accessed (during a simulation) through UserFuncs (see *historyFunc*). to provide feedback or to perform further calculations from the results of Tensor Histories.

Briefly, kinds of tensor histories are:

- `fieldArray`: Stores a subarray of a field (at desired time-steps)
- `functionOfTime`: Stores a function of time; generally this function involves the values of other tensor histories. For example, it might multiply two histories together.
- `cellFuncHist`: Calculates line, surface, and volume integrals, as well as finding field maxima, etc.
- `speciesAbsPtclData2`: Records data for particles absorbed on a boundary–either data for each particle, or statistics (averages, etc.). This history is similar to *speciesAbsPtclData*, but stores multiple data per particle in a single history.
- `speciesBinning`: Creates an array of bins, with each binning holding a vector, and for each particle, adds an amount to a bin chosen according to the particle properties. This history allows general particle distributions (e.g., velocity distributions) to be stored.

Besides the attribute specific to each kind, they take the following attributes (which should mostly be ignored):

`syncPeriod` (non-negative integer, optional, default = 0): Reduces memory use (for parallel simulations with multiple processes per node) by telling the processors to communicate their local histories (to form the global history) every `syncPeriod` updates. (E.g., if `applyPeriod = 10`, so the history adds a record every 10 time-steps, and `syncPeriod = 3`, then the the processors will communicate every 3 updates, which is every 30 time-steps.) With the default value (0), processors will communicate only when necessary, as at dump time.

This option may reduce memory use in parallel simulations with multiple processes on a single node. Roughly, it reduces memory use by a factor a bit less than the number of processes per node. For example, if there are 12 processes per node (all of which share the same memory), and the simulation runs out of memory after 100 history updates, then setting `syncPeriod` to a number less than 100 will allow approximately 1000 updates (a bit less than 100 times 12). After this, the only way to avoid running out of memory is to dump (by increasing the simulation's dumpPeriodicity) the data, which removes most past history records from memory.

Communication takes time, even in the limiting case of very little data, so one generally wants to avoid communicating small amounts of data at every time step. Histories that update at every time step should use larger `syncPeriod`. On the other hand, large histories that update infrequently should use small `syncPeriod`, perhaps 1 to 10.

`maxMemChunkSize` (integer, optional): This option should be rarely used. When used, it should be set to a rough size (in bytes) of an amount of memory that will always be available. The history will use *chunking* to try to avoid temporary use of chunks of memory much (a few times) larger than maxMemChunkSize. For example, when the history is dumped to disk (for which it must create a temporary array to store the history), it will dump the data in chunks of size (on the order of) maxMemChunkSize. This also affects temporary memory use when communicating histories among different ranks in a parallel simulation. Probably this should be at least 100 MB, and at least 1 kB times the number of processors in a parallel simulation.

`` option:`minRecordsToKeep `` (integer, optional): This option should usually not be needed, since Vorpal can almost almost always figure it out. This option can be set to a minimum number of history records that Vorpal will keep (e.g., for HistoryFuncs of `kind=historyFunc`). In particular:

Vorpal deletes history records that have been saved to disk (to save memory), but will retain at least minRecordsToKeep; and upon restart, Vorpal does not reload the entire history, but only minRecordsToKeep records. A situation that might require specification of minRecordsToKeep: if a simulation is run (with no HistoryFunc) and dumped, the input file is modified to add a HistoryFunc, and then the simulation is restarted from the dump. In this case, Vorpal could not figure out, from the first run, how many past records the second run would require.

3.15.2 Field History

acceleratingVoltage

Works with VSimBase, VSimEM, VSimPD, VSimPA, and VSimMD licenses.

Indicates Vorpal should perform the diagnostic activity of measuring the accelerating voltage received by an electron traveling at a fixed velocity across a gap in a cavity structure. This history uses the definition of accelerating voltage found in *[Pad09]*. This hiStory is useful for RF cavity simulations.

acceleratingVoltage Parameters

`field` (*string vector*, *required*)
The oscillating electric field that accelerates the test electron.

`startPosition` (*double vector*, *required*)
The starting point for the test electron.

`endPosition` (*double vector*, *required*)
The ending point for the test electron.

`velocity` (*double vector*, *required*)
The velocity of the test electron. This is usually set to the speed of light.

acceleratingVoltage History Example

```
<History accelVoltage>
  kind = acceleratingVoltage
  field = multiField.elecField
  startPosition = [0. 0. 0.]
  endPosition = [0. 0. CYL_LEN]
  velocity = LIGHTSPEED
</History>
```

bLoop

Works with VSimBase, VSimEM, VSimPD, VSimPA, and VSimMD licenses.

Vorpal computes the enclosed current in Amperes. **bLoop** does not integrate along a circle. **bLoop** performs quadrature to compute the B loop 1/mu_0 x S B . dl, which equals the enclosed current (i.e. the original Ampere's circuital law). Displacement current is neglected. The unit is Ampere.

Let the B loop go through cell centers so we only have to interpolate the B components between two values (Yee mesh is assumed).

The point on the loop closest to the origin is thus at

```
 LowerBounds+(1/2,1/2,1/2)=(xmin,ymin,zmin)


and the point on the loop furthest from the origin is at:
```

```
 UpperBounds-(1/2,1/2,1/2)=(xmax,ymax,zmax)


We'll divide the loop into 6 segments:
```

1. +x direction, from (xmin, ymin, zmin) to (xmax, ymin, zmin)
2. +y direction, from (xmax, ymin, zmin) to (xmax, ymax, zmin)
3. +z direction, from (xmin, ymax, zmin) to (xmax, ymax, zmax)
4. -x direction, from (xmax, ymax, zmax) to (xmin, ymax, zmax)
5. -y direction, from (xmin, ymax, zmax) to (xmin, ymin, zmax)
6. -z direction, from (xmin, ymin, zmax) to (xmin, ymin, zmin)

For NDIM=1, the loop then consists of segments 1 and 4, a back and forth line integral that exactly cancels out, so no need for quadrature.

For NDIM=2, the loop then consists of segments 1, 2, 4 and 5.

For NDIM=3, the loop consists of all 6 segments.

To maximize code reuse we'll then put conditionals around segments 3 & 6.

bLoop Parameters

Use the **bLoop History** kind with the following parameters:

`field` (*string*, *required*)
Indicates the field for a **History** kind that reports data for a single field. Currently, this field should be a magnetic field on the Yee mesh.

`lowerBounds` (*integer vector*, *required*)
Set loop corner closest to origin.

`upperBounds` (*integer vector*, *required*)
Set loop corner furthest from origin.

bLoop History Example

```
<History BLoop>
    kind = bLoop
    field = myEmField.magField
    lowerBounds = [5 5 12]
    upperBounds = [25 25 12]
</History>
```

EMfieldEnergy

Works with VSimEM, VSimPD, and VSimMD licenses.

Indicates Vorpal should calculate the total energy of the electromagnetic fields listed in the fields parameter in the special case where a **GridBoundary** is present. The fields in cut cells created by the **GridBoundary** are given special treatment to provide a more accurate result.

EMfieldEnergy Parameters

`fields` (*string vector*, *required*)
Indicates the fields for a **History** kind that reports data for a multiple fields. For this history there should be exactly two fields where the first is the electric field and the second is the magnetic field.

`gridBoundary` (*string*, *required*)
The **GridBoundary** used in the energy calculation.

EMfieldEnergy History Example

```
<History storedEnergy>
  kind = EMfieldEnergy
  fields = [multiField.elecField multiField.magField]
  gridBoundary = cylindricalCav
</History>
```

fieldAtCoords

Works with VSimBase, VSimEM, VSimPD, VSimPA, and VSimMD licenses.

Indicates point at which to record NDIM . Describes where in the system the data is to be taken (i.e. the physical coordinate, measured in meters) by using the position (float vector) parameter. Use with the following parameters:

fieldAtCoords Parameters

`field` (*string*, *required*)
Indicates the field for a **History** kind that reports data for a single field.

`position` (*float vector*, *required*)
Specify a physical coordinate where the field value is recorded.

`components` (*integer vector*, *optional*)
Specify which components of the field are recorded. If not specified, all available components of the field are written out.

fieldAtCoords History Example

```
<History  fieldAtCoords>
    kind = fieldAtCoords
    field = myField.testField
    position = [0.5 0.5 0.5]
```

(continues on next page)

(continued from previous page)

```
    components = [0 1 2]
</History>
```

fieldAtCoords History Macro

addFieldAtCoordHist(name,fieldName,comp,px,py,pz)

fieldAverage

Works with VSimBase, VSimEM, VSimPD, VSimPA, and VSimMD licenses.

Indicates Vorpal should calculate the average of a field over a specified region. Specify the region with the `upperBounds` and `lowerBounds` parameters, or a Slab.

fieldAverage Parameters

`field` (*string*, *required*)
Indicates the field for a **History** kind that reports data for a single field.

`components` (*integer vector*, *optional*)
Specify the average value of which components of the field are recorded. If not specified, all available components of the field are written out.

`region` (*slab object*, *required*)
A slab region defined by lowerBounds/upperBounds.

fieldAverage History Example

```
<History fieldAverage>
    kind =  fieldAverage
    field = myField.testField
    components = [0 1 2]
    <Slab region>
        lowerBounds = [4 4 4]
        upperBounds = [7 7 7]
    </Slab>
</History>
```

fieldEnergy

Works with VSimBase, VSimEM, VSimPD, VSimPA, and VSimMD licenses.

Indicates Vorpal should calculate the total energy of fields listed in the fields parameter. **fieldEnergy** may have `lowerBounds` and `upperBounds`. By default, if you do not specify boundaries for **fieldEnergy**, Vorpal calculates the EM field energy of the entire simulation space.

fieldEnergy Parameters

lowerBounds (*integer vector*)
: **fieldEnergy** may have `lowerBounds` and `upperBounds` to measure the field energy only in the region defined by `lowerBounds/upperBounds`. By default, if you do not specify boundaries for **fieldEnergy**, Vorpal calculates the field energy of the entire simulation space.

upperBounds (*integer vector*)
: **fieldEnergy** may have `lowerBounds` and `upperBounds` to measure the field energy only in the region defined by `lowerBounds/upperBounds`. By default, if you do not specify boundaries for **fieldEnergy**, Vorpal calculates the field energy of the entire simulation space.

fields (*string vector*, *required*)
: Indicates the fields for a **History** kind that reports data for a multiple fields. Depends on the following parameter `singleField`; the user must specify either one or two fields for energy calculation.

singleField (*string*, *elecField or magField*)
: If `singleField = elecField`, a single electric field must be specified via fields. The energy of this electrostatic field is calculated and recorded.

  If `singleField = magField`, a single magnetic field must be specified via fields. The energy of this magnetostatic field is calculated and recorded.

  If *`singleField`* is not specified. The energy of electromagnetic field is calculated and recorded. Exactly two fields must be specified in the fields keyword. The first one is the electric field. The second one is the associated magnetic field.

fieldEnergy History Example

Using the standard two field option.

```
<History  myFieldEnergy>
    kind = fieldEnergy
    fields = [multiField.elecField  multiField.magField]
</History>
```

Using the single field option.

```
<History  ElecFieldEnergy>
    kind = fieldEnergy
    lowerBounds = [0  0  0]
    upperBounds = [NX/2 NY/2 NZ/2]
    singleField = elecField
    fields = [multiField.elecField]
 </History>
```

fieldAtIndices

Works with VSimBase, VSimEM, VSimPD, VSimPA, and VSimMD licenses.

Similar to **fieldAtCoords**, but the point is specified in terms of grid indices (as opposed to physical coordinates).

(To store field values within a sub-array, see *fieldArray*.)

fieldAtIndices Parameters

Use the **fieldAtIndices History** kind with the following parameters:

`field` (*string*, *required*)
: Indicates the field for a **`History`** kind that reports data for a single field.

`point` (*integer vector*, *required*)
: Specify a grid indices where the field value is recorded.

`components` (*integer vector*, *optional*)
: Specify which components of the field are recorded. If not specified, all available components of the field are written out.

fieldAtIndices History Example

```
<History fieldAtIndices>
    kind = fieldAtIndices
    field = myField.testField
    point = [3 3 3]
    components = [0 2]
</History>
```

fieldMomen

Works with VSimBase, VSimEM, VSimPD, VSimPA, and VSimMD licenses.

Indicates Vorpal should calculate the total momentum of EM fields listed in the fields parameter. **fieldMomen** may have `lowerBounds` and `upperBounds`. By default, if you do not specify `lowerBounds` and `upperBounds` for **fieldMomen**, Vorpal calculates the momentum of the entire simulation space.

fieldMomen Parameters

`lowerBounds` (*integer vector*)
: **fieldMomen** may have `lowerBounds` and `upperBounds` to measure the field momentum only in the region defined by `lowerBounds`/`upperBounds` parameter. **fieldMomen** may have `lowerBounds` and `upperBounds`. By default, if you do not specify `lowerBounds` and `upperBounds` for **fieldMomen**, Vorpal calculates the momentum of the entire simulation space.

`upperBounds` (*integer vector*)
: **fieldMomen** may have `lowerBounds` and `upperBounds` to measure the field momentum only in the region defined by `lowerBounds`/`upperBounds` parameter. **fieldMomen** may have `lowerBounds` and `upperBounds`. By default, if you do not specify `lowerBounds` and `upperBounds` for **fieldMomen**, Vorpal calculates the momentum of the entire simulation space.

`fields` (*string vector*, *required*)
: Indicates the fields for a History kind that reports data for a multiple fields. Exactly two fields must be specified in the fields keyword. The first one is the electric field. The second one is the associated magnetic field.

fieldMomen History Example

```
<History myFieldMomen>
    kind = fieldMomen
    fields = [multiField.elecField multiField.magField]
</History>
```

fieldOnSemiCircle

Works with VSimBase, VSimEM, VSimPD, VSimPA, and VSimMD licenses.

Indicates Vorpal should perform the diagnostic activity of tracking any grid field along a circle. The **fieldOnSemiCircle** diagnostic dumps the E or B field along a circle at equally spaced points. At each point the field will be interplated from values in neighboring cells. **fieldOnSemiCircle** can be used for 2D or 3D simulations. The circle is defined by its center, normal (vector) and radius. In 2D simulations, the circle lies in the X-Y plane and a normal should not be specified. If you specify a circle which extends outside the simulation domain, points along the circle which are outside the simulation domain will be ignored. Caution: if the circle has no points inside the simulation domain, an error is generated.

fieldOnSemiCircle Parameters

`fields` (*string vector*, *required*)
Indicates the fields for a **`History`** kind that reports data. User can specify multiple fields, but must specify at least one field. If multiple fields are specified, the values of each component of all these fields are recorded.

`numComponents` (*integer*, *default value = 1*)
Sets how many number of components of each field to record. By default, only the first component is recorded.

`center` (*float vector*, *optional*)
Sets the coordinate of the center of the circle. By default, the center of the simulation domain.

`radius` (*float*, *optional*)
Sets the radius of the circle. By default, one third the length of the smallest dimension.

`normal` (*float vector*, *optional*)
For 3D only, a vector which is perpendicular to the plane of the circle. By default this vector points in the z-direction, so default 3D circles will lie in an X-Y plane with constant z value.

`numPts` (*integer*, *default value = 8*)
Sets the number of equally spaced interpolation points around the circle. If the circle extends outside the simulation domain the number of values returned will be less than numPts.

fieldOnSemiCircle History Example

```
<History fieldOnSemiCircle>
    kind = fieldOnSemiCircle
    fields = [multiField.elecField multiField.magField]
    numComponents = 3
</History>
```

fieldOnMovingPlane

Works with VSimBase, VSimEM, VSimPD, VSimPA, and VSimMD licenses.

Indicates Vorpal should perform the diagnostic activity of tracking any grid fields (electric magnetic or charge plus current) along a line (2D) or plane (3D) of cells that is perpendicular to one of the Cartesian axes and that can be moving along this axis. The **fieldOnMovingPlane** diagnostic dumps the global grid cells defining the line in the simulation domain. In 2D, **fieldOnMovingPlane** dumps X and Y. In 3D, **fieldOnMovingPlane** dumps X, Y, and Z. In addition, **fieldOnMovingPlane** dumps the initial position and speed of the plane. You can use **fieldOnMovingPlane** for simulations only 2D or higher.

fieldOnMovingPlane Parameters

fields (*string vector*, *required*)
Indicates the fields for a History kind that reports data. User can specify multiple fields, but must specify at least one field. If multiple fields are specified, the values of each component of all these fields are recorded.

numComponents (*integer*, *default value = 1*)
Sets how many number of components of each field to record. By default, only the first component is recorded.

direction (*integer*, *optional*, *default value = 0*)
Direction in which the plane (or line) is moving. `0`: x, `1`: y, `2`: z.

initialLoc (*float*, *required*)
Location, on the `direction` axis, of the plane (or line) at `t=0`.

speed (*float*, *required*)
Velocity of the plane (or line), in m/s, along the `direction` axis.

cutDir (*integer*, *default value = 0.0*)
Axis perpendicular to the line.

cutVal (*float*, *optional*, *default value = 0.0*)
Location of line along axis perpendicular to the line.

fieldOnMovingPlane History Example

```
# Specify history diagnostics
<History fieldAtMovingPlaneWaist>
    kind = fieldOnMovingPlane
    fields = [myField.elecField myField.magField]
    direction = 0
    initialLoc = HISTORY_LOC
    speed = $ -1. * VX_BOOST $
    numComponents = 3
</History>
```

fieldOnLine

Works with VSimBase, VSimEM, VSimPD, VSimPA, and VSimMD licenses.

Indicates Vorpal should perform the diagnostic activity of tracking any grid fields (electric magnetic or charge plus current) along a line of cells that is parallel to one of the Cartesian axes. The **fieldOnLine** diagnostic dumps the global grid cells defining the line in the simulation domain. In 2D, fieldOnLine dumps X and Y. In 3D, **fieldOnLine** dumps X, Y, and Z. In Vorpal, a line of cells is specified as a volume in which only one of the dimensions can have multiple cells. The concept of a line of cells does not imply that all dumped components are located on a straight line.

fieldOnLine Parameters

fields (*string vector*, *required*)
: Indicates the fields for a History kind that reports data. User can specify multiple fields, but must specify at least one field. If multiple fields are specified, the values of each component of all these fields are recorded.

lowerBoundsupperBounds (*integer vector*, *required*)
: Defines physical cell indices parallel to one of the Cartesian axes. All components of the given fields are recorded along each grid cell of this line of cells. The values that are dumped for each component always correspond to the values computed by the field solver routine. For example, in an Electromagnetic simulation, this diagnostic will return the values defined by the Yee Cell.

fieldOnLine History Example

```
# Specify history diagnostics
<History SeveralFieldsOnLine>
    kind = fieldOnLine
    fields = [multiField.elecField SumRhoJ]
    lowerBounds = [0 2 3]
    upperBounds = [NX 3 4]
</History>
```

fieldPoyn

works with VSimBase, VSimEM, VSimPD, VSimPA, and VSimMD licenses.

indicates Vorpal should calculate the integrated Poynting vector (energy flux) through the area defined by the *upperBounds* and *lowerBounds* of two fields.

fieldPoyn Parameters

lowerBounds
: Defines a surface area. Vorpal calculates the integrated Poynting vector through this surface area.

upperBounds
: Defines a surface area. Vorpal calculates the integrated Poynting vector through this surface area.

fields (*string vector*, *required*)
: Indicates the fields for a History kind that reports data for a multiple fields. Exactly two fields must be specified in the fields keyword. The first one is the electric field. The second one is the associated magnetic field.

fieldPoyn History Example

```
<History fieldPoynPt>
    kind = fieldPoyn
    lowerBounds = [1 1 1]
    upperBounds = [1 3 3]
```

(continues on next page)

(continued from previous page)

```
    fields = [multiField.elecField multiField.magField]
</History>
```

integratedSurfaceFieldSquared

Works with VSimBase, VSimEM, VSimPD, and VSimMD licenses.

Integrates the magnitude squared of a field over a grid boundary, or part of a boundary, and stores the result in a history. Units recorded by this history are dependent on the units used for the field for which the history is taken. That is, when you point this history at an electric field, for example, rather than a magnetic field or another arbitrary user-defined field, the units will be registered in the same units as the selected field squared times meters squared.

One common use for **integratedSurfaceFieldSquared** is to compute the power dissipated due to heating the boundary of a device. In the example that follows, the magnitude of the magnetic field (H) is squared and integrated over the surface of the gridBoundary. Since Vorpal technically uses the magnetic induction (B) the `coefficient` parameter is used to convert the B field to the H field. When the surface integral of the H field squared is multiplied by the resistance at the surface of the conductor, the result is the total power loss due to surface resistance. Therefore, provided that you have an estimate of the resistance in the conductor, you can use this code block to compute the total power loss due to ohmic heating at the boundary.

integratedSurfaceFieldSquared Parameters

alpha (*real*, *default value = 1.0*)
Defines the offset distance from the boundary where the field will be measured. Interpolation on the boundary is effected by values in the boundary itself so a more accurate value for the field is achieved by computing the field slightly off the surface. The normal distance from the surface where the field is measured is given by alpha times the cell hypotenuse.

coefficient (*real*, *optional*, *default = 1.0*)
Coefficient by which the surface integral can be multiplied.

field (*string*, *required*)
Name of the field whose magnitude is computed at each surface point.

gridBoundary (*string*, *required*)
Name of the gridBoundary object over which the integral is performed.

lowerBounds (*integer vector*)
Lower bounds of the region for which the magnitude squared calculation is to be integrated.

upperBounds (*integer vector*)
Upper bounds of the region for which the magnitude squared calculation is to be integrated.

integratedSurfaceFieldSquared History Example

```
<History integralOfMagFieldSquared>
  kind = integratedSurfaceFieldSquared
  alpha = 1.0
  coefficient = $1/(MU0*MU0)$ # Converts B to H
  gridBoundary = cylindricalCav
```

(continues on next page)

(continued from previous page)

```
  field = multiField.magField
</History>
```

kirchhoffSurfaceIntegral

Works with VSimEM license.

Performs a near to far field transformation of the electric field at a specified time and on a specified sphere.

kirchhoffSurfaceIntegral Parameters

`fields` (*string vector*, *required*)
The fields used in the transformation. The ordering E, dEdx, dEdy, dEdz, dEdt is assumed.

`lightSpeed` (*float*, *optional*)
Speed of light in medium.

`dtFar` (*float*, *optional*)
Time seperation of far field time points.

`numSurfacePoint` (*non-negative integer*, *optional*)
Number of integration points around Kirchhoff circle.

`timeInterval` (*float vector*, *required*)
Time interval over which to compute far field.

`position` (*float vector*, *optional*)
Position of far field point.

`centerSphere` (*float vector*, *required*)
Center of Kirchhoff sphere.

`radiusSphere` (*float*, *required*)
Radius of Kirchhoff sphere.

`haveNormalDeriv` (*0 or 1*, *optional*)
Set to read normal derivatives of fields on Kirchhoff sphere.

`radiusFarSphere` (*float*, *optional*)
Radius of far sphere.

`numTheta` (*non-negative integer*, *optional*)
Number of polar angles on the far sphere.

`numPhi` (*non-negative integer*, *optional*)
Number of azimuthal angles on the far sphere.

kirchhoffSurfaceIntegral History Example

```
# Far field parameters
# Radius of the Kirchhoff sphere
$ RS = 1.0
# Distance of far field from center of sphere
$ DFF = 10.0
# Far field time interval
```

(continues on next page)

(continued from previous page)

```
$ TFFMIN = (DFF + RS) / LIGHTSPEED
$ TFFMAX = TFFMIN + 1.0 / FREQ_ANT
$ NTFAR = 10
$ DTFAR = (TFFMAX - TFFMIN) / NTFAR
# Number of integration points around
#    a circular slice of the Kirchhoff sphere
$ NS = 10

<History farField>
  kind = kirchhoffSurfaceIntegral
  # The ordering E, dEdx, dEdy, dEdz and dEdt is assumed by kirchhoffSurfaceIntegral
  fields = [emField.centerE emField.dEdx emField.dEdy emField.dEdz emField.dEdt]
  radiusFarSphere = DFF
  numTheta = 9
  numPhi = 18
  dtFar = DTFAR
  # For a single far field time at TFFMIN, we use a timeInterval of length 0
  timeInterval = [TFFMIN TFFMIN]
  # The Kirchhoff sphere must be contained entirely within the computational domain
  centerSphere = [0.0 0.0 0.0]
  radiusSphere = RS
  numSurfacePoint = NS
</History>
```

kirchhoffSurfaceIntegral History hdf5 File Structure

The hdf5 datset of the kirchhoff surface integral is structured as follows. The rows represent each timestep of the simulation.

The columns go in order of numPhi and then numTheta at TFFMIN. Then numPhi and numTheta at TFFMAX. If NTFAR is greater than 1, there will be more sets of phi/theta points in between TFFMIN and TFFMAX. This yields a total number of columns of (numPhi+numTheta)*(NTFAR+1). The number of integration points (NS) is used in the calculation of the values of the history, but do not impact it's size.

See also *Add Far Field History Macro* to implement a kirchhoffSurfaceIntegral history using macros.

peakSurfaceFieldMagnitude

Works with VSimBase, VSimEM, VSimPD, and VSimMD licenses.

Computes the peak magnitude of a field over a grid boundary. The value is stored in a history along with the location. The first element of the history is the peak magnitude of the field, The units of this peak magnitude are expressed in the same the units of the field itself. The next three values represent the x, y and z positions of the peak value, which are expressed in meters. You can use **peakSurfaceFieldMagnitude** to compute the peak surface current observed on the surface of a device.

In the example that follows for the **peakSurfaceFieldMagnitude** field history kind, this block computes the maximum magnetic field (H) magnitude over a surface specified by `gridBoundary`, and records the value along with its position on the surface. The B field in Vorpal is converted to H by using the `coefficient` parameter. Since the peak H field on a surface corresponds to the peak in the surface current this use of the history can help identify potential locations on a SRF cavity surface where the quenching of the superconductor may occur.

peakSurfaceFieldMagnitude Parameters

alpha (*real*, *default value = 1.0*)
Defines the offset distance from the boundary where the field will be measured. Interopolation on the boundary is effected by values in the boundary itself so a more accurate value for the field is achieved by computing the field slightly off the surface. The normal distance from the surface where the field is measured is given by alpha multiplied by the cell hypotenuse.

coefficient (*real*, *optional*, *default = 1.0*)
Coefficient by which the peak magnitude can be multiplied.

field (*string*, *required*)
Name of the field whose magnitude is computed at each surface point.

gridBoundary (*string*, *required*)
Name of the gridBoundary object over which the peak will be determined.

lowerBounds (*integer vector*)
Lower bounds of the region for which the peak magnitude calculation is to be performed.

upperBounds (*integer vector*)
Upper bounds of the region for which the peak magnitude calculation is to be performed.

peakSurfaceFieldMagnitude History Example

```
<History peakMagneticField>
  kind = peakSurfaceFieldMagnitude
  alpha = 1.0
  coefficient = $1/MU0$ # Converts B to H
  gridBoundary = cylindricalCav
  field = multiField.magField
</History>
```

pseudoPotential

Works with VSimBase, VSimEM, VSimPD, VSimPA, and VSimMD licenses.

Calculates the pseudo-potential difference, in Volts, between two points, the reference point and the measure point. **pseudoPotential** performs quadrature to compute the difference in pseudo potential:

$$\Phi_{pseudo} = \int_a^b \mathbf{E} \cdot d\mathbf{l}$$

between the points a (referencePoint) and b (measurePoint). The vector potential is assumed time invariant, hence the *pseudo*. The unit is Volt.

pseudoPotential Parameters

field (*string*, *required*)
Name of the magnetic field used to compute the surface currents. Exactly one field name should be specified. For potential calculation purpose, this field should be an electric field.

referencePoint (*integer vector*, *required*)
The grid indices of the reference point.

measurePoint (*integer vector*, *required*)
The grid indices of the measured point.

pseudoPotential History Example

```
<History PseudoPotential3>
    kind = pseudoPotential
    field = myEmField.elecField
    referencePoint = [8 2 5]
    measurePoint = [3 8 5]
</History>
```

timeAverage

timeAverage: Works with VSimBase, VSimEM, VSimPD, VSimPA, and VSimMD licenses.

Incidates Vorpal should record the value of a specified field and take the time average. **timeAverage** require both a *history* parameter and a *timeWindow* parameter. This *history* time averages the output, in the form of a field, from another *history*.

timeAverage Parameters

history (*string*, *required*)
Indicates the `history` to reference whose output is to be time averaged.

timeWindow (*float*, *required*)
Specifies the time duration to average over.

timeAverage History Example

```
<History timeAvePseudo>
    kind = timeAverage
    history = potential
    timeWindow = $10*DT$
</History>
```

Finally, a history must be defined which computes the quantity which is to be time averaged. In the example below, a **pseudoPotential** *history* is used that defines the potential by integration of the electric field along a line.

```
<History potential>
        kind = pseudoPotential
        field = varEmField.ElecMultiField
        referencePoint = [$NX/2$ 0 $NZ/2$]
        measurePoint  = [$NX/2$ NY $NZ/2$]
</History>
```

3.15.3 Particle History

speciesAbsPtclData

Works with VSimBase, VSimEM, VSimPD, VSimPA, and VSimMD licenses.

Indicates Vorpal should calculate the specific information about particles absorbed by a given absorber. This History collects data on a per particle basis. It can track different kind of physical information of absorbed species via the choice of `ptclAttribute`. **speciesAbsPtclData** has multiple species or absorbers. If only one species is specified, multiple absorbers can be specified for this species via `ptclAbsorbers`. If more than one species are specified, same number of absorbers should be specified and each species corresponds to exactly one absorber associated with that species.

See also, *speciesAbsPtclData2*.

speciesAbsPtclData Parameters

species (*string vector*, *required*)
Indicates particle species type for a History kind that reports data for particle species. One or more species can be specified.

ptclAbsorbers (*string vector*)
Specifies name(s) of the particle absorber(s) on which the absorbed current due to incident charged particles are recorded. One or more absorbers can be specified.

ptclAttribute (*string*, *required*)
Corresponds to information about the absorbed particle data. One of the following options:

- **time:** Update time.
- **position:** Where a particle is absorbed.
- **velocity:** Velocity when a particle is absorbed.
- **energy:** Energy when a particle is absorbed.
- **weight:** Weight of a particle when it is absorbed.
- **speciesIndex:** Index of an absorbed particle.
- **absorberIndex:** Index of an absorbed where a particle is absorbed.
- **rank:** Rank of the grid where a particle is absorbed in parallel computation.
- **totalEnergy:** Total energy of all the species that is absorbed.
- **totalCurrent:** Total current of all the species that is absorbed.
- **massLoss:** Mass of the species that is absorbed at each time step.

component (*integer*, *default value = :samp:'0'*)
Only applies when **ptclAttribute** is position or velocity. Sets which component of the position or velocity vector of the absorbed particle to record.

speciesAbsPtclData History Example

```
<History beamTotCurr>
    kind = speciesAbsPtclData
    species = [beamElectrons testElectrons]
    ptclAbsorbers = [ beamLeftAbsorber testLeftAbsorber]
    ptclAttribute = totalCurrent
</History>
```

```
<History testEnergy>
    kind = speciesAbsPtclData
    species = [testElectrons]
    ptclAbsorbers = [testLeftAbsorber]
    ptclAttribute = energy
</History>
```

```
<History beamTotEnergy>
    kind = speciesAbsPtclData
    species = [beamElectrons]
    ptclAbsorbers = [beamLeftAbsorber]
    ptclAttribute = totalEnergy
</History>
```

```
<History  testPosz>
    kind = speciesAbsPtclData
    species = [ testElectrons ]
    ptclAbsorbers = [ testLeftAbsorber ]
    ptclAttribute = position
    component = 2
</History>
```

```
<History  testT>
    kind = speciesAbsPtclData
    species = [ testElectrons ]
    ptclAbsorbers = [ testLeftAbsorber ]
    ptclAttribute = time
</History>
```

```
<History  testVx>
    kind = speciesAbsPtclData
    species = [ testElectrons ]
    ptclAbsorbers = [ testLeftAbsorber ]
    ptclAttribute = velocity
    component = 0
</History>
```

```
<History  wmpiRank>
    kind = speciesAbsPtclData
    species = [ mpiElectrons ]
    ptclAbsorbers = [ mpiLeftAbsorber ]
    ptclAttribute = rank
</History>
```

speciesCurrAbs

Works with VSimBase, VSimEM, VSimPD, VSimPA, and VSimMD licenses.

Indicates Vorpal should calculate the absorbed current for a species by an absAndSav absorber. **speciesCurrAbs** requires both a *species* parameter and *ptclAbsorbers* parameter. This **History** tracks the current from charged particles as they impact the absorbers.

speciesCurrAbs Parameters

species (*string vector*, *required*)
: Indicates particle species type for a **History** kind that reports data for particle species. Exactly one species name should be specified.

ptclAbsorbers (*string vector*)
: Specifies name(s) the particle absorber(s), on which the absorbed current due to incident charged particles are recorded. At least one absorber should be specified. The absorber must be a kind of absorb-and-save. If multiple absorbers are specified, total absorbed current on all these absorbers are recorded.

speciesCurrAbs History Example

```
<History ptclHistCurrent>
    kind = speciesCurrAbs
    species = [ions]
    ptclAbsorbers = [rightAbsorber]
</History>
```

speciesCurrEmit

Works with VSimBase, VSimEM, VSimPD, VSimPA, and VSimMD licenses.

Indicates Vorpal should record the emitted current of a species from a particle source. **speciesCurrEmit** requires both a *species* parameter and *ptclSource* parameter. This **History** tracks the emitted current of a charged species from a particle source. It only tracks one particle source of one given species.

speciesCurrEmit Parameters

species (*string*, *required*)
: Indicates particle species type for a **History** kind that reports data for particle species. Exactly one species name should be specified.

ptclSource (*string*, *required*)
: Specifies the name of a ParticleSource. Only one source name should be specified. The ParticleSource must be a LoaderEmitter (xvLoaderEmitter) type particle source with `emit` set as true (the default value), or a `sourceType`.

sourceType (*integer*, *required*)
: Specifies the type of a ParticleSource, whether to measure the current from an emitter, a loader, or both. Set it 0 to measure emitter current only, 1 to measure loader current only, and 2 to measure currents from both emitter and loader combined.

speciesCurrEmit History Example

```
<History currEmit>
    kind = speciesCurrEmit
    species = [ electrons ]
    ptclSource = electrons.xvLoaderEmitterSource
    sourceType = 0
</History>
```

speciesDiag

Works with VSimBase, VSimEM, VSimPD, VSimPA, and VSimMD licenses.

Indicates VSim should calculate average quantities for a particular species in the entire simulation domain.

speciesDiag Parameters

diagVar (*string vector*, *required*)
Quantities to average. Options are:

aveX, aveY, aveZ (average x, y, and z position)

aveVx, aveVy, aveVz (average x, y, and z velocity)

aveEng (average energy)

numInBandN (number of electrons in conduction band N, where N = 0, 1, or 2)

speciesDiag Particle Species History Example

```
<History speciesDiag>
    kind = speciesDiag
    diagVar = [aveEng aveVx aveVy aveVz aveX aveY aveZ numInBand0 numInBand1␣
↪numInBand2]
    species = [electrons]
</History>
```

speciesEnergy

Works with VSimBase, VSimEM, VSimPD, VSimPA, and VSimMD licenses.

Indicates Vorpal should calculate total energy for a particular constant weight species in the whole simulation domain.

speciesEnergy Parameters

species (*string vector*, *required*)
Name of a species type to record its total record its total energy in units of Joules over the whole simulation domain.

speciesEnergy Particle Species History Example

```
<History energyChromiumAtoms>
    kind = speciesEnergy
    species = [chromiumAtoms]
</History>
```

speciesEngyAbs

Works with VSimBase, VSimEM, VSimPD, VSimPA, and VSimMD licenses.

Indicates Vorpal should calculate the energy absorbed by a given absorber. This history calculates the energy for all the particles absorbed during one time step. **speciesEngyAbs** requires both a `species` and `ptclAbsorbers`. This **`History`** tracks the energy from charged particles as they impact the absorber.

speciesEngyAbs Parameters

`species` **(string vector, required):** Name of a species type to record its total energy in the whole simulation domain.

speciesEngyAbs Particle Species History Example

```
<History ptclHistEnergy>
   kind = speciesEngyAbs
   species = [ions]
   ptclAbsorbers = [rightAbsorber]
</History>
```

speciesMomen

Works with VSimBase, VSimEM, VSimPD, VSimPA, and VSimMD licenses.

Indicates Vorpal should calculate total momentum for a particular species in the whole simulation domain. It always records all three components of the momentum. Thus, for some simulations in 1D or 2D, some components of the momentum may always be zero.

speciesMomen Parameters

`species` (*string vector*, *required*)
Name of a species type to record its total record its total momentum in the whole simulation domain.

speciesMomen Particle Species History Example

```
<History momenXenon>
    kind = speciesMomen
    species = [xenon]
</History>
```

speciesNumberOf

Works with VSimBase, VSimEM, VSimPD, VSimPA, and VSimMD licenses.

Indicates Vorpal should calculate the total number of macro-particles for a simulation species.

speciesNumberOf Parameters

species (*string vector*, *required*)
Name of a species type to record its total number of macro-particles in the whole simulation domain.

speciesNumberOf Particle Species History Example

```
# Specify history diagnostics
<History numElectrons>
    kind = speciesNumberOf
    species = [electrons]
</History>
```

speciesNumPhysical

Works with VSimBase, VSimEM, VSimPD, VSimPA, and VSimMD licenses.

Indicates Vorpal should calculate the total number of physical-particles for a simulation species.

speciesNumPhysical Parameters

species
Name of a species type to record its total number of physical particles in the whole simulation domain. (string vector, required)

speciesNumPhysical Particle Species History Example

```
# Specify history diagnostics
<History numPhysElectrons>
    kind = speciesNumPhysical
    species = [electrons]
</History>
```

speciesRmsDistToAxis

Works with any VSim license.

Collect the average RMS distance to a given grid axis for all the particles in a species.

speciesRmsDistToAxis Parameters

species (*string vector*, *required*)
The name of the species use in the calculation.

axis (*integer*, *required*)
The axis to which Vorpal should calculate the average RMS distance.

speciesRmsDistToAxis History Example

```
<History rmsDistToAxis>
  kind = speciesRmsDistToAxis
  species = [electrons]
  axis = 0
</History>
```

speciesRmsMomen

Works with any VSim license.

Calculate the average RMS momentum for all the particles in a species.

speciesRmsMomen Parameters

species (*string vector*, *required*)
The name of the species use in the calculation.

components (*integer vector*, *required*)
A vector of components of average RMS momentum that should be recorded.

speciesRmsMomen History Example

```
<History rmsMomentum>
  kind = speciesRmsMomen
  species = [electrons]
  components = [0 1 2]
</History>
```

speciesRmsPosition

Works with any VSim license.

Calculate the average RMS position for all the particles in a species.

speciesRmsPosition Parameters

species (*string vector*, *required*)
The name of the species use in the calculation.

components (*integer*, *required*)
A vector of components of average RMS position that should be recorded

speciesRmsPosition History Example

```
<History rmsPositions>
  kind = speciesRmsPosition
  species = [electrons]
  components = [0 1 2]
</History>
```

speciesTrackTag

Works with VSimBase, VSimEM, VSimPD, VSimPA, and VSimMD licenses.

Indicates Vorpal should track the trajectory (position) or the internal variables of a subset of species particles. The species tracked must be a tagged species where each particle has a unique integer tag assigned to it. If there is no particle associated with a tag or the particle with a particular tag is removed from the simulation then the position is reported as all zeros.

speciesTrackTag Parameters

species (*string vector*, *required*)
: Indicates particle species type for a **History** kind that reports data for particle species. This species should be a tagged kind, for example, `kind = relBorisTagged`.

tags (*integer vector*)
: List of tags of particles that will be tracked.

maximumTag (*integer*)

: **Any particle with a tag smaller than the maximumTag will be tracked.** The user must specify either *tags* or `maximumTag`.

getTagsFromSpecies (*bool*)
: If true then tags will be generated from initial distribution of particles.

getTagsFromAbsorber (*bool*)
: If true then tags will be generated when particles interact with a specific particle sink.

tagsFromFile (*string*)
: If this is set the tags will be taken from a text with the given name. The format of the file is just one column of numbers which are the tags to be tracked.

xComponents (*integer vector*)
: Indicates which position components to track.

iComponents (*integer vector*)
: Indicates which internal variables (velocities, weights, etc) to track.

speciesTrackTag Particle Species History Example

```
<History trajectory>
    kind = speciesTrackTag
    # List of tags to be tracked
    tags = [0 1 2 3]
    # Or give a maximum tag to be tracked. Any particle
    # with a tag less than the maximum tag will be tracked.
    # maximumTag = 4
```

(continues on next page)

(continued from previous page)

```
    species = [electrons]
</History>
```

speciesDataOnPlane

Works with VSimBase, VSimEM, VSimPD, VSimPA, and VSimMD licenses.

Indicates Vorpal should track the particle data (positions and internal variables) on a plane (3D), or a line (2D), perpendicular to one of the Cartesian axes and that can be moving along this axis.

speciesDataOnPlane Parameters

species (*string vector*, *required*)
Indicates particle species type for a **History** kind that reports data for particle species.

dir (*integer*, *default value = 0*)
Direction in which the plane (or line) is moving. `0`: x, `1`: y, `2`: z.

initialPlanePos (*float*, *required*)
Location, on the `dir` axis, of the plane (or line) at `t=0`.

planeVelocity (*float*, *required*)
Velocity of the plane (or line), in m/s, along the `dir` axis.

speciesDataOnPlane Particle Species History Example

```
<History ptclOnPlane>
    kind = speciesDataOnPlane
    initialPlanePos = PLANE_POS
    planeVelocity = -VX_BOOST
    species = [electrons]
</History>
```

3.15.4 History Operation Histories

binaryOperation

Works with VSimBase, VSimEM, VSimPD, VSimPA, and VSimMD licenses.

This history does a binary operation on two other histories. The operation is performed at every time step. Resulting values are recorded by this history. Available binary operations are add, subtract, multiply and divide.

binaryOperation Parameters

The **binaryOperation** takes the following parameters:

histories (*required string vector*)
A vector of the names of the two histories for which the operation should be performed.

coeffs (*required float vector*)
A vector containing a two float numbers, specifying coefficients for the binary operation.

operation (*required string*)
Operation to apply to the histories; one of `add`, `subtract`, `multiply` or `divide`. Operations are:

```
add:       (coeffs[0]*histories[0]) + (coeffs[1]*histories[1])
subtract:  (coeffs[0]*histories[0]) - (coeffs[1]*histories[1])
multiply:  (coeffs[0]*histories[0]) * (coeffs[1]*histories[1])
divide:    (coeffs[0]*histories[0]) / (coeffs[1]*histories[1])
```

In `divide` operation, if coeffs[1]*histories[1] is 0, the resulted value is taken as 0.

binaryOperation History Example

```
<History QE>
  kind = binaryOperation
  histories = [NtransSum NseedSum]
  coeffs = [1.0 1.0]
  operation = divide
</History>
```

unaryOperation

Works with VSimBase, VSimEM, VSimPD, VSimPA, and VSimMD licenses.

This history does a unary operation on one history. The operation is performed at every time step. Resulting values are recorded by this history. Available unary operation is sum.

unaryOperation Parameters

The **unaryOperation** takes the following parameters:

history (*required string*)
The names of the history for which the operation should be performed.

coeff (*optional float*)
A float number specifying the coefficient for the unary operation. It defaults to 1.

operation (*required string*)
Operation to apply to the history. Only `sum` is available at this stage. It is defined as

```
sum:       sum (coeff * history)
```

unaryOperation History Example

```
<History NtransSum>
  kind = unaryOperation
  history = Ntrans
  coeff = 1.0
  operation = sum
</History>
```

3.15.5 Feedback Histories

feedbackDesired

`feedbackDesired`: Works with VSimPD and VSimMD licenses.

FeedbackDesired history objects are used to adjust parameters (currents and fields for example) based on measured time dependent quantities. An example would be automatically adjusting the current in a simulation to generate a desired voltage. The feedback desired is calculated based on the computation of the measured voltage compared to the desired voltage or any measured quantity verses a desired quantity. This factor is then used to adjust the current (or other value that can be defined in an **`STFunc`** block) so that the measured voltage approaches the desired voltage.

The update equation for the feedback factor (calculated in **`feedback`**) is given by:

$$F[n+1] = F[N] + (2 * dt/\tau) * max(-1, min(1, \frac{D[n]-M[n]}{|D[n]|+|M[n]|})) * \frac{\alpha * F[N] + \beta * (|Dn|+|Mn|)}{\gamma * F[N] + \delta}$$

where:

- $F[n+1]$: feedback factor at time level n+1
- dt: the simulation time step
- τ: relaxation time constant (`timeConstant`)
- $D[n]$: desired value at time level n
- $M[n]$: measured value at time level n
- α: Exponential coefficient in feedback calculation (`alpha`)
- β: A constant determines linear feedback for small feedback (`beta`)
- γ: A constant determines linear feedback for large feedback (`gamma`)
- δ: A constant determines minimum denominator for small feedback (`delta`)

Using the default values for α =1.0, β =0.0, γ =0.0, δ =1.0 gives:

$$F[n+1] = F[N] * (1.0 + 2.0 * dt/\tau * max(-1, min(1, \frac{D[n]-M[n]}{|D[n]|+|M[n]|})))$$

An STFunc of kind feedbackSTFunc must be defined to make use of a feedbackDesired history. The sumRhoJ source contains STFunc blocks defining the current source and we would like this current to depend on a feedback parameter.

Note: Details of feedbackSTFunc and its parameters can be found in *feedbackSTFunc*.

Required feedbackDesired Parameters

`feedbackHist`

Points to another history, in this case a history called potential.

`timeConstant`

Defines how quickly the feedback responds to differences in measured and desired values. If timeConstant is too small the solution may oscillate about the correct answer quite a bit (or even diverge). If the value is too large it may take a long time for the feedback factor to reach the desired value. Experimenting with the *`timeConstant`* as in the example feedback code is a simple way to learn to understand this.

`numberOfStepsForErrorAverage` (*integer*, *optional*)

Number of time steps to use in moving average. This is used when the *`feedbackHist`* is oscillating.

desiredHistory (*STFunc block*)
Defines the desired value of the feedbackHist (i.e. the desired value of the potential in the example case).

alpha (*optional*, *default = 1.0*)
Exponential coefficient in feedback calculation. α in the above equation.

beta (*optional*, *default = 0.0*)
A constant determines linear feedback for small feedback. β in the above equation.

gamma (*optional*, *default = 0.0*)
A constant determines linear feedback for large feedback. γ in the above equation.

delta (*optional*, *default = 1.0*)
A constant determines minimum denominator for small feedback. δ in the above equation.

feedbackDesired History Example

```
<History feedbackDesiredHistory>
  kind=feedbackDesired
  feedbackHist=potential
  timeConstant=$75.0*DT$
  <STFunc desiredHistory>
    kind=expression
    expression = 1.0e5
  </STFunc>
</History>
```

Finally, a history must be defined which computes the *measured value* and which is compared to the *desired value* in the feedback equation. In the example below, a pseudo potential is used that defines the potential by integration of the electric field along a line.

```
<History potential>
    kind = pseudoPotential
    field = varEmField.ElecMultiField
    referencePoint = [$NX/2$ 0 $NZ/2$]
    measurePoint  = [$NX/2$ NY $NZ/2$]
</History>
```

Note: SumRhoJ sources are actually **STFunc** blocks, so the **feedbackSTFunc** can be used to adjust currents in simulations. Finally, restarts using feedback history requires history data to be dumped at every time step.

feedbackMeasured

Works with VSimPD and VSimMD licenses.

FeedbackMeasured history objects are used to pass the measured history information (currents and fields for example) based on measured time dependent quantities. An example where the the measured current in outer boundaries is used for setting the particle emission current by which the net loss of particle currents at the boundaries are adjusted in the simulation domain. Also user has the option to add any value to measured history data.

An STFunc of kind historySTFunc must be defined to make use of a feedbackMeasured history.

..note:: Details of historySTFunc and its parameters can be found in *historySTFunc*.

Required feedbackMeasured Parameters

`feedbackHist`
Points to another history.

`addToMeasuredValue` (*optional*, *default = 0.*)
Allows user to add a given value to the measured value from *`feedbackHist`* history.

feedbackMeasured History Example

```
<History feedbackMeasuredHistory>
  kind=feedbackMeasured
  feedbackHist=topWallElecCurrent
  addToMeasuredValue = wallCurrent
</History>
```

Finally, a history must be defined which computes the *measured value* and supply that information to feedbackMeasured. In the example below, history of topWallElecCurrent is used that measures the electron current collected at the top wall.

```
<History topWallElecCurrent>
    kind = speciesCurrAbs
    species = [ electrons ]
    ptclAbsorbers = [ topWall ]
</History>
```

3.15.6 Scalar Histories

scalarValue

`scalarValue` **(vector):** Works with VSimBase, VSimEM, VSimPD, VSimPA, and VSimMD licenses.

Records one or more scalar values in time, and saves them into History file.

scalarValue Parameters

`scalars` (*string vector*, *required*)
Indicates the names of scalars for a **History** to record their values in time.

scalarValue History Example

```
 <History scalarHist>
   kind = scalarValue
   scalars = [multiField.A1 multiField.A2]
</History>
```

3.15.7 Tensor Histories

fieldArray

Stores field values in a sub-array of the field.

Histories of `kind=fieldArray` are tensor histories that store values in a sub-array of a <Field>. For example, one might want to store the values of the electric field E in the $5 \times 6 \times 4$ sub-array from `lowerBounds=[2 2 2]` to `upperBounds=[7 8 6]`. In this case if one specifies `components = [0 1 2]` the resulting history will be a 1D array of $5 \times 6 \times 4 \times 3$ arrays, or a dataset of dimension $n \times 5 \times 6 \times 4 \times 3$.

(See *fieldAtIndices*, which is similar, but stores field values for one cell only.)

fieldArray Parameters

`field` (*string*, *required*)
The name of the <Field> for which to record values (the field name may need to be qualified: e.g., `myEmField.yeeE` instead of just `yeeE`).

`lowerBounds` (*vectors of integers*, *required*)
The lower bounds of the field values.

`upperBounds` (*vectors of integers*, *required*)
The upper bounds of the field values. As usual in Vorpal, the range is exclusive of the upper bound.

`components` (*vector of integers*, *required*)

The components of the field to be recorded; e.g., `components = [0 1 2]` to record all components of a 3-vector field, or `components = [0]` to store just the x-component.

fieldArray History Example

```
<History E1>
 kind = fieldArray
 field = emField.elecField
 components = [0 1 2]
 lowerBounds = [$NX/2$ $NY/2$ $NZ/2$]
 upperBounds = [$NX/2+1$ $NY/2+1$ $NZ/2+1$]
</History>
```

functionOfTime

Stores values calculated as a function of time and time-step, usually for the purpose of combining results from other histories

Histories of `kind=functionOfTime` are ostensibly that: they record a value that depends on the time t and time-step n. However, their main purpose is to look up results from other Histories and other Vorpal objects, and combine them in different ways.

If one places a UserFunc of `kind=historyFunc` inside histCalc (see *historyFunc*), then histCalc can use the result from another history in its calculation. It's important that one pay attention to the order of histories in the input file: usually one wants the other history (referenced by the historyFunc) to precede the history that uses it.

functionOfTime Parameters

`histCalc` (*code block*, *required*)
An Expression `<Expression histCalc>` that takes two arguments, `t` (a scalar float) and `n` (a scalar integer). Typically this Expression will contain local functions that look up values in other Vorpal objects (e.g., *fieldFunc*).

`backupHistCalc` (*code block*, *optional*)
An Expression `<Expression backupHistCalc>` with the same argument- and result-types as `<Expression histCalc>`, to be evaluated if evaluation of `histCalc` fails, and if `warningLevel < 3`.

`lastResortHistCalc` (*code block*, *optional*)
An Expression `<Expression lastResortHistCalc>` with the same argument- and result-types as `<Expression backupHistCalc>`, to be evaluated if evaluation of `backupHistCalc` fails, and if `backupWarningLevel < 3`. If evaluation of this function fails, the simulation will halt.

`resultVecDescriptions` (*vector of strings*, *optional*)
Associates a string description with each component of the history result, to be written in the dump file. If specified, the number of strings must equal the number of components in the history result; otherwise Vorpal will halt with an error.

`warningLevel` (*integer in {0, 1, 2}*, *default 0*)
Specifies what to do if evaluation of histCalc fails; if 0, simply evaluate backupHistCalc; if 1, issue a warning only after the first failure, and evaluate backupHistCalc; if 2, issue a warning and evaluate backupHistCalc; if 3, halt the simulation.

Note: In a parallel simulation, evaluation may fail on one rank but not on another, so it may be important to search the output on each rank for warnings and error messages.

`backupWarningLevel` (*same as warningLevel*)
The same as warningLevel, but applies to failure to evaluate backupHistCalc rather than histCalc. If backupWarningLevel is less than 3, and backupHistCalc fails, lastResortHistCalc will be evaluated instead (for that cell).

functionOfTime History Example

```
<History frequencyEstimate>
 kind = functionOfTime
 <Expression applySteps>
  expression = (n > 0)
 </Expression>
 <Expression histCalc>
  <UserFunc freqSqrEstWeight>
    kind = historyFunc
    history = frequencyMoments
    index = [0]
  </UserFunc>
  <UserFunc freqSqrEstSum>
    kind = historyFunc
    history = frequencyMoments
    index = [1]
  </UserFunc>
  <UserFunc freqSqrEstSumSqr>
```

(continues on next page)

(continued from previous page)

```
    kind = historyFunc
    history = frequencyMoments
    index = [2]
  </UserFunc>
  $ F2_AVGexpr = (freqSqrEstSum(0)/freqSqrEstWeight(0))
  $ MSDEVexpr = (freqSqrEstSumSqr(0)/freqSqrEstWeight(0) - F2_AVGexpr^2)
  expression = vector( \
     sqrt(F2_AVGexpr), \
     if(MSDEVexpr > 0., MSDEVexpr, 0.) )
  resultVecDescriptions = [ "frequency estimate (Hz)"  "mean square deviation of␣
↪frequency-squared (Hz^4)" ]
 </Expression>
</History>
```

cellFuncHist

Calculates a value in every cell, from a user-given function, which may reference fields, surfaces, etc., and combines all those values (e.g., by summing). This history is especially useful for surface and volume integrals, as well as peak fields.

Histories of `kind=cellFuncHist` are tensor histories that compute a result (a vector, or one-dimensional array of values) for every cell, and combine those results in some way to produce a result for the whole simulation.

For example, one might define the `<UserFunc histCalc>` to find a field value in a given cell, and multiply it by the volume of that cell. When the result is summed over all cells (`reduceMethod = sum`), it yields a volume integral.

Alternatively, one might use `reduceMethod = max`, in which case the maximum value (over all cells) would be returned.

cellFuncHist Parameters

These Histories take the following attributes:

lowerBounds (*vectors of integers*, *optional*, *default=entire simulation*)
The lowerBounds and upperBounds specify the range of cells (with an exclusive upper bound, as usual in Vorpal) for which this history will perform its calculation.

upperBounds (*vectors of integers*, *optional*, *default=entire simulation*)
The lowerBounds and upperBounds specify the range of cells (with an exclusive upper bound, as usual in Vorpal) for which this history will perform its calculation.

reduceMethod (*required string in {sum*, *min*, *max}*)
The method by which the results from individual cells are combined to create the final result. If sum, the results are simply summed. If min or max, then the history result is the result from the cell for which the *last component* (of the `histCalc` result) is minimal or maximal, respectively.

histCalc (*code block*, *required*)
A UserFunc `<UserFunc histCalc>` that takes one argument, named `cell`, which is a vector of NDIM integers specifying a grid-cell index. Typically this UserFunc will contain local functions that get recent information from other Vorpal objects (e.g., *fieldFunc* or *gridBoundaryFunc*). (See *UserFunc Block*)

lastResortHistCalc (*code block*, *optional*)
A UserFunc `<UserFunc lastResortHistCalc>` with the same argument- and result-types as `<UserFunc backupHistCalc>`, to be evaluated if evaluation of `backupHistCalc` fails, and if `backupWarningLevel < 3`. If evaluation of this function fails, the simulation will halt.

resultVecDescriptions (*vector of strings*, *optional*)
Associates a string description with each component of the history result, to be written in the dump file. If specified, the number of strings must equal the number of components in the history result; otherwise Vorpal will halt with an error.

warningLevel (*integer in {0, 1, 2}*, *default 0*)

Specifies what to do if evaluation of histCalc fails; if 0, simply evaluate `backupHistCalc`; if 1, issue a warning only after the first failure, and evaluate `backupHistCalc`; if 2, issue a warning and evaluate `backupHistCalc`; if 3, halt the simulation. In a parallel simulation, evaluation may fail on one rank but not on another, so it may be important to search the output on each rank for warnings and error messages.

backupWarningLevel (*same as warningLevel*)
The same as `warningLevel`, but applies to failure to evaluate `backupHistCalc` rather than `histCalc`. If `backupWarningLevel` is less than 3, and `backupHistCalc` fails, `lastResortHistCalc` will be evaluated instead (for that cell).

Use of backupHistCalc and lastResortHistCalc

Each cellFuncHist History has one to three <UserFunc>s that specify the function to be evaluated in each cell: `histCalc, backupHistCalc, lastResortHistCalc`. The function `histCalc` is evaluated for every cell; if, in any cell, evaluation fails, then `backupHistCalc` is evaluated for that cell instead; if evaluation fails again, `lastResortHistCalc` will be evaluated.

In such cases where field interpolation is performed, it is recommended that the `backupHistCalc` be used to obtain the same result, but (e.g, for a UserFunc of *fieldFunc*) with `interpolationOrder = 0` (assuming the `histCalc` used a higher order). Since that might also fail, it is further recommended that lastResortHistCalc perform a fail-safe calculation that will not harm the final result, if possible; for example, when performing an integral, it might simply return 0.

While a small number of evaluation failures will hardly affect the result in many cases, there are of course cases that cannot tolerate such failures. In those cases, one should use the options `warningLevel` and `backupWarningLevel` to issue a warning or even halt the simulation if failure occurs.

If evaluation of `lastResortHistCalc` fails, the simulation will halt with an error message.

The use of UserFuncs that obtain values from other simulation entities allows cellFuncHist to be a very powerful tool, computing locations of peak field values, surface integrals, volume integrals, etc.

cellFuncHist History Example

The best place to see examples of powerful and complicated cellFuncHists is in the history macro file, `hist.mac`, which contains macros implementing CellFuncHists for calculating surface and volume integrals, etc.

```
# sum the values of field F
<History sum>
  kind = cellFuncHist
  reduceMethod = sum
  lowerBounds = [0 0 0]
  upperBounds = [NX NY NZ]
  # calculate history every fourth time step
  <Expression applySteps>
    expression = (mod(n, 4) == 0)
  </Expression>
  <UserFunc histCalc>
    kind = expression
```

(continues on next page)

(continued from previous page)

```
    inputOrder = [cell]
    <Input cell>
      kind = uniformVector
      type = integer
      length = NDIM
    </Input>
    <UserFunc F>
      kind = fieldFunc
      result = fieldValue
      field = multiField.F
      interpolationOrder = 0
    </UserFunc>
    # since DX=1 and origin = 0, cell index and position are identical
      expression = F(cell)
  </UserFunc>
  resultVecDescriptions = ["sum of F"]
  # following is a rough memory size (in MB) that can always be
  # obtained: it's (very) small here for testing; normally it's best to
  # use the default value.
  maxMemChunkSize = 9.6e-5 # about 100 B
  # to use such a small value for testing, we need to set the following
  allowSmallMaxMemChunkSize = true
</History>
```

speciesAbsPtclData2

Stores data (individual or statistical) for particles absorbed by a boundary

The History of `kind=speciesAbsPtclData2` (intended as an eventual replacement for `kind=speciesAbsPtclData`, c.f., *speciesAbsPtclData*) is a tensor history that records data about particles that get absorbed (by a `kind=absAndSav` or similar ParticleSink, c.f., *absStairStep Parameters*).

Depending on the attribute `collectMethod` this can record data for each individual particle, or it can record sums or statistics for all particles absorbed within each time step.

speciesAbsPtclData2 Parameters

`ptclAbsorbers` (*vector of strings*, *required*)
A list of particle absorbers (of `kind=absAndSav` or similar, see *absStairStep Parameters*) for which this history will record data. If this is non-empty, `species` must contain a list of the absorbed species, one for each absorber.

`species` (*vector of strings*, *required*)
A list of Species, one for each ptclAbsorber, indicating the species that will be absorbed.

`ptclSources` (*vector of strings*, *optional*, *default = []*)
A list of ParticleSources (with `recordParticleData=true`) for which this history will record data. If this is non-empty, `ptclSourceSpecies` must contain a list of the sourced species, one for each source.

`ptclSourceSpecies` (*vector of strings*, *optional*, *default = []*)
A list of species, one for each ParticleSource in `ptclSources`.

`collectMethod` (*string*, *required*)
The method for collecting particle data. May be one of the following:

`recordForEachPtcl`:

Store the desired particle attributes for each absorbed particle in a separate record—–useful when one wants to know the data for each particle; in this case the absorption time will automatically be recorded for each particle (as the last component of the result).

In this case, the history dataset will be an array of size $N_p \times (1+m)$ where m is the number of `ptclAttributes` given, and N_p is the number of particles absorbed.

`sumForEachStep`:

Sum the desired particle attributes for all particles absorbed for each time step, yielding one record per time step

In this case, the history dataset will be an array of size $n \times m$ where m is the number of `ptclAttributes` given, and n is the number of time steps.

`statsForEachStep`:

At each step, record

- The total weight of absorbed particles, $\sum_p w_p$,
- The weighted sum of desired particle attributes, $\sum_p w_p a_p$, where a_p is the desired particle attribute, such as `xPosition` or `yVelocity`, and
- The weighted sum of squares, $\sum_p w_p a_p^2$.

(For non-weighted particles, w_p is one.)

If there are m quantities in the `ptclAttributes` string, then the history dataset will be an array of dimensions $n \times 3 \times m$, where in in the number of time steps. For each time step, the $3 \times m$ record has elements

- $(0, n)$, the total weight of particles absorbed;
- $(1, n)$ the weighted sum of particle attribute n; and
- $(2, n)$ the weighted sum of squares.

Dividing $(1, n)$ by $(0, n)$ yields the average of quantity n over all particles absorbed in a given time step.

`ptclAttributes` (*vector of strings*, *required*)
A list of the particle attributes that should be recorded by this history. Valid attributes are listed in *speciesBinning* under `ptclAttributes`.

speciesAbsPtclData2 History Example

```
<History testMoments>
 kind = speciesAbsPtclData2
 species = [ beamElectrons ]
 ptclAbsorbers = [ beamLeftAbsorber ]
 ptclAttributes = [position_0 position_1 position_2 gammaVelocity_0 gammaVelocity_1␣
↪gammaVelocity_2 relativisticGamma ]
 collectMethod = statsForEachStep
</History>
```

speciesBinning

A History that bins or sums particle data into an array according to a user-specified function; for example, the velocity distribution of a species can be recorded at every time step.

This history creates a multidimensional array of bins, with each bin contaning a vector; at every time step, a new array is created, and for each particle, a (vector) value is added to a particular bin; the bin is chosen based on the particle's properties.

Data is recorded for all particles in a simulation belonging to one of a list of specified species, or alternatively, for all particles absorbed by a list of specified particle absorbers.

To use the history, one must decide the dimensions of the array of bins, as well as the length of the vectors located at every array index. One then writes a Expression that, based on particle properties requested in `ptclAttributes`, determines an index (of the array), and a vector-value (to add to the bin with that index).

speciesBinning Parameters

`species` (*vector of strings*, *required*)
A list of Species for which data will be recorded.

If the `ptclAbsorbers` attribute is not given, then data for all the particles in these species will be recorded; otherwise, data will be recorded only for species absorbed by the specified absorbers (in this case, there should be one Species listed for each ptclAbsorber).

`ptclAbsorbers` (*vector of strings*, *optional*, *default = []*)
A list of ParticleSinks (of `kind=absAndSav` or similar, see *absStairStep Parameters*) for which this history will record data. If this attribute is given, only data for absorbed particles will be recorded. There must be, listed in the *`species`* attribute, one species for each absorber.

`ptclSources` (*vector of strings*, *optional*, *default = []*)
A list of ParticleSources (with `recordParticleData=true`) for which this history will record data. There must be, listed in the *`ptclSourceSpecies`* attribute, one species for each source.

`ptclSourceSpecies` (*vector of strings*, *optional*, *default = []*)
A list of species, one for each ParticleSource in *`ptclSources`*, indicating the emitted species (note that some ParticleSources can emit multiple species).

`binDims` (*vector of positive integers*, *required*)
The dimensions of the "array of bins" (the array of vectors) or equivalently, the dimensions of each history record not counting the last dimension (which is given by `arrayComponents`).

> For example, to bin by velocity direction, one might choose to have 40 divisions in longitude and 20 division in latitude, hence `binDims = [40, 20]`.

`binDimDescriptions` (*vector of strings*, *optional*)
A list of descriptions or labels for each array dimension. For example, when recording velocity angular distributions, the bin dimensions might represent the ϕ and θ coordinates, and so `binDimDescriptions` might be `["phi (radians, 40 divisions)" "theta (radians, 20 divisions)"]`, or whatever descriptions might be helpful for someone examining the output file.

`binComponents` (*positive integer*, *required*)
The length of the vector in each bin, or equivalently, the length of the last dimension of each history record.

`binComponentDescriptions` (*vector of strings*, *required*)
A list of descriptions or labels for each of the `binComponents`. For example, `binComponents = 2` and the 2 components are the total weight of particles and the number of macroparticles (in each bin), then one might use `binComponentDescriptions = ["total ptcl weight" "number of macroparticles"]`.

`indexAndResult` (*code block*, *required*)
An Expression `<Expression indexAndResult>` that takes as many scalar arguments as elements in `ptclAttributes`, and returns $n_d + n_c$ values, where n_d is the number of "bin" dimensions (the number of elements in `binDims`), and *n_c* is `binComponents`. See *expression*.

The first n_d values must be integers; moreover, they must be a valid index in arrayDims (in the expression, the funtions `int(x)`, `floor(x)`, and `ceil(x)` may be useful for creating integers from floats).

`warningLevel` (*integer*, *optional*, *default = 2*)
Specifies behavior if <Expression indexAndResult> produces an index that does not correspond to a bin (i.e., that is out of range).

- 0: ignore completely
- 1: warn the first time this happens (and ignore particle)
- 2: warn every time this happens (and ignore the particle)
- 3: halt the simulation when this happpens

`DatasetAttrib` (*vector of floats*)
`<DatasetAttrib>` contains a vector of floats to be added to attributes in the History's dataset stored on disc (code block, optional). The DatasetAttrib takes a single attribute, `value`. For example:

```
<DatasetAttrib binEdgesInJoules>
  value = [0. 4e-19 8e-19 1.2e-18 1.6e-18 2e-18]
</DatasetAttrib>
```

will store the above list of values under an attribute named `binEdgesInJoules`. (One can specify as many <DatasetAttrib> objects as desired.)

This may be useful in, e.g., the following situation. Suppose one is binning particles by energy, with 5 bins, from 0 to 2e-18 J. For post-simulation analysis, it might be useful if the history dataset also described the bins. By adding the above <DatasetAttrib>, that information gets stored along with the history dataset; someone analyzing the history dataset can then understand how the data is binned, without having to look in the input file.

`ptclAttributes` (*vector of strings*, *required*)
A list of the particle attributes that should be recorded by this history. The following strings are allowed.

`one`: The number one. This is useful, e.g., for History of kind *speciesAbsPtclData2*, where one might want to count the number of macroparticles—e.g., summing the number 1 for each macroparticle.

`ndim`: The simulation spatial dimensionality.

`numInternVars`: The number of internal variables per particle (e.g., velocity components, plus weight, plus tag, etc.)

`internVarsRole_0`, `internVarsRole_1`, etc.: The role of an internal variable, such as velocity or weight, expressed an an integer.

`time`: The time (e.g., at which the particle was absorbed), is seconds.

`weight`: The particle weight (always 1 except for variable-weight particles); usually `numPtclsInMacro` is the quantity desired for physical results, rather than `weight`, because the `weight` is relative to the macroparticle, not the physical particle. E.g., if each standard macro particle has 100 electrons, then a macro-particle with weight 1 is equivalent to 100 electrons, and weight 0.1 is equivalent to 10 electrons.

`tag`: The particle tag (or, if the species is untagged, returns -1.).

`numPtclsInMacro`: The number of physical particles in the macroparticle, or physical weight (i.e., the macro-particle's `weight` times the species's number of physical particles per macroparticle).

`mass`: The mass (in kg) of the macroparticle.

physPtclMass: The mass (in kg) of a physical particle.

charge: The charge (in Coulombs) of the macroparticle.

physPtclCharge: The charge (in Coulombs) of a physical particle.

cell_0, cell_1, etc.: The global index of the cell containing the particle.

position_0, position_1, position_2: The position of the particle, (e.g., in Cartesian geometry, `position_0` is x, `position_1` in y, etc., in meters). If the particle was absorbed or emitted, this is the location at or from which it was absorbed or emitted.

velocity_0, velocity_1, etc.: Components of velocity (m/s).

gammaVelocity_0, gammaVelocity_1, etc.: Components of gamma times velocity (m/s).

relativisticGamma: $1/\sqrt{1-v^2/c^2}$

velocityMagnitude: The magnitude of the particle velocity (m/s)

gammaVelocityMagnitude: The magnitude of gamma times the particle velocity (m/s)

physKineticEnergy: The (relativistic) kinetic energy of a physical particle in the macroparticle (in Joules)

kineticEnergy: The (relativistic) kinetic energy of the macroparticle (in Joules)

physNonRelativisticKineticEnergy: The non-relativistic kinetic energy ($mv^2/2$) of a physical particle in the macroparticle (in Joules) Note: this assumes that the species is non-relativistic (in the sense that its internal variables store the 3-velocity and not the 4-velocity; a relativistic species will calculate, meaninglessly, $0.5mv^2/(1-v^2/c^2)$).

nonRelativisticKineticEnergy: The non-relativistic kinetic energy ($mv^2/2$) of the macroparticle (in Joules). Note: this assumes that the species is non-relativistic (see `physNonRelativisticKineticEnergy`).

rank: The rank (in a parallel simulation) on which the particle was absorbed

speciesBaseNumPtclsInMacro: The default number of physical particles per macroparticle (e.g., in a macroparticle with weight 1.)

speciesWeightIndex: The component of the internal variables containing the weight (or -1 for species that don't have variable weight).

speciesTagIndex: The component of the internal variables containing the tag (or -1 for species that aren't tagged).

speciesIsRelativistic: 1 if the species is relativistic, 0 if not (currently this is not very trustworthy; the criteria for being relativistic are not consistent across all species; ideally, it should indicate whether a species stores the 4-velocity or the 3-velocity in its internal variables).

current: The current associated with the macroparticle charge (in A), i.e., the charge divided by the time-step

speciesIndex: The index of the species of the particle; i.e., if multiple species are collected.

absorberIndex: The index of the absorber that absorbed the particle, if particles are collected from multiple absorbers.

sourceIndex: The index of the source that emitted the particle, if particles are collected from multiple sources.

Particles that are collected from absorbers or emitters can have additional attributes:

surfaceNormalIntoAbsorber_0, surfaceNormalIntoAbsorber_1, etc.: The unit surface normal pointing into the absorber (or away from the direction of emission); this can be the zero vector in cases where the normal is not known (which can sometimes happen in pathological cases)

surfaceTangent1_0, surfaceTangent1_1, etc.: A vector tangent to the (absorption or emission) surface and in the plane of the particle's velocity (with a positive scalar product between the two).

surfaceTangent1_0, surfaceTangent1_1, etc.: A vector tangent to the (absorption or emission) surface, perpendicular to the normal and `surfaceTangent1`; The vectors (-`surfaceNormalIntoAbsorber`, `surfaceTangent1`, `surfaceTangent2`) should form a right-handed orthonormal basis (i.e., with the normal pointing away from the absorber, which is convenient for an emission-velocity basis).

incidentAngle: The angle (in radians) of incidence relative to the surface normal into absorber. An angle of 0 indicates a velocity pointing directly into the absorber.

cosIncidentAngle: The cosine of the angle of incidence.

speciesBinning History Example

```
<History gammaDist>
 kind = speciesBinning
 species = [electrons]
 ptclAttributes = [numPtclsInMacro relativisticGamma]
 $ NUMBINS = 200
 $ G_MIN = 1.
 $ G_MAX = 30.
 binDims = [NUMBINS]
 binDimDescriptions = ["relativisticGamma"]
 binComponents = 1
 binComponentDescriptions = ["number of physical particles in bin"]
<Expression indexAndResult>
 kind = expression
 $ g = relativisticGamma
 $ INDEX = min(NUMBINS-1, int(0.5 + (NUMBINS-1)*(g-G_MIN)/(G_MAX-G_MIN)))
 expression = vector(INDEX, numPtclsInMacro)
</Expression>
 # to explicate hdf5 dataset
 $ GAMMA_BIN_EDGES = [ G_MIN + n*(G_MAX-G_MIN)/NUMBINS for n in range(NUMBINS+1)]
<DatasetAttrib gammaBinEdges>
 value = GAMMA_BIN_EDGES
</DatasetAttrib>
</History>
```

3.16 External Circuit Model

3.16.1 External Circuit Model

General Theory

The external circuit module in Vorpal follows the method of V. Vahedi and G. DiPeso *[VD97]*. In Vorpal this method is extended to three-dimensional electromagnetic and electrostatic solvers, in additional to two-dimensional electrostatics. In principle, the total current, $J_t = \frac{\partial D}{\partial t} + J$, is conserved through the whole system. In the external circuit, the total current J_t is solved from the different equation based on its impedance (resistance R, inductance L, capacitance C) and source components of the circuit. In the simulated plasma region, the total current consists of displacement current $\frac{\partial D}{\partial t}$ and convective current carried by charged particles J. Matching these two at the boundary provides a time-dependent boundary condition for the simulated plasma region.

Considering a device that is connected to an external circuit via two electrodes, such as a electron gun or plasma discharging chamber, the total electric field between the electrodes is decomposed due to linear superposition as $\vec{E}_{tot}(t) = \vec{E}(t) + \phi(t)\vec{E}_0$, where $\vec{E}_{tot}(t)$ is the total electric field at time t that is used to advance charged particle dynamics. Electric field $\vec{E}(t)$ is obtained by solving the Maxwell‘s equations for electric ($\vec{E}$) and magnetic ($\vec{B}$) fields, assuming perfect electric conducting boundary conditions on the two electrodes that are both grounded. $\phi(t)$ is the time-dependent potential difference between the anode and the cathode that can be measured via pseudopotential. The electric field $\vec{E}_0$ is a constant field that is solved with 1 volt potential difference between the electrodes and no charged particles or currents between them. $\vec{E}_0$ is essentially a vacuum field with unit potential.

The potential $\phi(t)$ is related to the total charge $Q(t)$ on one electrode as $Q(t) = C_0\phi(t)$. As there is a fix capacitance C_0 for the device when vacuum is assumed, its potential is proportional to the total charge. The charge under 1 volt potential is Q_0, thus at any time t, the electrode potential is $\phi(t) = Q(t)/Q_0$. The total field is written as $\vec{E}_{tot}(t) = \vec{E}(t) + \frac{Q(t)}{Q_0}\vec{E}_0$, where $Q(t)$ is calculated at each time step via equations of external circuit, and Q_0 can be obtained after solving $\vec{E}_0$.

For a general series RCL circuit as shown in figure *Sketch of a plasma simulation with a general RLC external circuit.*, applying Kirchhoff’s law to the electrode yields the current equation, $\frac{dQ}{dt} = I_{conv} + I$, where I is the current in the external circuit that depends on its source and impedance. I_{conv} is the plasma convection current at the electrode, representing charges emitted or absorbed by the electrode at each time step.

The voltage equation is obtained by applying Kirchhoff’s law around the circuit, $L\frac{d^2Q_c}{dt^2} + R\frac{dQ_c}{dt} + \frac{Q_c}{C} = V(t) - \phi(t)$ where Q_c is the charge in the external circuit capacitor. It is related to the external circuit current via $I = dQ_c/dt$. Substituting this relation into current equation yields $\frac{dQ}{dt} = I_{conv} + \frac{dQ_c}{dt}$. In PIC simulations, the convective current I_{conv} and the potential difference between two electrodes are defined as $I_{conv} = I_{emitted} - I_{absorbed}$ and $\phi(t) = \int \vec{E}(t)dr$. Both can be measured in Vorpal via History blocks. Equations of current and voltage are solved for two unknowns $Q_c(t)$ and $Q(t)$. Both quantities determine the status of the external circuit and the boundary condition on the electrode.

The voltage equation is solved for capacitor charge Q_c at time step n as $Q_c^n = \frac{V(n)-\phi^n-K^n}{\alpha_0}$, where

$K^n = \alpha_1 Q_c^{n-1} + \alpha_2 Q_c^{n-2} + \alpha_3 Q_c^{n-3} + \alpha_4 Q_c^{n-4}$

$\alpha_0 = \frac{9}{4}\frac{L}{\Delta t^2} + \frac{3}{2}\frac{R}{\Delta t} + \frac{1}{C}$

$\alpha_1 = -6\frac{L}{\Delta t^2} - 2\frac{R}{\Delta t}$

$\alpha_2 = \frac{11}{2}\frac{L}{\Delta t^2} + \frac{1}{2}\frac{R}{\Delta t}$

$\alpha_3 = -2\frac{L}{\Delta t^2}$

$\alpha_4 = \frac{1}{4}\frac{L}{\Delta t^2}$

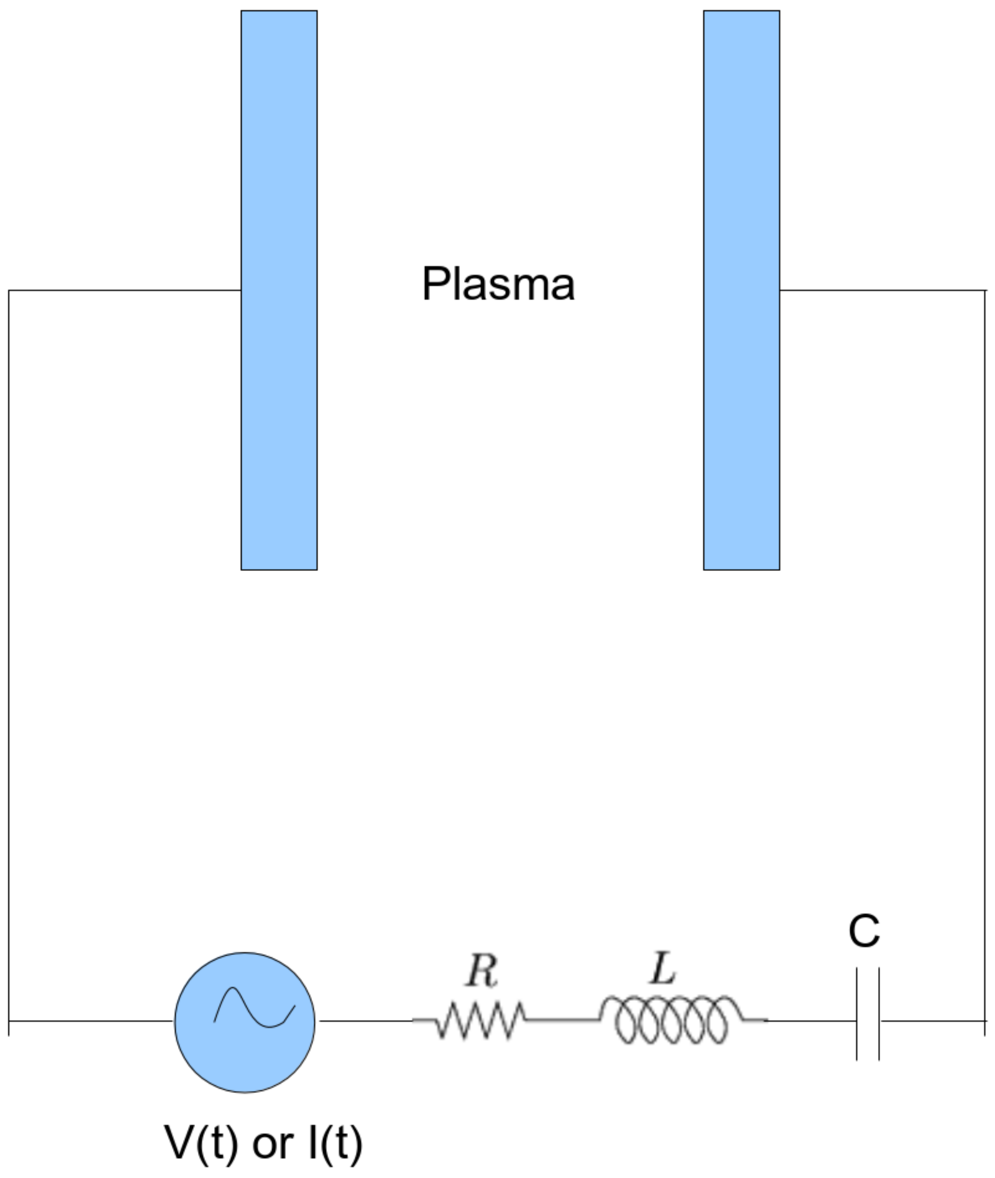


Fig. 3.3: Sketch of a plasma simulation with a general RLC external circuit.

The self-consistent solution for charges on the electrode is $Q^n = Q^{n-2} + I^{n-1}_{conv} \cdot 2\Delta t + Q^n_c - Q^{n-2}_c$. Both the potential ϕ^n and convective current I^n_{conv} can be measured in Vorpal. Equations of voltage and currents are solved with UserFuncUpdater in Vorpal to obtain the electrode charge Q^n at each time step. Q^n is then coupled with field equation to find the total electric field self-consistently in the system as $\vec{E}_{tot}(t) = \vec{E}(t) + \frac{Q^n}{Q_0}\vec{E}_0$. Beside the Courant stability condition that is required by the electromagnetic solver, the time step Δt should also follow certain constrains due to external circuits as discussed in reference *[VAVB93]*.

Simulation Procedures

Based on the above algorithm, a simulation with external circuit needs two stage execution. The first stage calculates $\vec{E}_0$ and Q_0 that are used in the second stage, where the actual simulation is carried out.

Stage One of the Simulation

The first stage runs a Poisson solver with unit potential in vacuum for only one time step to obtain the electric field $\vec{E}_0$ and the charge on electrode Q_0. As the electrodes are represented by GridBoundary, the Poisson solver looks as following

```
##########
#
# Electrostatic Poisson solver with GridBoundary
#
##########
<EmField myESField>
  kind = yeeStaticEmField

  # set the grid boundary
  gridBoundary = plates

  # Set potential on the grid boundary
  <STFunc boundaryFunc>
      kind = expression
      expression = 1.0*phiFunc(x,y,z)
  </STFunc>

   <Solver mysolver>
      kind = bicgstab
      precond = multigrid
      smoother = GaussSeidel
      nLevels = 7
      tolerance = 1.0e-10
      output = none
      dumpPotential = true
    </Solver>

</EmField>
```

The total charge on one electrode, such as the cathode used in this example, is calculated with cellFunHist with the following code.

```
# This is the total charge on the left plate (cathode)
# Its value is important and used for subsequent simulations
# Test its value for convergence
<History totalQ>
  kind = cellFuncHist
```

(continues on next page)

(continued from previous page)

```
  funcLookupScope = myFields
  reduceMethod = sum
  lowerBounds = [0 0 0]
  upperBounds = [$NX/2$ NY NZ]

  <Expression applySteps>
    expression = (mod(n, 1) == 0)
  </Expression>

  <UserFunc histCalc>
    kind = expression

    <CellFunc unitNormal>
      kind = gridBoundaryFunc
      gridBoundary = plates
      result = surfaceOutwardUnitNormal3D
    </CellFunc>

    <CellFunc surfacePos>
      kind = gridBoundaryFunc
      gridBoundary = plates
      result = surfaceCenter
    </CellFunc>

    <CellFunc dArea>
      kind = gridBoundaryFunc
      gridBoundary = plates
      result = surfaceArea
    </CellFunc>

    <SpaceFunc E>
      kind = fieldFunc
      result = fieldValue
      field = myESField.YeeStaticElecFld
      interpolationOrder = 1
      gridBoundary = plates
      polation = fromInOrOutside
      # the following option is not usually recommended
      #boundaryCondition = electricAtPEC
    </SpaceFunc>

    expression = -1.0 * sum(unitNormal(cell) * E(surfacePos(cell))) *␣
↪EPSILON0 * dArea(cell)
  </UserFunc>

  resultVecDescriptions = ["sum of total charge"]
  # following is a rough memory size (in MB) that can always be
  # obtained: it's (very) small here for testing; normally it's best to
  # use the default value.
  maxMemChunkSize = 9.6e-5 # about 100 B
  # to use such a small value for testing, we need to set the following
  allowSmallMaxMemChunkSize = true
  checkForUnaccessedAttribs = 2
</History>
```

After the run of this first stage, totalQ (Q_0) is read from the hdf5 file for History. It values, together with the electrostatic field (YeeStaticElecFldTrilinos), are used for stage two.

Stage Two of the Simulation

Stage two is to run the actual simulation that is coupled with external circuit. The convective current at the cathode is measured with the following two History blocks for emitted and absorbed currents.

```
<History cathodeEmitCurrent>
  kind = speciesCurrEmit
  species = [electrons]
  ptclEmitter = electrons.psource
</History>

<History cathodeAbsCurrent>
  kind = speciesCurrAbs
  species = [electrons]
  ptclAbsorbers = [cathodeAbs]
</History>
```

The electrostatic field at unit potential calculated from stage one is imported to an edge field name unitE0. It is imported from hdf5 file and only needs to update once at the beginning of the simulation.

```
<FieldUpdater initunitE0Updater>
  kind = importFromFileUpdater
  fileName = "./externalCircuit0_YeeStaticElecFldTrilinos_1.h5"
  dataset = "YeeStaticElecFldTrilinos"
  writeFields = [unitE0]
  writeComponents = [0 1 2]
  lowerBounds = [0   0   0]
  upperBounds = [NX1 NY1 NZ1]
</FieldUpdater>
```

Charges on the capacitor of external circuit and one of the electrodes are updated via two updaters that solve external circuit voltage and current equations. Q0 used in the chargeQUpdater is obtained form stage one simulation.

```
# This updated the Q at the capacitor C at time step n
# according to the external circuit voltage equation (Kirchoff's voltage law)
# Here the cuircuit is assumed to a general series RLC
<FieldUpdater capacitorQUpdater>
  kind = userFuncUpdater
  lowerBounds = [0 0 0]
  upperBounds = [NX1 NY1 NZ1]

  readFields = [phiField capacitorQnm1 capacitorQnm2 capacitorQnm3␣
↪capacitorQnm4]
  readComponents = [0 0 0 0 0]
  readFieldVarNames = [phi qnm1 qnm2 qnm3 qnm4]

  writeFields = [capacitorQn]
  writeComponents = [0]
  maxNumEvals = 64

  <UserFunc updateFunction>
    kind = expression
    $alpha0 = 2.25*L/DT/DT + 1.5*R/DT + 1/C
    $alpha1 = -6.0*L/DT/DT - 2.0*R/DT
    $alpha2 = 5.5*L/DT/DT + 0.5*R/DT
    $alpha3 = -2.0*L/DT/DT
    $alpha4 = 0.25*L/DT/DT
```

(continues on next page)

(continued from previous page)

```
    expression = (Vsource((n)*dt) - phi - alpha1*qnm1 - alpha2*qnm2 -␣
↪alpha3*qnm3 - alpha4*qnm4) / alpha0
  </UserFunc>
</FieldUpdater>


<FieldUpdater chargeQUpdater>
  kind = userFuncUpdater
  lowerBounds = [0 0 0]
  upperBounds = [NX1 NY1 NZ1]

  readFields = [totalQnm2 capacitorQn capacitorQnm2]
  readComponents = [0 0 0]
  readFieldVarNames = [qnm2 qcn qcnm2]

  writeFields = [totalQn]
  writeComponents = [0]
  maxNumEvals = 64

  <UserFunc updateFunction>
    kind = expression
    expression = qnm2 + (qcn - qcnm2) - 2.0*dt*(emitI - absI)
  </UserFunc>
</FieldUpdater>
```

Assuming both electrodes are perfect electric conductors, edgeE and faceB are solved with regular Faraday, Ampere and Dey-Mittra updaters. Combining edgeE with scaled unitE, we obtain the totalE field that is use to advance charged particles dynamics via Lorenz force together with faceB.

```
<FieldUpdater totalEUpdater>
  kind = userFuncUpdater
  lowerBounds = [0 0 0]
  upperBounds = [NX1 NY1 NZ1]

  readFields = [edgeE edgeE edgeE unitE0 unitE0 unitE0 totalQn]
  readComponents = [0 1 2 0 1 2 0]
  readFieldVarNames = [EX EY EZ E0X E0Y E0Z QN]

  writeFields = [totalE totalE totalE]
  writeComponents = [0 1 2]
  maxNumEvals = 64

  <UserFunc updateFunction>
    kind = expression
    expression = tensorProd(EX, EY, EZ) + tensorProd(E0X, E0Y, E0Z) * QN / Q0
  </UserFunc>
</FieldUpdater>
```

Notes for External Circuit

UserFuncs are heavily used in the external circuit module. Users are suggested to read relevant sections about UserFuncs. In the capacitorQUpdater, phi is measured via pseudopotential and read into a field via historySTFunc. For their usage, please refer to the sections about feedback and historySTFunc in *VSim Reference*.

3.17 Functions

3.17.1 Function Blocks

Function blocks built into Vorpal that are used to model various effects. The four types of **Function** blocks are:

NAfunc: No-argument function block. See *NAFunc Block*.

OAfunc: One-argument function block. See *OAFunc Block*.

STfunc: Space-Time function block. See *STFunc Block*.

SVTFunc: Space-Time-Velocity function block. See *SVTFunc Block*.

(Also see *Introduction to UserFuncs and Expressions*.)

3.17.2 NAFunc

NAFunc Block

Typically used for sequences, such as generating a set of velocities to be used in particle loading. Some NAFunc functions, for example, *randGauss*, take a nested NAFunc that uses the name `numberSequence`.

NAFunc Kinds

- *constNAFunc*
- *stretcher*
- *randGauss*
- *randExp*
- *randOAFunc (NAFunc)*

Example NAFunc Block

```
<NAFunc velocitySequence_0>
  kind = constNAfunc
  amplitude = 100.0
</NAFunc>
```

```
 <ParticleSource rampSrc>
   kind = bitRevDensSrc
   density = DENSITY
   lowerBounds = [X_LEFT_WALL 0. 0.]
   upperBounds = [X_RIGHT_WALL LY LZ]

 # Particle distribution uniform over initial phase space
  <NAFunc velocitySequence_0>
    kind = randGauss
    mean = 0.
```

(continues on next page)

(continued from previous page)

```
    sigma = VEL_500K
  </NAFunc>
  <NAFunc velocitySequence_1>
    kind = randGauss
    mean = 0.
    sigma = VEL_500K
  </NAFunc>
  <NAFunc velocitySequence_2>
    kind = randGauss
    mean = 0.
    sigma = VEL_500K
  </NAFunc>
  <NAFunc velocitySequence_3>
    kind = randExp
    mean = 1.
  </NAFunc>
</ParticleSource>
```

constNAFunc

Works with VSimBase, VSimEM, VSimPD, VSimPA, and VSimMD licenses. Constant no-argument function.

constNAFunc Parameters

amplitude (*float vector*)
This function accepts a value, which then is the amplitude value assigned to every value in the sequence.

Example constNAFunc Block

```
<NAFunc velocitySequence_0>
  kind = constNAFunc
  amplitude = 0.
</NAFunc>
```

bitRevNAFunc

Creates a bit-reversed sequence.

bitRevNAFunc Parameters

prime (*integer*)
The prime for the bit reverse.

Example bitRevNAFunc Block

```
<NAFunc velocitySequence_0>
  kind = bitRevNAFunc
  prime = 13
</NAFunc>
```

mTwister

Creates a bit-reversed sequence.

mTwister Parameters

prime (*integer*)
The prime for the bit reverse.

Example mTwister Block

```
<NAFunc numberSequence>
  seed = VX_SEED
  kind=mTwister
</NAFunc>
```

randExp

Works with VSimBase, VSimEM, VSimPD, VSimPA, and VSimMD licenses.

Creates a random sequence distributed according to the exponential probability density function:

$$f(x) = \begin{cases} \frac{1}{\mu}\exp\left(-\frac{x}{\mu}\right) & , x \geq 0 \\ 0 & , x < 0 \end{cases}$$

where $\mu > 0$ is the mean.

randExp Parameters

mean (*float*)
The mean.

seed (*integer*)
Random number seed.

numberSequence (*code block, optional*)
An NAFunc code block, `<NAFunc numberSequence>` generates a random number; if not specified, Vorpal's default random number generator (uniform in [0,1)) will be used to generate the initial random number.

Example randExp Block

```
<NAFunc velocitySequence_0>
  kind = randExp
  mean = 1.0
</NAFunc>
```

randGauss

Works with VSimBase, VSimEM, VSimPD, VSimPA, and VSimMD licenses.

Creates a random sequence distributed according to the Gaussian probability density function:

$$f(x) = \frac{1}{\sigma\sqrt{2\pi}} \exp\left[-\frac{(x-\mu)^2}{2\sigma^2}\right]$$

where $\mu \in \mathbb{R}$ is the mean and $\sigma > 0$ is the standard deviation.

randGauss Parameters

sigma (*float*)
: The standard deviation.

mean (*float*)
: The mean.

seed (*integer*)
: Random number seed.

numberSequence (*code block*, *optional*)
: The `<NAFunc numberSequence>` is an NAFunc code block that generates a random number; if not specified, Vorpal's default random number generator (uniform in [0,1)) will be used to generate the initial random number.

Example randGauss Block

```
<NAFunc velocitySequence_0>
  kind = randGauss
  mean = 0.0
  sigma = 1.0e6
</NAFunc>
```

randGaussLimit

Creates a random sequence distributed according to the Gaussian distribution with lower and upper bounds. See *randGauss* for standard Gaussian distribution.

randGaussLimit Parameters

sigma (*float*)
: The standard deviation.

mean (*float*)
The mean.

lowerLimit (*float*)
The lower limit for the distribution.

upperLimit (*float*)
The upper limit for the distribution.

seed (*integer*)
Random number seed.

numberSequence (*code block, optional*)
The `<NAFunc numberSequence>` is an NAFunc code block that generates a random number; if not specified, Vorpal's default random number generator (uniform in [0,1)) will be used to generate the initial random number.

Example randGaussLimit Block

```
<NAFunc velocitySequence_0>
  kind = randGaussLimit
  mean = 0.0
  sigma = 1.0e6
  lowerLimit = -2.0e6
  upperLimit = 2.0e6
</NAFunc>
```

randGamma

creates a random sequence distributed according to the Gamma probability density function:

$$f(x) = \begin{cases} \frac{1}{\Gamma(\alpha)}\beta^\alpha x^{\alpha-1}\exp(-\beta x) & , x > 0 \\ 0 & , x \le 0 \end{cases}$$

where $\alpha, \beta > 0$, α/β is the mean, $\sqrt{\alpha}/\beta$ is the standard deviation, and

$$\Gamma(x) = \int_0^\infty dt\, t^{x-1}\exp(-t)$$

is the gamma function.

randGamma Parameters

sigma (*float*)
The standard deviation.

mean (*float*)
The mean.

seed (*integer*)
Random number seed.

numberSequence (*code block, optional*)
`<NAFunc numberSequence>` is an NAFunc code block that generates a random number; if not specified, Vorpal's default random number generator (uniform in [0,1)) will be used to generate the initial random number.

Example randGamma Block

```
<NAFunc velocitySequence_0>
  kind = randGamma
  mean = 0.0
  sigma = 1.0e6
</NAFunc>
```

randKappa

Create a random sequence distributed according to the general kappa velocity distribution, assuming three velocity components.

randKappa Parameters

sigmas (*float*)
The widths of the distribution.

seed (*integer*)
Random number seed.

lowerLimits (*vector float*)
The lower limits for the velocities.

upperLimits (*vector float*)
The lower limits for the velocities.

numberSequence (*code block*, *optional*)
`<NAFunc numberSequence>` is an NAFunc code block that generates a random number; if not specified, Vorpal's default random number generator (uniform in [0,1)) will be used to generate the initial random number.

Example randKappa Block

```
<NAFunc velocitySequence_2>
  kind = randKappa
  sigmas = [$0.8*SIGMA_VEL$ SIGMA_VEL $1.2*SIGMA_VEL$]
</NAFunc>
```

randOAFunc (NAFunc)

Works with VSimBase, VSimEM, VSimPD, VSimPA, and VSimMD licenses.

A function that returns a random number based on a user-given random distribution.

randOAFunc Parameters:

seed (*non-negative integer*, *optional*)
A seed for Vorpal's default random number generator (used only if `<NAFunc numberSequence>` is not given). If seed is not given, a seed will be chosen by Vorpal's random number generator (N.B. this is a pseudo-random seed, not like using the current time for the seed.)

probDist (*code block*, *required*)
: `<OAFunc probDist>` is an OAFunc (function of one argument, returning a scalar) specifying a probability distribution, including the domain (which describes the range of possible values resulting from this random number generator). An OAFunc of `kind=expression` is the usual choice here.

maxRelErr (*float*, *optional*, *default = 1e-2*)
: The probability distribution will be approximated with a relative error below this level, subject to `maxNumPoints`.

maxNumPoints (*positive integer*, *optional*, *default = 40961*)
: The maximum number of points at which to sample the probability distribution. The number of sample points will be increased, trying to attain `maxRelErr`, up to `maxNumPoints`.

doComparison (*bool*, *optional*, *default = false*)
: For diagnostic purposes only, compare the approximation of the <OAFunc probDist>, printing out (to stdout) how close the approximation is.

printSampleValues (*non-negative integer*, *optional*, *default = 0*)
: For diagnostic purposes only, the number of sample random numbers to be printed (to stdout) so the user can see if they are distributed as desired.

numberSequence (*code block*, *optional*)
: `<NAFunc numberSequence>` is an NAFunc code block that generates a random number; if not specified, Vorpal's default random number generator (uniform in [0,1)) will be used to generate the initial random number.

  Random numbers will be generated with the desired distribution only if `numberSequence` generates a number uniformly in [0,1).

  Using this code block makes the `seed` option irrelevant.

stretcher

Works with VSimBase, VSimEM, VSimPD, VSimPA, and VSimMD licenses.

Function that stretches the distribution returned by another.

stretcher Parameters

minvalue (*float*)
: Minimum value of the random number.

maxvalue (*float*)
: Maximum value of the random number.

stretchFunc (*no-arg functor*)
: Function that is to be stretched.

seed (*optional*)
: Random seed of the stretched function *stretchFunc*, if it has one.

Example stretcher Block

```
<NAFunc velocitySequence_0>
  kind = stretcher
  minvalue = -1.0e6
  maxvalue =  1.0e6
</NAFunc>
```

sysRandom

Generates random sequence between 0 and 1 via system random number generator (drand48() from stdlib).

sysRandom Parameters

seed (*integer*)
The seed for the random generator.

Example sysRandom Block

```
<NAFunc velocitySequence_4>
  kind = sysRandom
  seed = 2341
</NAFunc>
```

3.17.3 OAFunc

OAFunc Block

Defines a function that depends on one variable only, which can be of two kinds: **interpolatedFromFile** or **expression**. For example, **OAFunc** are used in **MonteCarlo Interactions** to define interactions cross sections.

OAFunc Kinds

- *interpolatedFromFile*
- *expression (OAFunc)*
- *LXcatFile OAFunc*

Example OAFunc Block

```
<OAFunc mysp>
  kind = interpolatedFromFile
  filename = h2StoppingPower.dat
  # set bounds of the function from min and max x values in the file.
  # option specific to kind = interpolatedFromFile
  # setMinMaxFromFile = 1
  # f has to be >0.
  fmin = 0.
</OAFunc>
```

interpolatedFromFile

Works with VSimBase, VSimEM, VSimPD, VSimPA, and VSimMD licenses.

User provides a file with two columns where the first column is the argument of the function and the second column is the corresponding value of the function. The function is evaluated by linearly interpolating between the points.

interpolatedFromFile Parameters

filename (*string*)
Name of the file which contains the two-column data. It is assumed that the data in the first column is either in increasing or decreasing order.

setMinMaxFromFile (*boolean / integer*, *optional*, *default = false / 0*)
Set `xmin` and `xmax` parameters to the minimum and maximum values of the first column. If this parameter is `false`, and the function argument x is smaller (larger) than the minimum (maximum) value of the first column, linear interpolation from the two first (last) points is used.

Example interpolatedFromFile Block

```
<OAFunc oafunc1>
  # read data from a file
  kind = interpolatedFromFile
  filename = ../ionize/electronCSection.dat
</OAFunc>
```

expression (OAFunc)

Works with VSimBase, VSimEM, VSimPD, VSimPA, and VSimMD licenses.

A function of one argument, given via a UserFunc

expression Parameters

fmin (*float*, *optional*, *default = minimum float*)
The lower bound of the function's domain.

fmax (*float*, *optional*, *default = maximum float*)
The upper bound of the function's domain.

variable (*string*, *optional*, *default = x*)
Name of the variable in the expression.

expression (*string*)
The function expression (can be given directly for simple functions), required if a UserFunc code block is not given; in this case, a UserFunc of `kind=expression` is automatically created with this expression.

UserFunc (*code block*)
A UserFunc can be specified directly; it must take one argument, and return a scalar. This code block is required if a string `expression` is not given.

Example expression Block

```
<OAFunc oafunc1>
  kind = expression
  expression = cos(x)
</OAFunc>
```

3.17.4 STFunc

STFunc Block

Many blocks in Vorpal can use space-time functions. These are normally contained within other classes that require time signals or spatial profiles, such as *Initial and Boundary Conditions*.

When using an STFunc block, place the function in its own STFunc block, specifying the function using the `kind` parameter as in the *Example STFunc Block Usage*.

Users can make use of `kind = expression` to write their own functions.

Descriptions of STFunc parameters are described on each STFunc kind page.

STFunc Kinds

- *cacheFunc*
- *chirpWavePulse*
- *constantFunc*
- *cosineFlattop*
- *cosineRamp*
- *expression (STFunc)*
- *feedbackSTFunc*
- *gaussian*
- *gaussianPulse*
- *halfSinePulse*
- *ignoreArgFunc*
- *inverseFunc*
- *leakychannel*
- *multFunc*
- *planeWavePulse*
- *radialCosChannel*
- *sinePlaneWave*
- *sumFunc*
- *stPyFunc*
- *tagGen*

Example STFunc Block Usage

```
<FieldUpdater  currentSource_0>
  kind = setToSTFuncUpdater
  lowerBounds = [0 0 0]
  upperBounds = [10 20 30]
  component = 1 # J_x, rho is component 0
  writeFields = [SumRhoJ]

  <STFunc sourceFunc>
    kind = expression
    expression =  3.*exp(0.1*(t-10.)^2)*sin(3.14*t - 7.2*x)
  </STFunc>

</FieldUpdater>
```

cacheFunc

Works with VSimBase, VSimEM, VSimPD, VSimPA, and VSimMD licenses.

Defines a function that remembers its last arguments and value, and avoids recalculation if possible.

cacheFunc Parameters

STFunc (*block*, *required*)
The function to be cached.

timeIsIrrelevant (*bool*)
Whether the change is time is relevant to the function.

irrelevantDirs (*integer vector*)
Any direction where the change is irrelevant to the function.

Example cacheFunc Block

```
<STFunc memDrive>
  kind = cacheFunc
  irrelevantDirs = [1 2]
  <STFunc drive>
    kind = expression
    expression =  amp*H(T_DRIVE-t)*cos(K*x)*sin(OMEGA*t)
  </STFunc>
</STFunc>
```

chirpWavePulse

Works with VSimBase, VSimEM, VSimPD, VSimPA, and VSimMD licenses.

Produces a plane wave modulated by a pulse envelope, where the envelope is a half-sine function along the direction of motion, and it is Gaussian transverse.

chirpWavePulse Parameters

vg (*float*)
: Group velocity by which the origin of the envelope is moved.

amplitude (*float*)
: Amplitude of the pulse.

phase (*float*)
: Additional phase of the sinusoid.

keepon (*option*)
: If true, pulse is on after rises to max value.

origin (*float vector*)
: Origin of the envelope.

widths (*float vector*)
: Widths of the pulse.

- `widths[0]` (float)

  Width along the direction of motion. This is half the width of the non-zero extent of the half-sine, or the FWHM of the square of the half-sine.

- `widths[1]` (float)

  Width along the direction of perpendicular to **k** and the third (index 2) axis, unless **k** is along the third (index 2) axis, in which case this is the width along the first (index 0) axis. The width for transverse directions is the full width at the $\frac{1}{\sqrt{e}}$ points.

- `widths[2]` (float)

  Width along the direction of perpendicular to **k** and the direction for widths[1].

chirp (*float*)
: The chirp of the pulse is equal to the change in frequency during the full width at half power max; a positive chirp indicates rising frequency along the pulse.

skewness (*float*)
: Skewness in the pulse; positive skewness.

waistDisplacement (*float*)
: Displacement of the waist along the direction of propagation.

Example chirpWavePulse Block

```
<STFunc component0>
  kind = chirpWavePulse
  chirp = -0.9
  amplitude = 5.0e+12
  phase = 1.57
  k = [125.6 0. 0.]
  vg = LIGHTSPEED
  widths = [1.e-1  2.e-1 2.e-1]
  origin = [-1.e-1 0.e-5 0.e-5]
</STFunc>
```

constantFunc

Works with VSimBase, VSimEM, VSimPD, VSimPA, and VSimMD licenses.

Defines a constant space-time function.

constantFunc Parameters

amplitude **(float vector):** Value this function has.

Example constantFunc Block

```
<STFunc myConstFunc>
 kind = constantFunc
 amplitude = 1.0
</STFunc>
```

cosineFlattop

Works with VSimBase, VSimEM, VSimPD, VSimPA, and VSimMD licenses.

Function for a flat top function with cosine ramps up and down.

cosineFlattop Parameters

direction (*float vector*)
Direction of the ramp (unit vector $\hat{\mathbf{u}}$ of gradient).

startPosition (*float*)
Value of $\hat{\mathbf{u}} \cdot \mathbf{x}$ at which the ramp starts.

startFlattop (*float*)
Value of $\hat{\mathbf{u}} \cdot \mathbf{x}$ at which the flat top starts.

endFlattop (*float*)
Value of $\hat{\mathbf{u}} \cdot \mathbf{x}$ at which the flat top ends.

endPosition (*float*)
Value of $\hat{\mathbf{u}} \cdot \mathbf{x}$ at which the ramp ends.

startAmplitude (*float*)
Amplitude for $\hat{\mathbf{u}} \cdot \mathbf{x} < startPosition$.

endAmplitude (*float*)
Amplitude for $\hat{\mathbf{u}} \cdot \mathbf{x} > endPosition$.

flattopAmplitude (*float*)
Amplitude for $startFlattop < \hat{\mathbf{u}} \cdot \mathbf{x} < endFlattop$.

Example cosineFlattop Block

```
<STFunc cosFT>
  kind              = cosineFlattop
  direction         = [1.  0.  0.]
  startPosition     = 0.0
  startFlattop      = 0.1
  endFlattop        = 1.
  endPosition       = 1.
  startAmplitude    = 0.
  endAmplitude      = 0.
  flattopAmplitude = 1.0e9
</STFunc>
```

cosineRamp

Works with VSimBase, VSimEM, VSimPD, VSimPA, and VSimMD licenses.

Function for an initial ramp.

cosineRamp Parameters

direction (*float vector*)
Direction of the ramp (unit vector $\hat{\mathbf{u}}$ of gradient).

startPosition (*float*)
Value of $\hat{\mathbf{u}} \cdot \mathbf{x}$ at which the ramp starts.

endPosition (*float*)
Value of $\hat{\mathbf{u}} \cdot \mathbf{x}$ at which the ramp ends.

startAmplitude (*float*)
Amplitude for $\hat{\mathbf{u}} \cdot \mathbf{x} < startPosition$.

endAmplitude (*float*)
Amplitude for $\hat{\mathbf{u}} \cdot \mathbf{x} > endPosition$.

Example cosineRamp Block

```
<STFunc component0>
   kind = cosineRamp
   direction = [1. 0. 0.]
   startPosition = 0.0
   endPosition    = 0.1
   startAmplitude = 0.
   endAmplitude    = 1.0e17
 </STFunc>
```

expression (STFunc)

Works with VSimBase, VSimEM, VSimPD, VSimPA, and VSimMD licenses.

Function that can be defined by an arbitrary mathematical expression.

expression Parameters

expression (*string*)

Expression to be evaluated, involving the arithmetic operators + (addition), - (subtraction), * (multiplication), / (division), and ** (exponentiation), and the below functions of position and/or time:

Function	Mathematical Description	General Description
pow(x,y)	x^y	exponential, arbitrary base
exp(x)	e^x	exponential, base e
sin(x)	$\sin(x)$	sine
cos(x)	$\cos(x)$	cosine
tan(x)	$\tan(x)$	tangent
asin(x)	$\arcsin(x)$, $x \in [-1, 1]$	inverse sine
acos(x)	$\arccos(x)$, $x \in [-1, 1]$	inverse cosine
atan(x)	$\arctan(x)$, $x \in [-\pi/2, \pi/2]$	inverse tangent
atan2(y,x)	$\arctan(y/x)$; x and y not both 0,	inverse tangent, returns angles in correct quadrant
	$x = 0$ returns $\pm\pi/2$	
sinh(x)	$\sinh(x)$	hyperbolic sine
cosh(x)	$\cosh(x)$	hyperbolic cosine
tanh(x)	$\tanh(x)$	hyperbolic tangent
ln(x)	$\log_e(x)$	logarithm, base e
log(x)	$\log_e(x)$	logarithm, base e
log10(x)	$\log_{10}(x)$	logarithm, base 10
mod(x,y)	$x - \lfloor x/y \rfloor y$	floating point remainder
inv(x)	$-x$	additive inverse
H(x)	$H(x) = \begin{cases} 0 & , x < 0 \\ 0.5 & , x = 0 \\ 1 & , x > 0 \end{cases}$	Heaviside step function
J0(x)	$J_0(x)$	Bessel function of the first kind, order 0
J1(x)	$J_1(x)$	Bessel function of the first kind, order 1
J2(x)	$J_2(x)$	Bessel function of the first kind, order 2
J3(x)	$J_3(x)$	Bessel function of the first kind, order 3
abs(x)	$\|x\|$	absolute value
sqrt(x)	$\sqrt{x}$	square root
rand(x)		uniform random number in $[0, 1)$, independent of x
gauss(x,y)		Gaussian random number with standard deviation x and mean y
ceil(x)	$\lceil x \rceil$	smallest integer not less than x
floor(x)	$\lfloor x \rfloor$	largest integer not greater than x
min(x,y)	x if $x \leq y$, else y	minimum
max(x,y)	x if $x \geq y$, else y	maximum

Example expression Block

```
<STFunc component0>
  kind = expression
  expression = 100.*sin(2.0e9*t)
</STFunc>
```

feedbackSTFunc

Works with VSimPD and VSimMD licenses.

Function that pulls its values from a `feedbackDesired` history. See *feedbackDesired* history.

feedbackSTFunc Parameters

kind
For **STFunc**, must be set to feedbackSTFunc.

feedback
Points to the feedback history being used.

expression
For any space time function, this value is a guess as to what the adjustable term (current for example) will look like to produce the desired quantity (voltage in this case). The value obtained from the `feedback` is then multiplied by the value of *expression* to produce the resulting value of `addjustableTerm`. The value of `feedback` changes at every time step, but ideally will approach a constant value.

Example feedbackSTFunc Block

```
<STFunc adjustableTerm>
   kind = feedbackSTFunc
   feedback = feedbackHistory
   expression = 0.5*J_DRIVE*tanh(t/T0)
</STFunc>
```

gaussian

Works with VSimBase, VSimEM, VSimPD, VSimPA, and VSimMD licenses.

Function for a Gaussian.

gaussian Parameters

width (*float vector*)
Width of the Gaussian.

origin (*float*)
Origin of the center of the Gaussian.

velocity (*float*)
Velocity of the pulse.

amplitude (*float vector*)
Amplitude of the wave.

Example gaussian Block

```
<STFunc relMacroFluxFunc>
  kind = gaussian
  widths = [1. 0. 1.]
  origin = [0. 0. 0.]
  velocity = [0. 1.e5 0.]
  amplitude = 1.
</STFunc>
```

gaussianPulse

Works with VSimBase, VSimEM, VSimPD, VSimPA, and VSimMD licenses.

Function for a sinusoidal pulse in the form of a Gaussian beam, modulated by a Gaussian envelope longitudinally. The longitudinal envelope rises adiabatically on the tails of the Gaussian to avoid discontinuous truncation.

gaussianPulse Parameters

omega (*required float*)
Angular frequency of the wave ω.

k (*required float vector*)
The wave vector $\mathbf{k}$.

amplitude (*required float*)
Amplitude of the wave.

phase (*optional float, default = 0*)
The phase, added to $(\mathbf{k} \cdot \mathbf{x} - \omega t)$.

origin (*required float vector*)
The initial location of the peak of the pulse. This is usually located a distance `widths[0]` outside the simulation domain, so that the value of the function is initially 0 everywhere in the simulation. The pulse then propagates into the simulation domain.

widths (*required float vector*)
Widths of the pulse:

`widths[0]`

Full half-width of the pulse in the longitudinal direction (i.e. along $\mathbf{k}$). The trauncated Gaussian reaches 0 this distance away from the peak.

Note: The Gaussian envelope must be truncated, which implies a finite starting amplitude. Because a finite starting amplitude is unphysical and can significantly perturb the simulation, the truncated Gaussian is brought smoothly to zero with a simple cubic switching function. Hence, 90% of the envelope is a true Gaussian, but the 10% at the leading and trailing edges has a different functional form. Any ramifications should be negligible if the full half-width of the pulse is at least 3 RMS.

`widths[1]`

The width of the Gaussian beam in the direction perpendicular to $\mathbf{k}$ in the plane of $\mathbf{k}$ and $\hat{\mathbf{x}}$; if $\mathbf{k}$ is parallel to $\hat{\mathbf{x}}$, this is the width in the y direction. The width is the full-width at the $1/\sqrt{e}$ points (of the function value, not function squared) at the beam waist.

`widths[2]`

The width of the Gaussian beam in the direction perpendicular to both $\mathbf{k}$ and $\hat{\mathbf{x}}$; if $\mathbf{k}$ is parallel to $\hat{\mathbf{x}}$, this is the width in the z direction. The width is the full-width at the $1/\sqrt{e}$ points (of the function value, not function squared) at the beam waist.

vg (*required float*)
Group velocity, by which the origin of the envelope is moved.

waistDisplacement (*required float*)
Location of the pulse focus along its direction of propagation, measured from the initial peak location specified by *origin*.

L_fwhm (*required float*)
Longitudinal full width at half-maximum of the pulse intensity (function squared).

keepon (*optional integer*, *default = 0 (false)*)
If true, the pulse does not fall after getting to its peak value.

Example gaussianPulse Block

```
<STFunc component1>
  kind      = gaussianPulse
  omega     = OMEGA
  k         = [K_LASER  0.  0.]
  amplitude = EPUMP
  origin    = [XSTARTPUMP  0.      0.    ]
  widths    = [WXPUMP      WYPUMP  WYPUMP]
  vg        = LIGHTSPEED
  waistDisplacement = STARTFLAT
# Full length at half max of pulse envelope
  L_fwhm    = L_FWHM
</STFunc>
```

halfSinePulse

Works with VSimBase, VSimEM, VSimPD, VSimPA, and VSimMD licenses.

Function for a sinusoidal pulse in the form of a Gaussian beam, modulated by a longitudinal half-sine function.

halfSinePulse Parameters

omega (*required float*)
Angular frequency of the wave ω.

k (*required float vector*)
The wave vector $\mathbf{k}$.

amplitude (*required float*)
Amplitude of the wave.

phase (*optional float*, *default = 0*)
The phase, added to $(\mathbf{k} \cdot \mathbf{x} - \omega t)$.

`origin` (*required float vector*)
The initial location of the peak of the pulse. This is usually located a distance `widths[0]` outside the simulation domain, so that the value of the function is initially equal to zero everywhere in the simulation. The pulse then propagates into the simulation domain.

`widths` (*required float vector*)
The widths of the pulse:

`widths[0]`

Full half-width of the pulse in the longitudinal direction (i.e. along $\mathbf{k}$). The half-sine function reaches zero at this distance away from the peak.

`widths[1]`

The width of the Gaussian beam in the direction perpendicular to $\mathbf{k}$ in the plane of $\mathbf{k}$ and $\hat{\mathbf{x}}$; if $\mathbf{k}$ is parallel to $\hat{\mathbf{x}}$, this is the width in the y direction. The width is the full-width at the $1/\sqrt{e}$ points (of the function value, not function squared) at the beam waist.

`widths[2]`

The width of the Gaussian beam in the direction perpendicular to both $\mathbf{k}$ and $\hat{\mathbf{x}}$; if $\mathbf{k}$ is parallel to $\hat{\mathbf{x}}$, this is the width in the z direction. The width is the full-width at the $1/\sqrt{e}$ points (of the function value, not function squared) at the beam waist.

`vg` (*required float*)
Group velocity, by which the origin of the envelope is moved.

`waistDisplacement` (*required float*)
Location of the pulse focus along its direction of propagation, measured from the initial peak location specified by *`origin`*.

`skewness` (*optional float*, *default = 0*)
Skewness in the pulse, which is a distortion so that the pulse rises more rapidly than it falls (negative skewness) or vice versa.

`keepon` (*optional integer*, *default = 0 (false)*)
If nonzero, the pulse does not fall after getting to its peak value.

Example halfSinePulse Block

```
<STFunc component0>
  kind = halfSinePulse
  omega = 3.0e7
  k = [5. 0. 0.]
  amplitude = 0.7
  origin = [0.  0.  0.]
  widths = [14.9896e-6  8.48528e-6  8.48528e-6]
  vg = 2.9998e8
  waistDisplacement = 0.2
</STFunc>
```

historySTFunc

Works with VSimPD and VSimMD licenses.

Function that pulls its values from a `feedbackMeasured` history. See *feedbackMeasured* history.

historySTFunc Parameters

kind
For **STFunc** must be set to historySTFunc.

feedback
Points to the feedbackMeasured history being used.

expression
For any space time function, this value will be used to adjust the measured quantity. The feedbackMeasured `history` is then multiplied by the value of *expression* to produce the resulting value of `STFunc`.

Example historySTFunc Block

```
<STFunc currentDensityFunc>
  kind = historySTFunc
  feedback = feedbackTopWallCurrent
  expression = $ 1.0 / TOP_WALL_AREA $
</STFunc>
```

ignoreArgFunc

Works with VSimBase, VSimEM, VSimPD, VSimPA, and VSimMD licenses.

Provides an STFunc block that is an interface to an NAFunc block.

ignoreArgFunc Parameters

NAFunc (*code block*, *required*)
IgnoreArgFunc block must contain an NAFunc block that has the name `func`.

ignoreArgFunc STFunc Kind Example

```
<STFunc component0>
    kind = ignoreArgFunc
    <NAFunc func>
        kind = stretcher
        minvalue = VX_MIN_DF
        maxvalue = VX_MAX_DF
        seed = VX_SEED
    </NAFunc>
</STFunc>
```

inverseFunc

Works with VSimBase, VSimEM, VSimPD, VSimPA, and VSimMD licenses.

Defines a function that is an inverse of another function.

inverseFunc Parameters

STFunc (*block*)
The function to be inverted. Must be a named function.

Example inverseFunc Block

```
<STFunc sourceFunc>
  kind = inverseFunc
  <STFunc function>
    kind = expression
    expression = ((x+0.5*LX)*(y+0.5*LY))^2
  </STFunc>
</STFunc>
```

leakychannel

Works with VSimBase, VSimEM, VSimPD, VSimPA, and VSimMD licenses.

Function that is parabolic in radius, then drops linearly to zero.

leakychannel Parameters

direction (*float vector*)
Direction of the ramp (unit vector **û** of gradient).

channelPosition (*float*)
Position of the center of the channel.

maxParabRadius (*float*)
Radius (transverse to **û**) at which the channel starts turning from parabolic to linear.

maxRadius (*float*)
Radius (transverse to **û**) at which the channel vanishes.

centerAmplitude (*float*)
Value at the center of the channel.

quadCoef (*float*)
Coefficient of the quadratic term.

expanFunc (*float*)
Function whose values on axis give the expansion of the channel, with corresponding decrease of the radius.

Example leakyChannel Block

```
<STFunc channel>
  kind              = leakyChannel
  direction         = [1. 0. 0.]
  channelPosition   = [0. 0. 0.]
  maxParabRadius    = MAXPARABRADIUS
  maxRadius         = MAXRADIUS
  centerAmplitude   = DENSRAT
```

(continues on next page)

(continued from previous page)

```
  quadCoef          = QUADCOEF
</STFunc>
```

multFunc

Works with VSimBase, VSimEM, VSimPD, VSimPA, and VSimMD licenses.

Defines a function that is a product of one or more other functions.

multFunc Parameters

`STFunc` (*block*)
One of the functions to be multiplied.

Example multFunc Block

```
<STFunc sourceFunc>
  kind = multFunc
  <STFunc func1>
    kind = expression
    expression = x
  </STFunc>
  <STFunc func2>
    kind = expression
    expression = y
  </STFunc>
</STFunc>
```

planeWavePulse

Works with VSimBase, VSimEM, VSimPD, VSimPA, and VSimMD licenses.

A plane wave modulated by a Gaussian transversely and a half-sine longitudinally.

planeWavePulse Parameters

`omega` (*required float*)
Angular frequency of the wave ω.

`k` (*required float vector*)
The wave vector $\mathbf{k}$.

`amplitude` (*required float*)
Amplitude of the wave.

`phase` (*optional float, default = 0*)
The phase, added to $(\mathbf{k} \cdot \mathbf{x} - \omega t)$.

origin (*required float vector*)
The initial location of the peak of the pulse. This is usually located a distance `widths[0]` outside the simulation domain, so that the value of the function is initially zero everywhere in the simulation. The pulse then propagates into the simulation domain.

widths (*required float vector*)
The widths of the pulse:

`widths[0]`

Full half-width of the pulse in the longitudinal direction (i.e. along $\mathbf{k}$). The half-sine function reaches zero at this distance away from the peak.

`widths[1]`

The width of the Gaussian beam in the direction perpendicular to $\mathbf{k}$ in the plane of $\mathbf{k}$ and $\hat{\mathbf{x}}$; if $\mathbf{k}$ is parallel to $\hat{\mathbf{x}}$, this is the width in the y direction. The width is the full-width at the $1/\sqrt{e}$ points (of the function value, not function squared).

`widths[2]`

The width of the Gaussian beam in the direction perpendicular to both $\mathbf{k}$ and $\hat{\mathbf{x}}$; if $\mathbf{k}$ is parallel to $\hat{\mathbf{x}}$, this is the width in the z direction. The width is the full-width at the $1/\sqrt{e}$ points (of the function value, not function squared).

vg (*required float*)
Group velocity, by which the origin of the envelope is moved.

skewness (*optional float*, *default = 0*)
Skewness in the pulse.

keepon (*optional integer*, *default = 0 (false)*)
If true, the pulse does not fall after getting to its peak value.

Example halfSinePulse Block

```
<STFunc function>
  kind = planeWavePulse      # Function used for E_z
  omega = OMEGA              # Angular frequency parameter for function
  k = [KAY  0  0]            # k-vector parameter for function
  amplitude = EWAVE
  phase = 1.57
  widths = [ 5.e-6  1.e-5  1.e-5]    # widths of pulses, along k, transverse
  origin = [-5.e-6  0.e-5  0.e-5]    # Origin parameter for function
  vg = LIGHTSPEED            # Group velocity parameter for function
</STFunc>
```

radialCosChannel

Works with VSimBase, VSimEM, VSimPD, VSimPA, and VSimMD licenses.

Function for an initial ramp into a region of a channel.

radialCosChannel Parameters

direction (*float vector*)
Direction of the ramp (unit vector $\hat{\mathbf{u}}$ of gradient).

channelPosition (*float*)
Position of the center of the channel.

startRadius (*float*)
Radius (transverse to $\hat{\mathbf{u}}$) at which the channel starts to turn into the outer region.

endRadius (*float*)
Radius (transverse to $\hat{\mathbf{u}}$) at which the channel stops changing and the outer region begins.

startAmplitude (*float*)
Amplitude for $\hat{\mathbf{u}} \cdot \mathbf{x} < startPosition$.

endAmplitude (*float*)
Amplitude for $\hat{\mathbf{u}} \cdot \mathbf{x} > endPosition$.

scalarFunc

Works with VSimPD and VSimMD licenses.

Function that pulls its values from a scalar. See scalar in multiFields.

scalarFunc Parameters

kind
For **STFunc** must be set to scalarFunc.

scalar (*required string*)
The name of the scalar being used.

Example scalarFunc Block

```
<STFunc fA5>
   kind = scalarFunc
   scalar = A5
</STFunc>
```

sinePlaneWave

Works with VSimBase, VSimEM, VSimPD, VSimPA, and VSimMD licenses.

Plane wave pulse based on a sine wave.

sinePlaneWave Parameters

omega (*required float*)
Angular frequency of the wave ω.

k (*required float vector*)
The wave vector $\mathbf{k}$.

amplitude (*required float*)
Amplitude of the wave.

phase (*optional float, default = 0*)
The phase, added to $(\mathbf{k} \cdot \mathbf{x} - \omega t)$.

Example simplePlaneWave Block

```
<STFunc component0>
  kind = sinePlaneWave
  omega = OMEGA_RF
  k = [0. 0.01 0.]
  amplitude = 1.0e5
  phase = 0.0
</STFunc>
```

sumFunc

Works with VSimBase, VSimEM, VSimPD, VSimPA, and VSimMD licenses.

Defines a function that is a sum of one or more other functions.

sumFunc Parameters

STFunc (*block*)
One of the functions to be summed.

Example sumFunc Block

```
<STFunc sourceFunc>
  kind = sumFunc
  <STFunc func1>
    kind = expression
    expression = x
  </STFunc>
  <STFunc func2>
    kind = expression
    expression = y
  </STFunc>
</STFunc>
```

stPyFunc

Works with VSimBase, VSimEM, VSimPD, VSimPA, and VSimMD licenses.

Function defined by a Python script, which is contained in a separate file. The parameter name is used to point to this file.

stPyFunc Parameters

name (*string*)
Name of the function.

Example stPyFunc Block

Python block in a Python file with same name as the input file.

```
def imagAmp(x, y, z, t):
   if NDIM == 1:
      return 0.0
   elif NDIM == 2:
      return gausBeam2D(LIGHTSPEED * (t - T_WAIST), x - X_WAIST, y).imag
   else:
      return gausBeam3D(LIGHTSPEED * (t - T_WAIST), x - X_WAIST, y, z).imag
```

Block in Vorpal referencing the Python function defined in the Python file.

```
<InitialCondition ImagA>
  lowerBounds = [ 0   0   0]
  upperBounds = [NX NY NZ]
  kind = variable
  components = [1]
  <STFunc component1>
    kind = stPyFunc
    name = imagAmp
  </STFunc>
</InitialCondition>
```

tagGen

Works with VSimPD, VSimPA, and VSimMD licenses.

Very specialized function used only to generate tags for tagged particles species.

tagGen Parameters

The tagGen function has no additional parameters.

Example tagGen Block

```
<STFunc component3>
  kind = tagGen
</STFunc>
```

userFuncExpression

Works with VSimBase, VSimEM, VSimPD, VSimPA, and VSimMD licenses.

Defines a function that wraps an *Expression Block*.

userFuncExpression Parameters

`Expression` (*block*)
The Expression wrapped by the STFunc.

Example userFuncExpression Block

```
<STFunc function>
  kind = userFuncExpression
  <Expression updateFunction>
    expression = min(vector(R_CAV^2 -x^2-y^2, z - Z_START +1e-6*DZ, Z_START+LZ+1e-
↪6*DZ-z))
  </Expression>
</STFunc>
```

stRgnMask

Assigns properties of a fluid to an area or volume specified by an STRgn.

stRgnMask Parameters

region (*string*, *required*)
Name of the space-time region that will serve as the element of the array.

Example stRgnMask Block

```
<STRgn ellipsoid1>
  kind = ellipsoid
  center = [0.0  0.0  0.0]
  semiaxes = [1.0  0.5  0.3]
</STRgn>

<Fluid testFluid>
  kind = neutralGas
  gasKind = H
  <InitialCondition density>
    kind = variable
    lowerBounds = [-1 -1 -1]
    upperBounds = [NX NY NZ]
    components = [0]
    <STFunc component0>
      kind = stRgnMask
      region = ellipsoid1
    </STFunc>
  </InitialCondition>
</Fluid>
```

3.17.5 SVTFunc

SVTFunc Block

Space-Velocity-Time function objects are used for defining phase space functions, such as those needed for delta-f PIC simulations.

SVTFunc Kinds

- *maxwellian*

maxwellian

Works with VSimBase, VSimEM, VSimPD, VSimPA, and VSimMD licenses.

An SVTFunc created using a Maxwellian function with `kind = maxwellian`.

maxwellian Parameters

`density`
Density of the plasma in the region of simulation.

`velocityDim`
Number of velocity coordinates.

`densGrad_x`
x-derivative of the density function specified above.

`densGrad_y`
y-derivative of the density function specified above.

`densGrad_z`
z-derivative of the density function specified above.

`vthermal`
Thermal velocity of the particles (related to energy level).

`vdrift_x`
Specifies drift velocity of particles in x-direction.

`vdrift_y`
Specifies drift velocity of particles in y-direction.

`vdrift_z`
Specifies drift velocity of particles in z-direction.

Example maxwellian Block

Example showing that all parameters (except .. attribute:: velocityDim‘) are specified through STFuncs of kind .. attribute:: expression‘ or .. attribute:: constantFunc‘.

```
<SVTFunc equilibDist>

  kind = maxwellian
  velocityDim = 3

  <STFunc density>
    kind = expression
    expression = DENSITY_P*x/LX
  </STFunc>

  <STFunc densGrad_x>
    kind = expression
```

(continues on next page)

(continued from previous page)

```
    expression = DENSITY_P/LX
  </STFunc>

  <STFunc densGrad_y>
    kind = constantFunc
    amplitude = 0.
  </STFunc>

  <STFunc densGrad_z>
    kind = constantFunc
    amplitude = 0.
  </STFunc>

  <STFunc vthermal>
    kind = constantFunc
    amplitude = ELECTHERMSPEED
  </STFunc>

  <STFunc vdrift_z>
    kind = constantFunc
    amplitude = 0.
  </STFunc>

</SVTFunc>
```

3.17.6 UserFuncs, Expressions, and related functions

Introduction to UserFuncs and Expressions

A *UserFunc Block* is like an STFunc or OAFunc or NAFunc, etc., but the UserFunc framework is more general and powerful (and complicated). UserFuncs can call other (user-defined) UserFuncs, and they allow more general function signatures, with vector arguments and results

This is an example of a very simple UserFunc that takes a single scalar argument (of float type) and squares it:

```
<UserFunc square>
  kind = expression
  inputOrder = [b]
  <Input b>
    kind = arbitraryVector
    types = [float]
  </Input>
  expression = b^2
</UserFunc>
```

A few terms, very briefly:

- An <Expression> block is like a UserFunc of `kind = expression`, except that the <Expression> block does not have to specify the inputOrder or <Input> blocks. See *expression*.
- An *Input Block* block describes the (vector-)length and types of an input argument to the function. Along with the `inputOrder` attribute, the <Input> blocks give the input signature of the function: the number and types of arguments.
- *Local UserFuncs* are UserFunc (sub)blocks that appear within a UserFunc block (usually of `kind = expression`).

- A *Term Block* is a cross between an <Input> and a local <UserFunc>. It is a term that can be used (e.g., in an expression), but its value is not passed to the function as an argument, but rather found by evaluating a local function. Even when a Term appears more than once (or not at all) in an expression, the Term is evaluated exactly once; in contrast, a local UserFunc may be evaluated once for each time it appears, or not at all if its result is not needed. Moreover, a local UserFunc can be called multiple times in the same expression with different arguments.

UserFunc Block

A block for functions specifed by the user

UserFuncs can be used to specify a function, e.g., as a string expression. While already very powerful and useful, UserFuncs are still an experimental feature that will require continued development, and the interfaces to them will probably change as they evolve.

A UserFunc is a vector-valued function that takes vector arguments; in general, a UserFunc can take an arbitrary number of vector arguments, and return a vector; each vector (in general) can contain any number of scalar elements.

Note: For convenience, we will often write vectors using brackets, e.g., [1.5, 2, -3, 4.1]. However, this bracket-notation is not recognized within a UserFunc expression, where a vector must be written `vector(1.5, 2, -3, 4.1)`.

UserFuncs distinguish 3 types of scalars: `float` (a floating point number), `integer`, and `boolean` (0 or 1).

Note: Type information helps avoid unexpected values before any evaluation is performed; type information is never used during the evaluation itself. In other words, if the number 1 is passed to a function, the result will never depend on whether 1 is a boolean, an integer, or a float; in particular, 1.0/2.0 = 1/2 = 0.5 as far as UserFuncs are concerned.

A function that takes an integer argument will also take a boolean; similarly, a function that takes a float argument will also take an integer or a boolean. The reverse is not true: e.g., a function that takes a boolean will not take a general integer or float.

A specific UserFunc has a signature that describes the number, lengths, and types of arguments, as well as the length and types of the result. Here are the signatures of some common functions:

$$\begin{aligned} \text{sin}: & \quad ([\text{float}]) \rightarrow [\text{float}] \\ ==: & \quad ([\text{float}], [\text{float}]) \rightarrow [\text{boolean}] \\ \text{floor}: & \quad ([\text{float}]) \rightarrow [\text{integer}] \end{aligned}$$

In these simple examples, each vector (argument or result) has only one element.

UserFuncs use 0-indexing: the first argument is argument 0; the first element of a vector has index 0.

Simple UserFuncs of `kind = expression` are easy to use when, for instance, they are employed in the same way as STFuncs. For example, suppose some attribute block expects a UserFunc that is a function of space and time, i.e., of scalars `x, y, z`, and `t`. Example input might be:

```
<OuterBlock someObjectThatNeedsAUserSpecifiedFunction>
  ...
  <UserFunc someName>
    kind = expression
    inputOrder = [t x y z]
    <Input t>
      kind = arbitraryVector
```

(continues on next page)

(continued from previous page)

```
      types = [float]
    </Input>
    <Input x>
      kind = arbitraryVector
      types = [float]
    </Input>
    expression = sin(3*x) * cos(0.2*t)
  </UserFunc>
</OuterBlock>
```

To learn how to use UserFuncs, it might be easiest to look first at examples (such as in *Local UserFuncs* and *Term Block*), and then refer to the following explanation as needed.

Kinds of UserFuncs

There are several kinds of UserFuncs: an `expression` simply takes a string expression in the input file.

An expression UserFunc block must include its input signature (`inputOrder` and the <Input> blocks), but most other UserFuncs have only one possible input signature and therefore don't require the input signature to be specified. (And <Expression> is like UserFuncs of `kind=expression` but require no signature; see *expression*.)

UserFuncs specified at the top-level of the input file are special because they can be called by any other UserFuncs (or <Expression>s) in the entire simulation.

- *constant* (a function returning a constant value)
- *distributedRandom* (a random number generator with arbitrary distribution)
- *expression* (a function specified by a string expression)
- *fieldFunc* (returns a field value)
- *gridFunc* (returns information about grid geometry)
- *gridBoundaryFunc* (returns information about surface geometry)
- *historyFunc* (returns values from a Tensor History)

Some of these kinds get their values from other simulation objects; when running parallel simulations, it is important to remember that the entire object may not be accessible to all processors. For example, a *fieldFunc* can in principle be used to look up a field value at any spatial location; however, each rank has access to just a part of the field—attempting to find the field value outside that range will halt the simulation. Some objects that use UserFuncs (notably *cellFuncHist*) offer backup-functions that can be called should the main UserFunc fail to evaluate.

UserFunc Parameters

General UserFuncs can take the following attribute:

`maxNumEvals` (*positive integer*, *optional*, *default = 128*)
The maximum number of simultaneous evaluations a function can perform (see *Speed and Multiple Evaluations of UserFuncs*). Increasing this number increases memory use and evaluation speed: 16–128 is recommended for `kind = expression`.

Expression Block

A block for functions specifed by the user.

Expressions are essentially UserFuncs of `kind = expression` except that they do not need to specify the `inputOrder` attributes, nor any <Input> blocks.

Expressions try to adapt their input signatures to what the surrounding block needs.

See *expression*

UserFunc Kinds

constant

Describes a function with a constant return value.

Constant UserFunc Parameters

UserFuncs with `kind=constant` take the following attribute:

`value` (*vector of floats*)
: The vector-valued result of the function.

Examples

This example shows a function that returns whether the point (x,y) falls within a rod, where the rod centers are specified in a list (and the rods extend in the z direction). (Such a list cannot be used directly in a UserFunc expression, though one can use the vector(. . .) function. However, here the constant function gets around that problem.)

```
$ ROD_RADII = 0.2
$ ROD_CENTERS_X = [1. 0.5   -0.5 -1. -0.5   0.5]
$ ROD_CENTERS_Y = [0. 0.866 0.866 0. -0.866 -0.866]
<UserFunc isInRod>
  kind = expression
  inputOrder = [ x y ]
  <Input x>
    kind = arbitraryVector
    types = [float]
  </Input>
  <Input y>
    kind = arbitraryVector
    types = [float]
  </Input>
  <UserFunc rcx>
    kind = constant
    value = ROD_CENTERS_X
  </UserFunc>
  <UserFunc rcy>
    kind = constant
    value = ROD_CENTERS_Y
  </UserFunc>
  expression = or( (x-rcx)^2 + (y-rcy)^2 < ROD_RADII^2 )
</UserFunc>
```

This expression uses folding and threading to return 1 if (x,y) is within at least one of the rods, or 0 if (x,y) is not within any of the rods. Briefly, the subtracting, squaring, addition, and less-than operators are threaded through the vector values of rcx and rcy, yielding a vector such as vector(0,0,0,1,0,0), indicating that (x,y) falls within the fourth rod (but not any of the other rods). The or-function is then folded over the vector, yielding 1 if any element of the vector is 1.

distributedRandom

A UserFunc that takes no arguments, and returns a random number with a user-given distribution.

distributedRandom Parameters

`seed` (*non-negative integer*, *optional*)
: A seed for Vorpal's default random number generator (used only if `<NAFunc numberSequence>` is not given). If seed is not given, a seed will be chosen using the current time.

`probDist` (*code block*, *required*)
: `<OAFunc probDist>` is an OAFunc (function of one argument, returning a scalar) specifying a probability distribution, including the domain (which describes the range of possible values resulting from this random number generator). An OAFunc of `kind = expression` is the usual choice here.

`maxRelErr` (*float*, *optional*, *default = 1e-2*)
: The probability distribution will be approximated with a relative error below this level, subject to `maxNumPoints`.

`maxNumPoints` (*positive integer*, *optional*, *default = 40961*)
: The maximum number of points at which to sample the probability distribution. The number of sample points will be increased, trying to attain `maxRelErr`, up to `maxNumPoints`.

`doComparison` (*bool*, *optional*, *default = false*)
: For diagnostic purposes only, compare the approximation of the <OAFunc probDist>, printing out (to stdout) how close the approximation is.

`printSampleValues` (*non-negative integer*, *optional*, *default = 0*)
: For diagnostic purposes only, the number of sample random numbers to be printed (to stdout) so the user can see if they are distributed as desired.

`numberSequence` (*code block*, *optional*)
: `<NAFunc numberSequence>` is an NAFunc code block that generates a random number; if not specified, Vorpal's default random number generator (uniform in [0,1)) will be used to generate the initial random number.

  Random numbers will be generated with the desired distribution only if `numberSequence` generates a number uniformly in [0,1).

  Using this code block makes the `seed` option irrelevant.

expression

Describes a function with an string expression, e.g., `sin(k*x)`.

A <UserFunc> block of `kind = expression` and an <Expression> block take all the same attributes, except that <Expression> blocks do not take the `inputOrder` attribute, nor <Input> blocks.

Generally, UserFuncs or Expressions are required by some object to compute a user-defined value; each object will specify whether it requires a UserFunc or an Expresssion. However, UserFuncs may also be defined at the top level of the input file; these UserFuncs may then be referenced in other UserFuncs throughout the input file. For example, one could define, at the top level, a `<UserFunc rotate>` that takes (x, y, z) and performs a specific three-dimensional rotation, returning (x', y', z'). The `rotate` function can then be called conveniently from other UserFuncs and Expressions. Expessions may not be defined at the top-level.

Expression Parameters

UserFuncs with `kind = expression` take several attributes in addition to the ones for general UserFuncs:

expression (*string*, *required*)
: The function expression, e.g., $x * y - sin(z)$. The following sections describe how to write an expression.

inputOrder (*vector of strings: UserFunc only*)
: The order of the arguments, by name; e.g., `inputOrder=[x y z numElephants]`. Each element of inputOrder should be the name of an Input block. (This attribute is not given to <Expression> blocks.)

Input (*code block*, *required in UserFuncs for each member of inputOrder*)
: For each input variable named in `inputOrder`, there must be a corresponding <Input> block with the same name as the variable; the <Input> block specified the length and types of each input. For example, if `inputOrder=[x y z numElephants]` then one would also specify:

```
<Input x>
  kind = arbitraryVector
  types = [float]
</Input>
<Input y>
  kind = uniformVector
  types = float
  length = 3
</Input>
<Input z>
  kind = arbitraryVector
  types = [boolean integer]
</Input>
<Input numElephants>
  kind = arbitraryVector
  types = [integer]
</Input>
```

to tell Vorpal that `x` is a scalar floating point number, `y` is a 3-vector of floats, `z` is a vector of length 2, the first element a boolean, the second an integer, and `numElephants` is a scalar integer.

See *Input Block*.

UserFunc (*code block*, *optional*)
: A local function, called by the expression. A local <UserFunc> (unlike a <Term>) can be called with different arguments in different places in the expression; if its result is not needed, it may not be evaluated at all.

See *UserFunc Block* and *Local UserFuncs*.

Term (*code block*, *optional*)
: A local UserFunc that is evaluated with the same arguments as the containing <UserFunc>; the <Term> is evaluated exactly once each time the <UserFunc> is evaluated, and the result of the <Term> can be used in the expression like an input variable. The attributes of a <Term> block are identical to those of a <UserFunc>.

See *Term Block*.

Most users will never need to worry about the following optimization options, which should rarely be altered from their default values:

optimize (*bool*, *default true*)
The default value for the following optimizations (unless one suspects a bug, only useMemory and shortCircuitUnneededArgs should ever be turned off; the other optimizations can only improve things)

useMemory (*bool*, *default = optimize*)
Whether functions that take a long time to evaluate (e.g., sin, but not +) should remember their previous argument and result, and re-use the result if the same argument is called twice in a row; invoking this optimization increases run-time evaluation overhead, but may save time if the same argument is passed for multiple evaluations (however, Vorpal is semi-intelligent about this optimization; if the same argument doesn't often appear twice in a row, it stops using the memory feature).

shortCircuitUnneededArgs (*bool*, *default = optimize*)
Whether certain functions (such as `if` and multiplication) should bypass arguments that don't matter. For example, `if(x > y, sin(x), cos(y))` needs to evaluate `sin(x)` but not `cos(y)` when `x > y`. Similarly, `H(x) * sin(x)` needs to evaluate `sin(x)` only if $x \geq 0$ (hence $H(x) \neq 0$). Like useMemory, this optimization may increase evaluation costs in some cases.

pruneConsts (*bool*, *default = optimize*)
Whether to replace constant sub-expressions with a single constant; e.g., whether to replace `3+4+cos(0)` with `8`.

pruneCSEs (*bool*, *default = optimize*)
Whether to perform common sub-expression elimination.

pruneIfs (*bool*, *default = optimize*)
Whether to bypass if-functions when the first argument can be determined (independent of the function arguments); for example, whether to replace `if(0 < 1, x, y)` with `x`.

pruneSelects (*bool*, *default = optimize*)
Whether to bypass select-functions when the indices (in the second argument) are constant and sequential.

Basic built-in scalar functions

UserFuncs specified at the top level of the simulation can be used in other expression UserFuncs (anywhere); and some UserFuncs recognize local functions (defined in the same UserFunc block as the `expression`). In addition, there are many built-in functions. The recognized unary scalar functions follow, along with descriptions of the non-standard ones:

`identity`	identity(x) = x
`inv`	inv(x) = $-x$
`sqr`	sqr(x) = x^2
`cube`	cube(x) = x^3
`sqrt`	
`sin`	
`cos`	
`tan`	
`exp`	
`asin`	
`acos`	
`atan`	(see also binary function `atan2(y,x)`)
`sinh`	
`cosh`	

Continued on next page

Table 3.2 – continued from previous page

`ln`	
`H`	H(x) = 0 if x<0, 1/2 if x=0, 1 if x>0
`J0`	Bessel function J_0
`J1`	Bessel function J_1
`J2`	Bessel function J_2
`J3`	Bessel function J_3
`abs`	
`erf`	
`rand`	rand(x) = a random number in [0,1) (x is irrelevant)
`gauss`	gauss(x) = a random number gaussian-distributed with std. dev. x (and mean 0)
`print`	the identity, but prints the result to stdout
`ceil`	
`floor`	
`modmod`	modmod(x) = y in (-0.5,0.5] such that y = x (mod 1)
`not`	the logical not operator: not(0) = 1, not(1) = 0
`bool`	bool(x)=1 for all x, except bool(0)=0
`int`	casts to an integer (floor and ceil are usually preferable)
`assertBool`	the identity, but raises an exception at run-time if the argument is not 0 or 1
`assertInt`	the identity, but raises an exception at run-time if the argument is not an integer

The following standard binary operators are recognized: `==`, `!=`, `>`, `>=`, `<`, `<=`, `+` (also `sum(x,y)`), `-`, `*` (also `prod(x,y)`), `/` (N.B. this is always float division), $\wedge$ and `**` (two notations for exponent). Other built-in binary functions are:

`min`	
`max`	
`sum`	sum(x,y) = x+y (addition)
`prod`	prod(x,y) = x*y (multiplication)
`pow`	pow(x,y) = x^y
`mod`	mod(x,y) = x mod y (N.B.: mod(-4,3)=-1, mod(4,-3)=1, mod(-4,-3)=-1.)
`atan2`	atan2(y,x) = arctan(y/x)
`gaussRand`	gaussRand(x,y) is a random number, gaussian-distributed with mean y and std. dev. x
`and`	logical and
`or`	logical or

Built-in functions generally have obvious argument and return types. For example, logical operators (such and `not` and `and`) take only boolean arguments and return booleans; `floor` will take a float and return an integer. Built-in functions are fairly smart about argument and return types; for example, the plus operator will return a float if either of its arguments are float-type, but will return an integer if both of its arguments have boolean or integer types.

Built-in functions for use with GridBoundaries

There are a handful of special functions meant for dealing with results related to whether something (such as a point or a cell) is inside our outside a GridBoundary, or whether something is cut by the GridBoundary.

In the following, x and y are "interiorness" values, representing the "interiorness" of a point or a cell or some other region. See *gridBoundaryFunc* for sources of interiorness values. Interiorness values are integers that represent whether something is inside, outside, or on (or cut by) a boundary: one should use the following functions rather than interpreting the values, but typically, 1 is interior, 0 is on the boundary, and -1 is exterior.

Usually (when we remember) we use the language "inside" and "outside" when we want to separate objects into 2 disjoint categories; and we use “interior,” “exterior,” and “cut by the boundary” when we want to separate objects into 3 disjoint categories.

isInside(x)	returns 1/0: whether x is inside a boundary
isOutside(x)	returns not(isInside(x))
isInterior(x)	returns 1/0: whether x is inside but not on or cut by boundary
isExterior(x)	returns 1/0: whether x is outside but not on or cut by boundary
isCutByBndry(x)	returns 1/0: whether x is on or cut by the boundary
onSameSideOfBndry(x, y)	returns 0/1 if x and y are on [different sides/the same side] of a boundary, according to isInside()

Special built-in functions

Some special functions are: `if, vector, select, and len`.

- `if(a, x, y)` returns x if $a == 1$ (true) and y if $a == 0$ (false). x and y must have the same length, and a must be a scalar boolean value.
- The vector functions concatenates vectors (taking any number of arguments, each of any length and types)

$$\text{vector}([x_0, \dots, x_k], [y_0, \dots, y_\ell], \dots, [z_0, \dots, z_m]) = \quad [x_0, \dots, x_k, y_0, \dots, y_\ell, \dots, z_0, \dots, z_m].$$

  The bracket notation `[0,1,2]` is not recognized by UserFunc expressions; we simply use it for convenience here. In an expression, one must write `vector(0,1,2)`.

- The `select` function selects certain elements from a vector; it takes 2 arguments, a vector of arbitrary types, and a vector of arbitrary (non-negative) integers:

$$\text{select}([x_0, \dots, x_k], [i_0, \dots, i_n]) = [x_{i_0}, \dots, x_{i_n}].$$

- The `len` function returns the (integer) length of an arbitrary vector.

$$\text{len}([x_0, x_1, \dots, x_{k-1}]) = k.$$

Special treatment of scalar functions for vector arguments

Functions defined to be scalar (taking arguments of length 1 and returning a vector of length 1) are automatically altered to take non-scalar arguments in two very natural ways: threading and folding (terms from Mathematica).

Threading: For an example of threading, we “thread” the scalar function $\sin(x) = y$ through its vector argument:

$$\sin([x_0, \dots, x_n]) = [\sin(x_0), \dots, \sin(x_n)].$$

Multi-argument scalar functions can be threaded if each argument is a vector of the same length or length 1. Thus the scalar plus function, $x + y = z$ can be used as follows:

$$[x_0, \dots, x_n] + [y_0, \dots, y_n] = [x_0 + y_0, \dots, x_n + y_n]$$

or

$$[x_0, \dots, x_n] + [y] = [x_0 + y, \dots, x_n + y].$$

Folding: Folding allows any binary scalar function $f(x, y) = z$ to take a single vector argument of arbitrary length and return a scalar value:

$$f([v, w, x, \ldots, y, z]) = f(f(\cdots f(f(v, w), x) \ldots, y), z).$$

By definition, a folded function applied to a single scalar is the identity:

$$f([x]) = [x].$$

For example, the `sum` function is simply an alias for scalar plus: $\texttt{sum}(x, y) = x + y$. Therefore,

$$\text{sum}([x_0, \ldots, x_n]) = \sum_{i=0}^{n} x_i.$$

The `prod` function is an alias for scalar times, and so can be folded to calculate the product of all elements of a vector. Folding may also be especially useful with the `and, or, min,` and `max` functions.

Defining $f(x) = x$ for a binary scalar function f and scalar argument x makes sense for binary functions that one typically wants to fold; for example, $\texttt{sum}(x) = x$, $\texttt{prod}(x) = x$, $\texttt{min}(x) = x$. Be forewarned, however, that this behavior may sometimes hide a mistake; if one has a (scalar) function of 2 scalar arguments $f(x, y)$, and accidentally calls it with a single scalar argument, $f(x)$, then Vorpal treats that as a folded function (which is the identity, $f(x) = x$), rather than giving an error message as usual when a function is called with the wrong number of arguments.

A word on optimization

Expression UserFuncs are pretty good at performing simple optimizations, so it is generally recommended to write expressions in a way that is simplest to read and understand, rather than trying to guess which of several equivalent expressions will evaluate fastest.

For example, constant expressions are evaluated: `x + 3*4*0.1 + y` will be reduced to `x + 1.2 + y`.

Common sub-expressions are evaluated only once, so the `sin` function is evaluated only once in the expression `sin(x) + exp(sin(x))`.

Functions that take a relatively long time to evaluate (such as transcendental functions) will remember the previous argument and result, and avoid a new evaluation if the argument did not change. This memory feature tends to be less helpful when performing multiple simultaneous evaluations (see *Speed and Multiple Evaluations of UserFuncs*).

Also, the expression evaluator can sometimes determine when part of an expression need not be evaluated; in such a case, it can skip (or short-circuit) an evaluation whose result is unneeded. The most important example is the special `if(a, expr1, expr2)` function—-if $a = 1$ then `expr1` will be evaluated, but not `expr2`, and vice-versa if $a = 0$. Again, this feature is less helpful when performing multiple simultaneous evaluations (see *Speed and Multiple Evaluations of UserFuncs*).

fieldFunc

A UserFunc that returns the value of a <Field> at the given position.

A `kind=fieldFunc` UserFunc takes a single argument, a spatial position (a vector of floats with as many components as spatial dimensions).

fieldFunc parameters

`field` (*required string*)
The name of a <Field> to query for information (It may be necessary to qualify the field name; for example, if <Field yeeE> appears within <MultiField emField>, then `field=emField.yeeE` should be specified.)

result (*string*, *required*)
Specifies what information about the <Field> the function will return; two results are currently available, `fieldValue` and `zeroFieldValue`. Depending on the result, different attributes apply.

Following is a list of results, the length and type of the vector returned, and the extra attributes (if any) that pertain to each specific result.

zeroFieldValue
`result = zeroFieldValue` returns a vector of zeros with the same length as the number of components in the field. This can be useful in anif-statement. For example, it would be useful for a function that does the following–if a cell is interior to a boundary, then return the (vector) field value in that cell, otherwise return all zeros.

fieldValue
`result = fieldValue` returns the (vector) value of a field at the given point in space. See below for more detail. Returns as many floats as components in the field; or 1 float if `component` is specified.

component (*integer*, *optional*)
The desired component of the field to be returned (if not specified, all components of the field are returned in a vector)

interpolationOrder (*non-negative integer*, *default = 1*)
The order of interpolation used to find the field value at a point; e.g., 1 means linear interpolation, which has second-order error in the length of a grid cell. N.B. the higher interpolationOrder is, the more field values from surrounding cells will be needed to perform the interpolation. E.g., when interpolating a field value to a point in cell [5] (in a 1D simulation), interpolationOrder=0 requires only the field value in one cell (probably cell [5]), whereas interpolationOrder = 6 might require field values from cells [2] through [8].

gridBoundary (*string*, *optional*)
The name of a <GridBoundary>; the field will be interpolated or extrapolated avoiding cells near the gridBoundary; that is, the gridded field values used to interpolate/extrapolate to the given point will be taken from cells far (enough) away from the boundary.

polation (*string*)
Required when `gridBoundary` is specified, with choices `fromInOrOutside`, `fromInside`, `fromOutside`. Whether field values will be interpolated/extrapolated from cells inside or inside the boundary (if fromInOrOutside, cells will be used on the same side of the boundary as the position at which the field value is desired).

boundaryCondition (*string*, *optional*)
Choices: `none`, `dirichlet`, `neumann`, `electricAtPEC`, `magneticAtPEC` with default = `none`. Except `none`, these options are not generally recommended, because they tend not to do exactly what one wishes. This attribute specifies a boundary condition to be used to extrapolate field values (near) to the given gridBoundary surface. The choice `none` is the same as specifying no boundaryCondition. Choices `dirichlet` or `neumann` assume that all field components, or the normal-derivatives of all field components, vanish at the boundary. For 3-vector fields (intended for electric and magnetic fields), the choice `electricAtPEC` assumes boundary conditions appropriate for an electric field at a perfectly conducting metal boundary (or for a magnetic field at a magnetically-conducting boundary); the choice `magneticAtPEC` is appropriate for a magnetic field at a perfectly-conducting metal boundary (or for an electric field at perfect magnetic conductor). The latter choices are appropriate for divergenceless

fields only.

gridFunc

UserFuncs of `kind = gridFunc` return a result with information about a <Grid> object.

UserFuncs of `kind = gridFunc` take one argument, a cell index (a vector of integers with as many elements as spatial dimensions), and return information about that grid cell, depending on the `result` attribute.

gridFunc parameters

grid
: The name of a `<Grid>` object to query for information.

result (*string*, *required*)
: Specifies what information the function should return; depending on `result`, different attributes will apply.

  Following is a list of results, the length and type of the vector returned, and the extra attributes (if any) that pertain to each specific result.

  nodePos (*returns NDIM floats*)
  : `result = nodePos` returns the position of the cell node.

  edgeCenterPos (*NDIM floats*)
  : `result = edgeCenterPos` returns the position of the center of an edge of the cell (the edge that touches the cell node).

    dir (*required*, *either 0*, *1*, *or 2*)
    : The direction of the desired edge; e.g., 0 for the cell edge parallel to x that touches the cell node.

  faceCenterPos (*NDIM floats*)
  : `result = faceCenterPos` returns the position of the center of a face of the cell (the face that touches the cell node).

    dir (*required*, *either 0*, *1*, *or 2*)
    : The direction normal to the desired face; e.g., 2 for the cell face normal parallel to x and y that touches the cell node.

  cellCenterPos (*NDIM floats*)
  : `result = cellCenterPos` returns the position of the cell center.

  cellVolume (*1 float*)
  : `result = cellVolume` returns the volume of the cell.

gridBoundaryFunc

UserFuncs of `kind = gridBoundaryFunc` return a result with information a `<GridBoundary>` object, i.e., information related to a surface (described by the `<GridBoundary>`).

UserFuncs of `kind = gridBoundaryFunc` take one argument, a cell index (a vector of integers with as many elements as spatial dimensions), and return information about the GridBoundary surface in that cell, depending on the `result` attribute.

gridBoundaryFunc parameters

gridBoundary (*string*, *required*)
The name of the `<GridBoundary>` object to query for information.

result (*string*, *required*)
Specifies what information the function should return; depending on `result`, different attributes will apply.

gridBoundaryFunc results that return an "interiorness" value specify whether a region is inside, outside, or cut by the boundary; these results should, if possible, be used as arguments to the functions isInside, isInterior, isOutside, isExterior, isCutByBndry (see *Built-in functions for use with GridBoundaries*). In the following, references to a node, a face, or an edge, refer to the node, face, or edge of the cell specified by the CellFunc argument (and also the `dir` attribute if appropriate).

Following is a list of results, the length and type of the vector returned, and the extra attributes (if any) that pertain to each specific result.

nodeInteriorness (*returns 1 integer*)
`result = nodeInteriorness` returns the interiorness of a node.

cellInteriorness (*1 integer*)
`result = cellInteriorness` returns the interiorness of a cell.

dualCellInteriorness (*1 integer*)
`result = dualCellInteriorness` returns the interiorness of a dual-grid cell.

cellCenterInteriorness (*1 integer*)
`result = cellCenterInteriorness` returns the interiorness of the cell center (a point).

edgesInteriorness (*3 integers*)
`result = edgesInteriorness` returns the interiorness of all 3 edges.

dualEdgesInteriorness (*3 integers*)
`result = dualEdgesInteriorness` returns the interiorness of all 3 edges of the dual-grid cell.

facesInteriorness (*3 integers*)
`result = facesInteriorness` returns the interiorness of all 3 faces.

dualFacesInteriorness (*3 integers*)
`result = dualFacesInteriorness` returns the interiorness of all 3 faces of the dual-grid cell.

cellInsideCentroid (*NDIM floats*)
`result = cellInsideCentroid` eturns the centroid of the interior volume of the cell.

cellOutsideCentroid (*NDIM floats*)
`result = cellOutsideCentroid` returns the centroid of the exterior volume of the cell.

surfaceCentroid (*NDIM floats*)
`result = surfaceCentroid` returns the centroid of the boundary surface cutting through the cell, or the zero vector if the cell is not cut by the boundary.

surfaceCenter (*NDIM floats*)
`result = surfaceCenter` returns a point which is approximately the centroid of the boundary surface cutting through the cell, but is approximately on the surface even if the surface is curved; if the cell is not cut by the boundary, returns the zero vector.

cellVolFrac (*1 float*)
`result = cellVolFrac` returns the fraction of the cell volume that is interior to the grid-Boundary.

`facesAreDmNeglected` (*3 boolean*)
: `result = facesAreDmNeglected` returns whether the 3 faces (that touch the node) of the cell would be neglected by the Dey-Mittra Faraday update, given the gridBoundary's dmFrac.

`surfaceArea` (*1 float*)
: `result = surfaceArea` returns the area of the gridBoundary surface cutting through the cell.

`surfaceOutwardArea` (*NDIM floats*)
: `result = surfaceOutwardArea` returns a vector with direction in the surface-outward-normal and length equal to the surfaceArea within the cell.

`surfaceOutwardArea3D` (*3 floats*)
: `result = surfaceOutwardArea3D` returns a vector with direction in the surface-outward-normal and length equal to the surfaceArea within the cell.

`surfaceOutwardUnitNormal` (*NDIM floats*)
: `result = surfaceOutwardUnitNormal` returns a unit vector with direction in the surface-outward-normal.

`surfaceOutwardUnitNormal3D` (*3 floats*)
: `result = surfaceOutwardUnitNormal3D` returns a unit vector with direction in the surface-outward-normal.

`ndimZeroVector` (*NDIM booleans*)
: **`result = ndimZeroVector` returns NDIM zeros (this can** be convenient for figuring out the spatial dimension).

`edgeInteriorness` (*1 integer*)
: `result = edgeInteriorness` returns the interiorness of an edge.

  `dir` (*required; 0, 1, or 2*)
  : The direction of the desired edge.

`dualEdgeInteriorness` (*1 integer*)
: `result = dualEdgeInteriorness` returns the interiorness of a dual-grid edge.

  `dir` (*required; 0, 1, or 2*)
  : The direction of the desired edge.

`faceInteriorness` (*1 integer*)
: `result = faceInteriorness` returns the interiorness of a face.

  `dir` (*required; 0, 1, or 2*)
  : The direction normal to the desired face.

`dualFaceInteriorness` (*1 integer*)
: `result = dualFaceInteriorness` returns the interiorness of a dual-grid face.

  `dir` (*required; 0, 1, or 2*)
  : The direction normal to the desired face.

`faceInsideCentroid` (*NDIM floats*)
: `result = faceInsideCentroid` returns the centroid of the interior part of the face.

  `dir` (*required; 0, 1, or 2*)
  : The direction normal to the desired face.

`faceOutsideCentroid` (*NDIM floats*)
: `result = faceOutsideCentroid` returns the centroid of the exterior part of the face.

  `dir` (*required; 0, 1, or 2*)
  : The direction normal to the desired face.

`faceAreaFrac` (*1 float*)

`result = faceAreaFrac` returns the fraction of the cell face that is interior to the gridBoundary.

`dir` (*required; 0, 1, or 2*)
The direction normal to the desired face.

`edgeLengthFrac` (*1 float*)
`result = edgeLengthFrac` returns the fraction of the cell edge that is interior to the gridBoundary.

`dir` (*required; 0, 1, or 2*)
The direction of the desired edge.

`faceIsDmNeglected` (*1 boolean*)
`result = faceIsDmNeglected` returns whether a given face would be neglected by the Dey-Mittra Faraday update, given the gridBoundary's dmFrac.

`dir` (*required; 0, 1, or 2*)
The direction normal to the desired face.

`edgeBordersDmNeglectedFace` (*1 boolean*)
`result = edgeBordersDmNeglectedFace` returns whether the given cell edge borders a cell face that would be neglected by the Dey-Mittra Faraday update, given the gridBoundary's dmFrac.

historyFunc

A UserFunc of `kind = historyFunc` returns values from a `<History>`.

Currently only Tensor Histories can be probed by historyFunc; see *Tensor Histories*.

A UserFunc of `kind = historyFunc` is a function that takes a single argument of a single integer, and returns a (scalar) value from a `<History>` record. The argument is meant to be the difference between the current time step and a past time step. For example, a `kind = historyFunc` function will return its most recent record given an argument of 0; it will return its next-most-recent record given argument of 1, and so on.

historyFunc parameters

`history` (*string*, *required*)
The name of a <History> from which to retrieve data.

`index` (*vector of integers*, *required*)
The array-index of the desired element in the History. For example, if the History stores records that are 3×2 arrays (or tensors), then index might be `[0, 0]` or `[2,1]`. For a History that stores scalar records, this should simply be `[0]`. (N.B. the array dimensions of a history can be looked up in the History dump-file, keeping in mind that the first dimension is the record-index; for example, a scalar history stores a dataset of dimensions $n \times 1$.)

UserFunc Blocks

Local UserFuncs

Local UserFuncs are <UserFunc> blocks that appear within a UserFunc block of `kind = expression` (or within an <Expression> block), and can be called by the expression. Local functions are accessible only within the surrounding block.

Local functions can be used to simplify an expression; they can also be used to lookup values based on other simulation objects (such as fields).

The following UserFunc contains a local UserFunc named mySqr:

```
<UserFunc f>
  kind = expression
  inputOrder = [ y ]
  <Input y>
    kind = arbitraryVector
    types = [float]
  </Input>
  <UserFunc mySqr>
    kind = expression
    inputOrder = [x]
    <Input y>
      kind = arbitraryVector
      types = [float]
    </Input>
    expression = x*x
  </UserFunc>
  expression = mySqr(y)*mySqr(y+1) + y
</UserFunc>
```

The expression of `<UserFunc f>` specifies the argument when it calls mySqr. This example can be contrasted with that using Terms (see *Term Block*).

While local UserFuncs and Terms (*Term Block*) of `kind = expression` may helpfully simplify expressions, this may result in decreased performance. Where speed is important, it may be better to use pre-processor macros and pre-processor variables instead. For example, in the above we could probably benefit by replacing UserFunc mySqr with:

```
<macro mySqr(x)>
(x*x)
</macro>
```

Term Block

> A block that represents a local function whose value can be found and used as a term in a UserFunc expression (or in an <Expression>).

<Term>s are similar to *Local UserFuncs*; the difference is that <Term>s must take exactly the same arguments as the containing function, and they are treated like an <Input> in the containing function (although they shouldn't appear in the `inputOrder`, and they are not arguments of the function).

Differences between Terms and local functions:

- A <Term> must take the exact same arguments as the UserFunc in which it appears.
- A <Term> is evaluated exactly once for every evaluation of its containing UserFunc; if, e.g., it appears multiple times in an expression, the same value will be used each time (even if the Term returns a random number). In contrast, a local function will be called anew when needed.
  - Often this distinction is moot; it becomes important when evaluation is contingent. For example, suppose a `<Term f>` and a local `<UserFunc g>` appear in `expression = if(a, f, g())`. The expression `f` is needed only if `a` evaluates to true, and `g()` must be evaluated only if `a` is false. In this case `f` will be evaluated regardless, while `g()` is evaluated only if `a` is false. Therefore, using a local function instead of a variable can sometimes save time.

  - Another difference involves non-deterministic functions, e.g., random-number generators. If `f` is a Term, and `g()` a local function, each of which returns a random number, then in `expression = if(f < 0.5, f, 0.5) + if(g() < 0.5, g(), 0.5)` the same value will be used for both appearances of `f`, but each appearance of `g()` will evaluate to a different number.

Kinds of Terms

Terms take all the same kinds (and corresponding attributes) as UserFuncs (see *UserFunc Block*).

Examples

This example can be contrasted with that using local functions (see *Local UserFuncs*).

```
<UserFunc f>
  kind = expression
  inputOrder = [ y ]
  <Input y>
    kind = arbitraryVector
    types = [float]
  </Input>
  <Term mySqrVar>
    kind = expression
    inputOrder = [ y ] # same args as UserFunc f
    <Input y>
      kind = arbitraryVector
      types = [float]
    </Input>
    expression = y*y
  </Term>
  <Term myPlusOneSqrVar>
    kind = expression
    inputOrder = [ y ] # same args as UserFunc f
    <Input y>
      kind = arbitraryVector
      types = [float]
    </Input>
    expression = (y+1)*(y+1)
  </Term>
  expression = mySqrVar*myPlusOneSqrVar + y
</UserFunc>
```

In the above example, each Term is like a UserFunc that takes the same arguments as function `f` (namely, a scalar float named `y`). The Terms are then used in f's expression as variables.

All Terms are evaluated before f's expression is evaluated; this differs from local functions, which are typically evaluated only as needed.

Input Block

A block that describes the name, length, and types of a vector input argument to a UserFunc or Term.

Each (vector) input argument has a given length, and each element has a type (either boolean, integer, or float).

Input kinds

kind (*string*)
: Specifies the kind of Input block.

  arbitraryVector
  : A vector of arbitrary length, with arbitrary types.

  uniformVector
  : A vector argument of arbitrary length, but uniform type.

arbitraryVector Parameters

types (*vector of strings*, *required*)
: The type of each element of the input argument, in a list. For example, [integer float float boolean] specifies of vector of length 4, with the first element of type integer, the second of type float, etc.

uniformVector Parameters

The `kind = uniformVector` is especially useful for specifying NDIM-dimensional arguments (with the same dimension as the simulation), for simulations that can be run with arbitrary NDIM. Instead of constructing a list and truncating it to NDIM elements, one just adds `length = NDIM`.

type (*string*, *required*)
: The type of each element of the input argument; either float, integer, or boolean.

length (*optional non-negative integer*, *default=1*)
: The length of the vector.

More details

Speed and Multiple Evaluations of UserFuncs

Some UserFuncs can decrease computation time by vectorization, or simultaneous multiple evaluations. For example, consider the function $f(x, y) = x + y$, which we want to evaluate for several values of (x, y), namely $(0, 0)$, $(0, 1)$, $(2, 3)$, and $(-1, 2)$. We can often get results faster by performing all 4 evaluations at once, essentially trading 4 calls to the plus function with scalar arguments for 1 call to a plus function with 4-vector arguments. Since the addition of two numbers can usually be done faster than a function call, this is often a good trade-off, even though there are extra computational costs to allowing multiple (or vector) evaluation.

Unfortunately, vectorization is not always possible, and is not always efficient even when it is possible.

UserFuncs of kind expression allow multiple evaluations, limited by the maxNumEvals attribute. Expressions are pretty efficient evaluated one at a time; however, performing 64 evaluations at once typically reduces the computation time (per evaluation) by 30%–50%. There seems to be little benefit to going beyond 128 simultaneous evaluations.

The userFuncUpdater can efficiently use the ability to perform multiple evaluations at once. However, other Vorpal objects that use user-given functions cannot.

Changes in UserFunc syntax from 6.0 to 6.2

Vorpal 6.2 introduced changes in the experimental UserFunc framework, to bring UserFunc input more inline with other Vorpal input. The manual describes UserFuncs in detail, along with the current options.

These changes are not backwards compatible; input files that previously used UserFuncs and related objects (such as Tensor Histories) may have to be converted. This process involves several kinds of small changes.

Expression blocks are now used instead of **UserFunc** blocks in cases where the function signature is known/determined. **Expression**'s are otherwise very similar to **UserFunc**'s of `kind = expression`. The documentation for the surrounding block will state whether a **UserFunc** or an **Expression** is required.

For example, inside a FieldUpdater of `kind = userFuncUpdater`, the <UserFunc updateFunction> becomes an <Expression updateFunction> (with no kind attribute).

For example,

```
<UserFunc applySteps>
   kind = expression
   expression = (n == 1)
</UserFunc>
```

becomes

```
<Expression applySteps>
   expression = (n == 1)
</Expression>
```

and

```
<UserFunc updateFunction>
  kind = expression
  <SpaceFunc F>
    kind = fieldFunc
    result = fieldValue
    field = F
    component = 2
    interpolationOrder = 4
  </SpaceFunc>
  expression = F(posz)
</UserFunc>
```

becomes

```
<Expression updateFunction>
  <UserFunc F>
    kind = fieldFunc
    result = fieldValue
    field = F
    component = 2
    interpolationOrder = 4
  </UserFunc>
  expression = F(posz)
</Expression>
```

Blocks **SpaceFunc**, **HistoryFunc**, **CellFunc** should be changed to blocks of type **UserFunc** (with kind = spaceFunc, historyFunc, gridBoundaryFunc, etc.). See above example.

UserFunc blocks must specify the function signature; this is often done by specifying the kind—however, the signature must be explicitly given for `kind = expression`.

- `varOrder` is replaced by *`inputOrder`* (where `varOrder` was previously optional, *`inputOrder`* is now required)
- An **Input** block should be specified for each element of *`inputOrder`*.

- **Variable** blocks should be replaced by **Term** blocks.

For example,

```
<Variable i>
  kind = expression
  varOrder = [cellPos]
  cellPos = [integer integer integer]
  expression = select(cellPos,0)
</Variable>
```

becomes

```
<Term i>
  kind = expression
  inputOrder = [cellPos]
  <Input cellPos>
    kind = uniformVector
    type = integer
    length = 3
  </Input>
  expression = select(cellPos,0)
</Term>
```

Some specific cases where changes will be needed are:

- In FieldUpdaters of `kind = userfuncUpdater`, <Expression upadateFunction> replaces <UserFunc updateFunction>.
- **UserFunc**'s applySteps and applyTimes are replaced with **Expression**'s.

Other effects of these changes:

- In Tensor Histories, the funcLookupScope attribute is no longer valid.
- The `checkForUnaccessedAttribs` is no longer useful in most cases.

3.18 Slab Block

3.18.1 Slab

A Slab block defined the lower and upper bounds to a volume. Slab blocks are used in several different contexts in Vorpal.

PositionGenerator Blocks

PositionGenerator

```
<Slab loadSlab>
  lowerBounds = [0 0 0]
  upperBounds = [1 1 1]
</Slab>
```

```
<Slab emitSurface>
  lowerBounds = [0 0 0]
```

(continues on next page)

(continued from previous page)

```
  upperBounds = [0 1 1]
</Slab>
```

gridLoader ParticleSource Blocks

gridLoader

```
<Slab initLoadSlab>
  lowerBounds = [0 0 0]
  upperBounds = [1 1 1]
</Slab>
```

fieldAverage History Blocks

fieldAverage

```
<Slab region>
  lowerBounds = [0 0 0]
  upperBounds = [1 1 1]
</Slab>
```

3.19 Macros

3.19.1 Introduction to Macros

Note: In VSim 9, the transition to *direct macros* has been completed, so the previous macros are being deprecated. These include:

The macros are organized following a well defined-methodology. The philosophical change is that rather than having only macros that write out input file snippets directly, now there are macros that instead write the snippets into "accumulation variables". Then, at the end of the file, a call is made to `finalize()`, which writes out the accumulated information with the nesting needed for correct interpretation by Vorpal. A requirement for use of these version 8 macros is that the input (.pre) file must adhere to some structure, but the result is a much less constrained structure compared with the previous macros, which had to be placed in the pre-file in the location where the substitution was desired.

Here we provide an overview of these depcrecated macros. We will define *direct macros* to be those that directly write snippets. They can be used just as macros were used before. We define *accumulation macros* to be those that add snippets to accumulation variables. The latter are often effected by the former, as we shall see. The intention is to deprecate the old macros in version 9 and remove them in version 10.

Direct Macros Overview: Namespacing and input file structure

All global variables, e.g., the accumulation variables (to be discussed later), for the new macros start with `VPM_`. The standard macros are not namespaced, but may be namespaced in a future release. Also, there are variables that are namespaced starting with `TXU_`; for instance, variables that describe units.

With the new macros, the input file has to have a particular structure:

- Preamble: Setting of user-defined variables and variables that determine the type of simulation, e.g., electromagnetic versus electrostatic.
- Macro file importing.
- The user-defined regular and space-time functions.
- The ordered macros, i.e., grid and timing.
- The unordered macros (for all other dynamics).
- Finalization.

The Preamble

The preamble consists of the setting of the variables that one might want to use in the input file. For pre files generated from visual setup, the input file preamble begins with

```
#######################################################
#
# This PRE file,
#   emcartfull.pre,
# was generated from the simulation definition file,
#   emcartfull.sdf,
# by the sdf2vpre translator.
# Any changes to this file will be lost if the
# translator is rerun.
#
#######################################################

#######################################################
#
# Ordered blocks
#
#######################################################

#######################################################
# User defined constants and parameters
#######################################################

$ import mathphys

$ WAVELENGTH = 1.00000000000000e-01
$ WIDTH = 3*WAVELENGTH
$ DX = WAVELENGTH/16
$ QWID = 0.5*HALFWIDTH
$ EXPFAC = 0.5/(QWID^2)
$ XeMass = 131.3*PROTMASS
```

Of course, the comments (anything after the `#`) are not processed, nor are the blank lines. The user has freedom to choose the variable names provided they do not clash with the names in mathphys.mac, nor do they begin with `VPM_` or `TXU_`, which are reserved for distinguishing variables that are defined for system use. The macro system defines various boolean variables, so the user should not define variables with names like `True` or `FALSE`. Similarly, common math functions and Python system functions are reserved and should not be used as variable names. For this reason, users should not assign variable names like `cos`, `sqrt`, `open`, or `float`.

Next follows the basic variables that define the simulation:

```
#######################################################
# Simulation definition from the basic settings
#######################################################

$ VPM_COORDINATE_SYSTEM = "cartesian"
# cylindrical not found.
$ VPM_NDIM = "3"
$ VPM_PRECISION = "double"
$ VPM_USE_GPU = "False"
$ VPM_SIMULATION_TYPE = "electromagnetic"
# electrostatic not found.
$ VPM_GRID_SPACING = "uniform"
$ VPM_INCLUDE_PARTICLES = "False"
# include particles.estimated max electron density not found.
# include particles.estimated min electron temperature (eV) not found.
$ VPM_TOP_LEVEL_VERBOSITY = VPM_INFO
$ VPM_GRID_TYPE = UniformCartesian
```

and if `VPM_INCLUDE_PARTICLES` is `True`, one must also define

```
$ VPM_MAX_ELECTRON_DENSITY = "1.e18"
$ VPM_MAX_ELECTRON_TEMP_EV = "1.0"
```

These variables are used later to calculate the time step, if one is not provided, by later macros.

Macro Importing

To obtain that standard macros, the user need import only a single file, VSim,

```
$ import VSim
```

This in turn imports all of the standard macro files needed for the simulation of type defined in the preamble. This also initializes all of the accumulation variables, which will be further discussed below. If a user creates custom macro files, they should be imported after the above statement. For example,

```
$ import mymacrofile
```

User-defined regular and space-time functions.

The user defines regular and space-time functions in this part of the file in the regular way. Examples are shown below.

```
#######################################################
# User defined functions
#######################################################

<function SomeFunc(x,y)>
  cos(x)*sin(y)
</function>

#######################################################
# User defined space-time functions
#######################################################

$ Jz = exp(-EXPFAC*(y^2+z^2))*sin(OMEGA*t)
```

The Ordered Macro Calls

There are only two ordered macro calls. Each must be preceded by the definition of some global variables. The grid macro call comes first and has the form,

```
#######################################################
# Translation of grid
#######################################################

$ VPM_BGN0 = 0 * TXU_METERS
$ VPM_BGN1 = NEGHW * TXU_METERS
$ VPM_BGN2 = NEGHW * TXU_METERS

$ VPM_L0 = $LENGTH - 0$ * TXU_METERS
$ VPM_L1 = $HALFWIDTH - NEGHW$ * TXU_METERS
$ VPM_L2 = $HALFWIDTH - NEGHW$ * TXU_METERS

$ VPM_N0 = NX
$ VPM_N1 = NPERP
$ VPM_N2 = NPERP

$ VPM_PERIODIC_DIRS = []

setGridData(VPM_NDIM, VPM_GRID_TYPE)
```

with the following definitions

VPM_BGN0: The start of the grid in the first coordinate.

VPM_BGN1: The start of the grid in the second coordinate.

VPM_BGN2: The start of the grid in the third coordinate.

VPM_L0: The extent of the grid in the first coordinate.

VPM_L1: The extent of the grid in the second coordinate.

VPM_L2: The extent of the grid in the third coordinate.

VPM_N0: The number of cells in the first direction.

VPM_N1: The number of cells in the second direction.

VPM_N2: The number of cells in the third direction.

VPM_PERIODIC_DIRS: An array listing the coordinate directions in which one has periodic boundary conditions.

Following that is the timing setup, which has the form,

```
#######################################################
# Translation of the time group
#######################################################

$ VPM_DT = "0.0"
$ VPM_CFL_NUMBER = "0.95"
$ VPM_NSTEPS = 100
$ VPM_DUMP_PERIOD = 20

setTimingData()
```

VPM_DT: The time step for the simulation. If it is set to zero, the setTimingData macro will provide a best guess.

VPM_CFL_NUMBER: This is used when VPM_DT is set to zero. The time step is set to this factor times the maximum known stable time step.

VPM_NSTEPS: The number of time steps for the simulation.

VPM_DUMP_PERIOD: The number of time steps between each data dump.

The Unordered Macro Calls

With the new version-8 macros, one can now call the *accumulation* macros in any order. For example, one can add boundary conditions via

```
addBoundaryLauncher(boundaryLauncher0, ElectricField, 0.0, None, 0.0, upperY)
addMalBoundary(mal0, emField, lowerY, MALWID)
addOpenBoundary(open0, emField, upperX)
```

and one can add particle species, sinks and loaders with

```
addParticleSpecies(Xenon, 1.602176487e-19, XeMass, relBoris, variableWeights, 1.e18,␣
↪5.0, None)

addParticleSpeciesSink(absorbAndSave0, electrons0, absAndSav, lowerX)
addParticleSpeciesLoader(particleLoader0, xvLoaderEmitter, electrons0, 0.
↪00000000000000e+00, 0.00000000000000e+00, relativeDensity, 1.0, grid,␣
↪beamVelocityGen, [0.00000000000000e+00, 0.00000000000000e+00, 0.00000000000000e+00],
↪[5.00000000000000e+06, 5.00000000000000e+06, 5.00000000000000e+06], [-0.5, -0.5, -0.
↪5], [0.5, 0.5, 0.5])
addParticleSpecies(electrons0, ELECCHARGE, ELECMASS, relBoris, variableWeights, 1.e18,
↪ 5.0, None)
```

From the above one can see that the order is independent, as the species, `electrons0`, is added after its loaders and sinks. However, if a loader or sink is added for a species that ultimately is not added, no such species shows up in the simulation.

Finalization

Finalization consists of a single line,

```
$ finalize()
```

This macro takes all of the accumulated descriptions of the simulation, held in the accumulation variables, and writes it into the file with correct ordering and nesting.

Version 8 Macros Methodology

The basic units of the new macro system are the *direct macros*. An example of one from solvers.mac is

```
# Add the base solver block to an accumulation variable.
# @param name the accumulation variable
# @param krySize the Krylov subspace size (for gmres)
# @param orthogType the orthogonality type for the Krylov subspace (for gmres
<macro writeBaseSolverBlock(name, krySize, orthogType)>
  <BaseSolver myBS>
    kind = name
```

(continues on next page)

(continued from previous page)

```
    $ if isEqualString(name, generalized minimal residual solver)
      kspace = krySize
      orthog = orthogType
    $ endif
  </BaseSolver>
</macro>
```

Like all macros, this macro has basic documentation in the macros file, giving the macro name along with any parameters. In this case the macro directly writes an input file fragment, hence its name. Direct macros can be used just like any of the traditional macros, where the user controls the file structure, and the user knows where to place this macro call for the desired effect. Because all direct macros directly write, their names begin with `write`.

The above is actually an *independent direct macro* in that it can be used at any place in the file (though it may not make sense in any place). In particular, it need not be preceded with any other macros that set any accumulation variables, which we now turn to.

However, for deferred writing to allow the system to get the order of the macros correct, instead of writing directly, we accumulate these blocks in *accumulation variables* using *accumulation macros*, and then we write those out at finalization time. An example of an accumulation macro is

```
# Add the base solver block to the accumulation variable, VPM_MATRIX_SOLVER.
# Used for iterative solvers, such as gmres.
# @param name the name of the base solver block
# @param krySize the Krylov subspace size (for gmres)
# @param orthog the orthogonality type for the Krylov subspace (for gmres)
<macro addBaseSolver(name, krySize, orthog)>
  $ requires VPM_MATRIX_SOLVER
  $ VPM_MATRIX_SOLVER = appendToBlock(VPM_MATRIX_SOLVER, writeBaseSolverBlock(name,␣
↪krySize, orthog) )
  $ global VPM_MATRIX_SOLVER
</macro>
```

This macro counts on there being a global accumulation variable, **VPM_MATRIX_SOLVER**, which is defined in solvers.mac. It is important to initialize accumulation variables to an empty string prior to accumulating input file lines into them. Thus, the definition of an accumulation variable should be of the form,

```
$ VPM_MATRIX_SOLVER = ""
```

Accumulation into a variable appends to that variable the output of the previous macro, e.g. `writeBaseSolverBlock(...)`. With this device, we now have the base solver stored in the variable **VPM_MATRIX_SOLVER**, and we can write that variable out where it is needed.

For example, to write out a linear solver, one calls the dependent direct macro,

```
# Write the interative poisson solver block.  When used, it must come
# after addBaseSolver and addPreconditioner, as it writes the accumulation
# variables for the base solver and the preconditioner.
# @param maxIt the maximum number of iterations allowed.
# @param tol the tolerance for the solution
# @param met the metric for convergence, e.g., L1, L2
# Verbosity set to same as top level verbosity
<macro writePoissonSolverIterativeBlock(maxIt, tol, met)>
  $ requires VPM_TOP_LEVEL_VERBOSITY
  <LinearSolver linearSolver>
    kind = iterativeSolver
    verbosity = VPM_TOP_LEVEL_VERBOSITY
    maxIterations = maxIt
```

(continues on next page)

(continued from previous page)

```
    tolerance = tol
    metric = met
    VPM_MATRIX_SOLVER
    VPM_PRE_CONDITIONER
  </LinearSolver>
</macro>
```

This is called *dependent* because it relies on previous macro calls to have defined **VPM_MATRIX_SOLVER** and **VPM_PRE_CONDITIONER**. This writes (and so is a direct macro) a linear solver block, which can be used inside a multifield block. A user may use this macro just like any of the pre-version-8 macros, provided that the variables, **VPM_MATRIX_SOLVER** and **VPM_PRE_CONDITIONER** have been appropriately initialized.

In the application to multifields, one has at least the updaters, the update steps, and the update step order. The above macro would be used to write an update step. It is accumulated in the variable, **VPM_ES_SOLVER_UPDATERS**, which is finally written into the multifield block during the call to `finalize()`.

Version 8 Macro Files Organization

There are three sections in a macro file for (1) public macros, (2) global (including accumulation) variables, and (3) private macros. When working in the visual setup, the problem description is written out in an xml-like format that the translator translates to macro calls that are then processed to produce input file lines. Because most users only ever see these macro calls, they come first in the macro file. After those macro calls, the file defines the accumulation variables and then the other macros.

Public Macros

These are the macros currently written by the translator plus a few more, so they are sometimes called translator macros. They can be order dependent. For example, the macros calls for defining an iterative solver must occur as

```
addBaseSolver(bicgstab)
addPreconditioner(multigrid, SA, 30, GaussSeidel, 3, before, Jacobi, 1.
↪33300000000000e+00, 0.00000000000000e+00, increasing)
# This writes preconditioner, so it must be last.
addPoissonSolver(relPerm, 1000, 1.00000000000000e-08, r0)
```

Global (including accumulation) Variables

These are the variables into which one accumulates blocks for later printing in the correct order. All variables are namespaced with **VPMT_**, **VPM_** or **TXU_** so that the user can confidently use names that are not namespaced with **VPMT_**, **VPM_**, or **TXU_** provided they also avoid the names in mathphys.mac.

There is no guarantee that these variables will not change with future versions.

Private Macros

The translator macros make use of many other macros to accumulate blocks. These other macros take many forms: direct macros, utility macros, string manipulation, and so forth. These macros are not guaranteed to be stable between versions unless they appear in the documentation.

Top Level Macro Files

The top-level macro files are VSim.mac, VSimEm.mac, and VSimEs.mac. These control which other macro files are included, provide some universally used macros, and define the basic accumulation variables. In particular, if **VPM_SIMULATION_TYPE** equals Electromagnetic, then VSimEm.mac is loaded and it loads the macro files needed for electromagnetic simulations. If **VPM_SIMULATION_TYPE** equals Electrostatic, then VSimEs.mac is loaded and it loads the macro files needed for electrostatic simulations. Subsequently, VSim.mac loads the macro files for particles, fluids, collisions, etc., depending on what the preamble requested.

Remaining Macro Files

The remaining macro files are organized by functionality. For example, electrostatic boundary condition macros are in esbcs.mac. The particle macros are in particles.mac. When following the standard organization decribed above, the accumulation variables for any macros in a macro file are defined and initialized in either that file or in a previously loaded file.

3.19.2 Selective Processing

The preprocessor, txpp.py, processes macros by substitution, giving numerical values that can be interpreted by Vorpal. However, there are times when one wuld like to selective process, e.g., one might like expand all of the algorithms in order to substitute others or to access features that are not yet exposed in Visual Setup, or to combine algorithms in new and creative ways. Selective processing, slso known as partial preprocessing, allows one to do this. With selective processing one substitutes only a subset of the macros that txpp.py normally substitutes.

Default Selective Processing

Default selective processing is set to process as much as possible while still being able to have a final processing. As usual, one must set up paths by doing,

on Linux:

```
source /path/to/Contents/MacOS/VSimComposer.sh
```

on MacOS:

```
source /Applications/VSim-9.0/VSimComposer.app/Contents/MacOS/VSimComposer.sh
```

on Windows:

```
call \path\to\Contents\MacOS\VSimComposer.bat
```

Then proceed with the default selective processing by

```
python txppp.py -n simnameT.pre simname
```

(The initial python is needed on Windows only.)

This gives the default text-based input file, simnameT.pre, that one can further modify. A good practice is to copy simnameT.pre to simnameM.pre and modify the second one, so that if one changes simname.sdf, one can re-create simnameT.pre and merge in any differences to simnameM.pre.

Custom Selective Processing

The preprocessor, txpp.py, can also perform selective processing using the `--selective` flag to process a subset of the macros. At present, this has been tested on files that were generated from visual setup.

Running txpp.py with the selective option, e.g.

```
txpp.py --selective=fn input.pre -o input.ppp
```

expands the macros listed in the file named, fn. With

```
txpp.py --selective="" input.pre -o input.ppp
```

the file, expandmacros.txt, will be sought. It finds the file by following the usual import path, which includes the macros directory in which there is expandmacros.txt, containing, e.g.,

```
$ cat expandmacros.txt
# Below are the write macros that must be expanded
writeGrid
writeMultiField
writeEsFieldUpdaterBlock
writeParticleSpeciesBlock
writeImpactColliderBlock
writeParticleAbsorberEmitterHistoryBlock
# Below or the variables that must be expanded
VPMT_NDIM  # Expanded for clean limits
VPMT_INCLUDE_PARTICLES # Remove condition blocks depending on this
# Add more expansions here
```

To have more macros or variables expanded, copy this file next to your input file and edit it to add all the macros you want expanded, one per line.

By this method you should be able to arbitrarily morph an sdf-generated pre file to any .ppp file that is between the sdf-written pre file and the final .in file. One can then run vorpal using the ppp file as input, or one can further or fully process the ppp file.

3.19.3 Utility Macros

These macros will be mostly used by the framework, but the user may want to import and use them (especially like the mathphys macro), so they are docemented here.

All utility macros are available to all packages.

- *listUtilities.mac*
- *logicals.mac*
- *mathphys.mac*
- *timing.mac*
- *units.mac*
- *utilities.mac*
- *verbosity.mac*
- *vputilities.mac*

listUtilities.mac

This macro file can be imported to an input file with

```
$ import listUtilities
```

Macros available in listUtilities are used to append blocks and lists.

logicals.mac

This macro file can be imported to an input file with

```
$ import logicals
```

This macros file simply defines the logical variables: true, TRUE, false, and FALSE.

mathphys.mac

This macro file can be imported to an input file with

```
$ import mathphys
```

In many of the input file examples that are supplied in VSimComposer, you will see macros from `mathphys` invoked. Macros available in mathphys define a series of physical and mathematical constants that are commonly used in simulations. The constants defined in this macro file are:

```
# Math constants
$ PI = math.pi
$ PIO2 = 0.5*PI
$ TWOPI = 2.*PI

# Physical constants from http://physics.nist.gov/cuu/Constants/index.html
$ LIGHTSPEED = 2.99792458e8     # m/s
$ C2 = LIGHTSPEED*LIGHTSPEED
$ MU0 = 4e-7*PI
$ EPSILON0 = 1/(MU0 * C2)
$ ELECMASS = 9.10938215e-31     # kG
$ PROTMASS = 1.672621637e-27    # kG
$ MUONMASS = 1.88353130e-28     # kg
$ ELEMCHARGE = 1.602176487e-19  # C
$ KB = 1.3806504e-23 # J/K

# gyromagnetic anomaly of the proton (CODATA 2014)
$ PROTON_GMA = 1.7928473508        # (85), dimensionless
# gyromagnetic anomaly of the electron (CODATA 2014)
$ ELECTRON_GMA = 1.15965218091e-3  # (26), dimensionless

# Secondary parameters
$ ELECCHARGE = -ELEMCHARGE      # C
$ ELECMASSEV = ELECMASS*C2/ELEMCHARGE   # eV
```

timing.mac

This macro file can be imported to an input file with

```
$ import timing
```

It is imported by VSim.mac. This macro file defines time-related parameters and makes them global variables.

Public Macros

setTimingData()

In the typical use case, `VPM_DT`, `VPM_NSTEPS`, `VPM_DUMP_PERIOD`, `VPM_DMFRAC`, and `VPM_CFL_NUMBER` are set before entry. This macro then computes `VPM_SIMULATION_TIME`, and `VPM_NUM_DUMPS`. `VPM_DT` is respected if not zero. Otherwise it is computed from the `VPM_CFL_NUMBER` and the minimum CFL time steps from electromagnetics (if an electromagnetic simulation) and (if there are particles present) the plasma frequency and the thermal electron cell crossing time computed from `VPM_MAX_ELECTRON_DENSITY` and `VPM_MAX_ELECTRON_TEMP_EV`.

units.mac

This macro file can be imported to an input file with

```
$ import units
```

This macro file contains conversions factors for units to SI units.

utilities.mac

This macro file can be imported to an input file with

```
$ import utilities
```

This macro file contains various utilities for testing equality, converting coords, and the like.

Public Macros

isEqualString(*aString*, *bString*)

The isEqualString macro tests if two strings are equal. It returns True or False.

Parameters

- **aString** – The first of two strings to compare.
- **bString** – The second of two strings to compare.

isNotEqualInt(*i*, *j*)

The isNotEqualInt macro tests if two integers are not equal. It returns True or False.

Parameters

- **i** – The first of two integers to compare.
- **j** – The second of two integers to compare.

entryToBool(*pd*)

The entryToBool macro converts an array of 0/1’s to booleans True/False.

Parameters **pd** – An array of integers representing booleans, e.g. periodicity.

verbosity.mac

This macro file can be imported to an input file with

```
$ import verbosity
```

This macro file contains the constants used in setting the level of output in a simulation.

After importing, it can be used by setting the verbosity to the desired level, e.g. `verbosity = VPM_NOTICE`.

The verbosity levels are as follows:

```
$ VPM_EMERGENCY
$ VPM_ALERT
$ VPM_CRITICAL
$ VPM_ERROR
$ VPM_WARNING
$ VPM_NOTICE
$ VPM_INFO
$ VPM_DEBUG
$ VPM_DEBUG2
$ VPM_DEBUG3
```

vputilities.mac

This macro file can be imported to an input file with

```
$ import vputilities
```

It is imported by VSim.mac. This macro file defines vsim-specific utility macros.

Public Macros

`writeIndicesFromCoords` (*coordBounds*)

Convert coordinates to the corresponding indices.

Parameters **`coordBounds`** – The coordinates to be converted. A float array of length VPM_NDIM.

`write1CellVolume` (*p*, *outwardNormal*, *dl0*, *dl1*, *dl2*, *l0*, *l1*, *l2*, *st0*, *st1*, *st2*)

Create a 1-cell slab from in (x,y,z) tuple and a direction outwardNormal. It is a tuple that points in the direction of the outward normal to the surface.

Parameters

- **`p`** – A value that gives the position to place the slab.
- **`outwardNormal`** – Array of integers specifying the outward normals.
- **`dl0`** – Cell length in the zeroth direction, e.g. VPM_DX.
- **`dl1`** – Cell length in the first direction, e.g. VPM_DY.
- **`dl2`** – Cell length in the second direction, e.g. VPM_DZ.
- **`l0`** – Length of the domain in the zeroth direction, e.g. VPM_LX.
- **`l1`** – Length of the domain in the first direction, e.g. VPM_LY.
- **`l2`** – Length of the domain in the second direction, e.g. VPM_LZ.

- **`st0`** – Starting point in the zeroth direction, e.g. VPM_BGNX.
- **`st1`** – Starting point in the first direction, e.g. VPM_BGNY.
- **`st2`** – Starting point in the second direction, e.g. VPM_BGNZ.

`getBoundaryOffsets` (*location*)
Apply offsets to our bounds if a boundary condition has already been set to an orthogonal face.

Parameters **`location`** – The boundary location, one of lower/upper[X,Y,Z].

3.19.4 Directed Macros

beamemitters.mac

This macro file can be imported to an input file with

```
$ import beamemitters
```

It is imported by VSim whenever a simulation contains particles.

This macro file is available to all packages.

This macro file defines macros for adding beam emitters for particles to be emitted from.

Public Macros

`addBeamEmitter` (*sourceName*, *owningSpecies*, *start*, *stop*, *vbar*, *vsig*, *fluxSetting*, *fluxOrCurrent*, *location*)
Add a particle emitter source block to the simulation. Velocity is given by the mean and standard deviation of the velocity distribution of the emitted particles. Current density is given by explicitly being set or by setting the desired flux whereby a calculation is done later. Location of emission can be off a GridBoundary shape or a boundary of the simulation domain. Delegates to addSettableFluxEmitter(15 args).

Parameters

- **`sourceName`** – The name of the particle source block.
- **`owningSpecies`** – The name of the emitted particle the source belongs to. Can be an electron, an ion, or a neutral particle.
- **`start`** – The simulation time at which emission begins.
- **`stop`** – The simulation time at which emission ends.
- **`vbar`** – A three-vector of mean velocities.
- **`vsig`** – A three-vector of rms velocities.
- **`fluxSetting`** – A string used to determine whether to explicitly set the current density or to calculate it.
- **`fluxOrCurrent`** – The function or numerical value used to set the current density or to calculate it.
- **`location`** – The boundary plane of the simulation domain to emit off.

`addBeamEmitter` (*sourceName*, *owningSpecies*, *start*, *stop*, *vbar*, *vsig*, *fluxSetting*, *fluxOrCurrent*, *emOff*, *maskFunction*, *object*)
Add a particle emitter source block to the simulation. Velocity is given by the mean and standard deviation of the velocity distribution of the emitted particles. Current density is given by explicitly being set or by setting

the desired flux whereby a calculation is done later. Location of emission can be off a GridBoundary shape or a boundary of the simulation domain. Delegates to addSettableFluxEmitter(15 args).

Parameters

- **`sourceName`** – The name of the particle source block.
- **`owningSpecies`** – The name of the emitted particle the source belongs to. Can be an electron, an ion, or a neutral particle.
- **`start`** – The simulation time at which emission begins.
- **`stop`** – The simulation time at which emission ends.
- **`vbar`** – A three-vector of mean velocities.
- **`vsig`** – A three-vector of rms velocities.
- **`fluxSetting`** – A string used to determine whether to explicitly set the current density or to calculate it.
- **`fluxOrCurrent`** – The function or numerical value used to set the current density or to calculate it.
- **`emOff`** – The distance away from the emitting object to emit.
- **`maskFunction`** – A function used to tailor the shape of emission off of the emitting object.
- **`object`** – The shape object, or GridBoundary as it is called in vorpal, off of which to emit.

`addBeamEmitter`(*sourceName*, *owningSpecies*, *start*, *stop*, *vbar*, *vsig*, *fluxSetting*, *fluxOrCurrent*, *coordinate*, *minValue*, *maxValue*, *location*)

Add a particle emitter source block to the simulation. Velocity is given by the mean and standard deviation of the velocity distribution of the emitted particles. Current density is given by explicitly being set or by setting the desired flux whereby a calculation is done later. Location of emission can be off a GridBoundary shape or a boundary of the simulation domain. Delegates to addSettableFluxEmitter(15 args).

Parameters

- **`sourceName`** – The name of the particle source block.
- **`owningSpecies`** – The name of the emitted particle the source belongs to. Can be an electron, an ion, or a neutral particle.
- **`start`** – The simulation time at which emission begins.
- **`stop`** – The simulation time at which emission ends.
- **`vbar`** – A three-vector of mean velocities.
- **`vsig`** – A three-vector of rms velocities.
- **`fluxSetting`** – A string used to determine whether to explicitly set the current density or to calculate it.
- **`fluxOrCurrent`** – The function or numerical value used to set the current density or to calculate it.
- **`coordinate`** – The coordinate of the plane of emission. For a Z-Plane, an R-coordinate.
- **`minValue`** – A coordinate defining the minimum spatial value for emission. For a Z-Plane a Z-coordinate.
- **`maxValue`** – A coordinate defining the maximum spatial value for emission. For a Z-Plane a Z-coordinate.
- **`location`** – The boundary plane of the simulation domain to emit off.

collisions.mac

This macro file can be imported to an input file with

```
$ import collisions
```

It is imported by VSim whenever a simulation contains particles.

This macro file is available to all packages.

This macro file defines macros for adding collisions between particles and particles and between particles and fluids into a simulation.

Uses: collisionsinz.mac, collisionsmc.mac, and collisionsrxn.mac

To add collisions to your simulation with the collisions.mac macros involves 3 defining steps.

- Create the neutral gas using the fluids.mac macro *addNeutralGas*
- Specify the type of collisions (electrons or ions) by including a macro for either *addElectronNeutralFluidCollisions* or *addIonNeutralFluidCollisions*
- Specify which types of collisions for each species where choices are
  - Electrons: Elastic, Excitation, Ionization
  - Ions: Charge Exchange, Momentum Exchange

Examples

Here are some example uses of the macros in this macro file:

```
addNeutralGas(XeNeutralGas, Xe, initGasDen, 300., True, [zMin, rMin], [zMax, rMax])

addElectronNeutralFluidCollisions(elecNeutralColl, electrons, XeNeutralGas)

addElectronNeutralFluidIonizationCollision(elecIonization, elecNeutralColl, electrons,
↪ XePlus, isotropic, EvaluatedElectronDataLibrary, eedl.dat)

addElectronNeutralFluidElasticCollision(elecElastic, elecNeutralColl, isotropic,␣
↪EvaluatedElectronDataLibrary, eedl.dat)

addElectronNeutralFluidExcitationCollision(elecExcitation, elecNeutralColl, isotropic,
↪ EvaluatedElectronDataLibrary, eedl.dat)
```

Public Macros

`addElectronNeutralFluidCollisions`(*name*, *impactElectrons*, *neutralGas*)

Add collisions between an electron species and a neutral background gas.

Parameters

- **`name`** – The name of this collision process.
- **`impactElectrons`** – The name of the impacting electron species.
- **`neutralGas`** – The name of the neutral background gas.

`addIonNeutralFluidCollisions`(*name*, *impactIons*, *neutralGas*)

Add collisions between an ion species and a neutral background gas.

Parameters

- **`name`** – The name of this collision process.
- **`impactElectrons`** – The name of the impacting electron species.
- **`neutralGas`** – The name of the neutral background gas.

`addElectronNeutralFluidElasticCollision`(*name*, *owningCollider*, *productDist*, *crossSectionType*)

Add elastic collision with built in cross-sectioning.

Parameters

- **`name`** – The name of this collision process.
- **`owningCollider`** – The collisions parent that defines the impact species and the neutral gas. The *name* used for the *addElectronNeutralFluidCollisions* macro.
- **`productDist`** – The distribution of product velocities after collision. Choice of *isotropic* or *VahediSurendra*.
- **`crossSectionType`** – The cross section calculation method. Choice of *userDefinedCrossSection* or *EvaluatedElectronDataLibrary*.

`addElectronNeutralFluidElasticCollision`(*name*, *owningCollider*, *productDist*, *crossSectionType*, *dataFile*)

Add elastic collision with file defined in cross-sectioning.

Parameters

- **`name`** – The name of this collision process.
- **`owningCollider`** – The collisions parent that defines the impact species and the neutral gas. The *name* used for the *addElectronNeutralFluidCollisions* macro.
- **`productDist`** – The distribution of product velocities after collision. Choice of *isotropic* or *VahediSurendra*.
- **`crossSectionType`** – The cross section calculation method. Choice of *userDefinedCrossSection* or *EvaluatedElectronDataLibrary*.
- **`dataFile`** – The file containing the cross section data.

`addElectronNeutralFluidExcitationCollision`(*name*, *owningCollider*, *productDist*, *crossSectionType*)

Add excitation collision with built in cross-sectioning.

Parameters

- **`name`** – The name of this collision process.
- **`owningCollider`** – The collisions parent that defines the impact species and the neutral gas. The *name* used for the *addElectronNeutralFluidCollisions* macro.
- **`productDist`** – The distribution of product velocities after collision. Choice of *isotropic* or *VahediSurendra*.
- **`crossSectionType`** – The cross section calculation method. Choice of *userDefinedCrossSection* or *EvaluatedElectronDataLibrary*.

`addElectronNeutralFluidExcitationCollision`(*name*, *owningCollider*, *productDist*, *crossSectionType*, *dataFile*)

Add excitation collision with file defined in cross-sectioning.

Parameters

- **name** – The name of this collision process.
- **owningCollider** – The collisions parent that defines the impact species and the neutral gas. The *name* used for the *addElectronNeutralFluidCollisions* macro.
- **productDist** – The distribution of product velocities after collision. Choice of *isotropic* or *VahediSurendra*.
- **crossSectionType** – The cross section calculation method. Choice of *userDefinedCrossSection* or *EvaluatedElectronDataLibrary*.
- **dataFile** – The file containing the cross section data.

addElectronNeutralFluidIonizationCollision(*name*, *owningCollider*, *productElectrons*, *productIons*, *productDist*, *crossSectionType*)

Add ionization collision with built in cross sectioning.

Parameters

- **name** – The name of this collision process.
- **owningCollider** – The collisions parent that defines the impact species and the neutral gas. The *name* used for the *addElectronNeutralFluidCollisions* macro.
- **productElectrons** – The name of the product electron species.
- **productIons** – The name of the product ion species.
- **productDist** – The distribution of product velocities after collision. Choice of *isotropic* or *VahediSurendra*.
- **crossSectionType** – The cross section calculation method. Choice of *builtIn*, *userDefinedCrossSection* or *EvaluatedElectronDataLibrary*.

addElectronNeutralFluidIonizationCollision(*name*, *owningCollider*, *productElectrons*, *productIons*, *productDist*, *crossSectionType*, *dataFile*)

Add ionization collision with file defined in cross-sectioning.

Parameters

- **name** – The name of this collision process.
- **owningCollider** – The collisions parent that defines the impact species and the neutral gas. The *name* used for the *addElectronNeutralFluidCollisions* macro.
- **productElectrons** – The name of the product electron species.
- **productIons** – The name of the product ion species.
- **productDist** – The distribution of product velocities after collision. Choice of *isotropic* or *VahediSurendra*.
- **crossSectionType** – The cross section calculation method. Choice of *builtIn*, *userDefinedCrossSection* or *EvaluatedElectronDataLibrary*.
- **dataFile** – The file containing the cross section data.

addIonNeutralFluidChargeExchangeCollision(*name*, *owningCollider*, *productDist*, *crossSectionType*)

Add charge exchange collision with built in cross-sectioning.

Parameters

- **name** – The name of this collision process.

- **owningCollider** – The collisions parent that defines the impact species and the neutral gas. The *name* used for the *addIonNeutralFluidCollisions* macro.
- **productDist** – The distribution of product velocities after collision. Choice of *backward*.
- **crossSectionType** – The cross section calculation method. Choice of *builtIn* or *userDefinedCrossSection*.

addIonNeutralFluidChargeExchangeCollision(*name*, *owningCollider*, *productDist*, *crossSectionType*, *dataFile*)

Add charge exchange collision with file defined in cross-sectioning.

Parameters

- **name** – The name of this collision process.
- **owningCollider** – The collisions parent that defines the impact species and the neutral gas. The *name* used for the *addIonNeutralFluidCollisions* macro.
- **productDist** – The distribution of product velocities after collision. Choice of *backward*.
- **crossSectionType** – The cross section calculation method. Choice of *builtIn* or *userDefinedCrossSection*.
- **dataFile** – The file containing the cross section data.

addIonNeutralFluidMomentumExchangeCollision(*name*, *owningCollider*, *productDist*, *crossSectionType*)

Add momentum exchange collision with built in cross-sectioning.

Parameters

- **name** – The name of this collision process.
- **owningCollider** – The collisions parent that defines the impact species and the neutral gas. The *name* used for the *addIonNeutralFluidCollisions* macro.
- **productDist** – The distribution of product velocities after collision. Choice of *backward*
- **crossSectionType** – The cross section calculation method. Choice of *builtIn* or *userDefinedCrossSection*

addIonNeutralFluidMomentumExchangeCollision(*name*, *owningCollider*, *productDist*, *crossSectionType*, *dataFile*)

Add momentum exchange collision with file defined in cross sectioning

Parameters

- **name** – The name of this collision process
- **owningCollider** – The collisions parent that defines the impact species and the neutral gas. The *name* used for the *addIonNeutralFluidCollisions* macro.
- **productDist** – The distribution of product velocities after collision. Choice of *backward*.
- **crossSectionType** – The cross section calculation method. Choice of *builtIn* or *userDefinedCrossSection*.
- **dataFile** – The file containing the cross section data.

embcs.mac

This macro file can be imported to an input file with

```
$ import embcs
```

It is imported by VSimEM in all cases.

This macro file is available to all packages.

This macro file defines numerous different electromagnetic boundary conditions such as Ports, PortLauncers, PMLs and MALs, magnetic symmetries and electric symmetries.

Public Macros

addElectricBoundary (*name*, *fieldName*, *location*)
Add an electric (PEC) boundary condition.

Parameters

- **name** – The name of the updater.
- **fieldName** – Unused.
- **location** – The location of the boundary condition, [lower/upper][X/Y/Z].

addMagneticBoundary (*boundaryConditionName*, *fieldName*, *location*)
Add a magnetic boundary condition.

Parameters

- **boundaryConditionName** – The name of the updater.
- **fieldName** – Unused.
- **location** – The location of the boundary condition, [lower/upper][X/Y/Z].

addOpenBoundary (*boundaryConditionName*, *fieldName*, *location*)
Add an open boundary.

Parameters

- **boundaryConditionName** – The name of the updater.
- **fieldName** – Unused.
- **location** – The location of the boundary condition, [lower/upper][X/Y/Z].

addMalBoundary (*boundaryConditionName*, *fieldName*, *location*, *malThickness*)
Add a Matched Absorbing Layer (MAL), a layer with increasing resistance out to the simulation boundary. Sets default pwr = 3.0, frac = 0.5, and malThickness = 1.0.

Parameters

- **boundaryConditionName** – The name of the updater.
- **fieldName** – Name of the field. Determines updater and step names.
- **location** – The location of the boundary condition, [lower/upper][X/Y/Z].
- **malThickness** – Real valued thickness of MAL into sim domain, same units as the grid. There are macro guards on the tickness that restrict the MAL thickness to be no greater than 50% of the simulation domain and no thinner than 2 grid cells regardless of whether the user specify quantities greater/lesser than these values.

addBoundaryLauncher (*boundaryConditionName*, *fieldName*, *xProfile*, *yProfile*, *zProfile*, *location*)
Add boundary launcher, formats the bounds.

Parameters

- **boundaryConditionName** – The name of the updater.
- **fieldName** – Name of the field. Determines updater and step names.
- **xProfile** – The x component of the field.
- **yProfile** – The y component of the field.
- **zProfile** – The z component of the field.
- **location** – The location of the boundary condition, [lower/upper][X/Y/Z].

emfilters.mac

This macro file can be imported to an input file with

```
$ import emfilters
```

It is imported at the top of all simulation runs.

This macro file is available to all packages.

This macro file defines the macro for applying an EM noise filter. Currently, the filter is a numerical Cerenkov filter.

Public Macros

applyNumericalCerenkovFilter(*stages*)
Applies a Cherenkov filter of a given stage type.

Parameters **stages** – The type of filter to use. Select from: none, weak, medium, strong, and extreme.

esmatrix.mac

This macro file can be imported to an input file with

```
$ import esmatrix
```

It is imported by VSimEs.mac.

This macro file is available to all packages.

This macro file defines macros that set up the matrix for solving electrostatics. Includes setting up the bulk matrix and the boundary conditions. Use by first adding all the needed boundary conditions. Then write the matrix, which will insert those boundary conditions.

Default Accumulation Macros

addEsDirichletBC(*name*, *value*, *location*)
Add a Dirichlet boundary condition using a simulation boundary location.

Parameters

- **name** – The name of the boundary condition block.
- **value** – The voltage for this boundary condition.

- **location** – The boundary condition location, one of lower/upper[X,Y,Z], e.g. for Cartesian coordinates.

addEsDirichletBC(*name*, *value*, *lb*, *ub*)
Add a Dirichlet boundary condition using an index slab.

Parameters

- **name** – The name of the boundary condition block.
- **value** – The voltage for this boundary condition.
- **lb** – The lower bound indexes of the slab over which this boundary condition is applied.
- **ub** – The upper bound indexes of the slab over which this boundary condition is applied.

addEsNeumannBC(*name*, *value*, *location*)
Add a Neumann boundary condition using a simulation boundary location.

Parameters

- **name** – The name of the boundary condition block.
- **value** – The value of the potential difference between two nodes.
- **location** – The boundary condition location, one of lower/upper[X,Y,Z], e.g. for Cartesian coordinates.

addEsNeumannBC(*name*, *value*, *direction*, *multiplier*, *lb*, *ub*)
Add a Neumann boundary condition using an index slab.

Parameters

- **name** – The name of the boundary condition block.
- **value** – The value of the potential difference between two nodes. (See multiplier).
- **direction** – The direction of the potential difference. One of [0,1,2], corresponding to [X,Y,Z] in Cartesian coordinate systems, e.g.
- **multiplier** – The multiplier for the potential difference. If this is positive, one takes the value one cell up minus the value at the node. If this is negative or zero, one takes the value at the node minus the value one cell down.
- **lb** – The lower bound indexes of the slab over which this boundary condition is applied.
- **ub** – The upper bound indexes of the slab over which this boundary condition is applied.

fluids.mac

This macro file can be imported to an input file with

```
$ import fluids
```

It is imported by VSim whenever a simulation contains particles.

This macro file is available to all packages.

This macro file defines macros for adding fluids that particles can collide off of.

Public Macros

`addNeutralGas` (*name*, *gasKind*, *gasDensity*, *gasTemp*, *isvar*, *lc*, *uc*)

Add a fluid block to the simulation. Adds to the fluid blocks accumulation variables and adds to accumulation variables for parameters like excitation and ionization thresholds for use in other blocks

Parameters

- **`name`** – The name of the fluid block.
- **`gasKind`** – The kind of neutral gas. Typically the symbol for elemental fluids, e.g. Xe for xenon.
- **`gasDensity`** – The function or numerical value for the initial density.
- **`gasTemp`** – The neutral gas temperature.
- **`isvar`** – Whether the initial density is a spatially dependent expression, f(x,y,z), or a constant value (amplitude).
- **`lc`** – The lower coordinates of the fluid location. Real array of length VPM_NDIM.
- **`uc`** – The upper coordinates of the fluid location. Real array of length VPM_NDIM.

grids.mac

This macro file can be imported to an input file with

```
$ import grids
```

It is imported at the top of all simulation runs.

This macro file is available to all packages.

This macro file defines the variable for the grids. The grid is written at time of finalize.

Public Macros

`setGridData` (*nDim*, *coordSystem*)

Set the grid variables,

VPM_BGN0: The start of the grid in the first coordinate,

VPM_BGN1: The start of the grid in the second coordinate,

VPM_BGN2: The start of the grid in the third coordinate,

VPM_L0: The extent of the grid in the first coordinate,

VPM_L1: The extent of the grid in the second coordinate,

VPM_L2: The extent of the grid in the third coordinate,

VPM_N0: The number of cells in the first direction,

VPM_N1: The number of cells in the second direction,

VPM_N2: The number of cells in the third direction,

VPM_PERIODIC_DIRS: An array listing the coordinate directions in which one has periodic boundary conditions,

then calls setGridData(4 args) to set the coordinate-system dependent variables.

Parameters

- **`nDim`** – The dimensionality of the simulation.
- **`coordSystem`** – The coordinate system. One of "UniformCartesian" or "Cylindrical".

`setGridData` (*nDim*, *coordSystem*, *numCells*, *lengths*, *startPositions*)

Sets the coordinate-system dependent grid variables, VPM_NDIM and VPM_GRID_TYPE, and sets the periodic directions.

For UniformCartesian, computes and sets: VPM_NX, VPM_NY, VPM_NZ, VPM_LX, VPM_LY, VPM_LZ, VPM_BGNX, VPM_BGNY, VPM_BGNZ, VPM_ENDX, VPM_ENDY, VPM_ENDZ, VPM_DX, VPM_DY, VPM_DZ, VPM_NXP1, VPM_NYP1, VPM_NZP1, VPM_NXM1, VPM_NYM1, VPM_NZM1, VPM_DXI, VPM_DYI, VPM_DZI, VPM_DL, VPM_DLI, depending on the dimensionality.

For Cylindrical, computes and sets: VPM_NZ, VPM_NR, VPM_NPHI, VPM_LZ, VPM_LR, VPM_LPHI, VPM_BGNZ, VPM_BGNR, VPM_BGNPHI, VPM_ENDZ, VPM_ENDR, VPM_ENDPHI, VPM_DZ, VPM_DR, VPM_DPHI, VPM_NZP1, VPM_NRP1, VPM_NPHIP1, VPM_NZM1, VPM_NRM1, VPM_NPHIM1, VPM_DZI, VPM_DRI, VPM_DPHII, VPM_DL, VPM_DLI, depending on the dimensionality.

Parameters

- **`nDim`** – The dimensionality of the simulation.
- **`coordSystem`** – The coordinate system. One of "UniformCartesian" or "Cylindrical".
- **`numCells`** – An integer array of length nDim indicating the number of cells in each direction.
- **`lengths`** – A real-valued array of length nDim indicating the lengths of the simulation domain in each direction.
- **`startPositions`** – A real-valued array of length nDim indicating the start position of each direction. I.e. the coordinate of the node with indices [0,0,0].

histories.mac

This macro file can be imported to an input file with

```
$ import histories
```

It is imported by VSim.mac.

This macro file is available to all packages.

This macro file provides macros for all vorpal histories supported in this release. Histories pertain primarily to fields and particles.

Public Macros

`addParticleMomentumHistory` (*name*, *speciesName*)

Add particle history block that records the momentum of a particle species.

Parameters

- **`name`** – The name of the history block.
- **`speciesName`** – The name of the species to be recorded.

addParticleAbsorberLog (*name*, *absorberName*, *ptclQty*, *component*)
Add particle history block that records a specific quantity about particles absorbed by the given absorber.

Parameters

- **name** – The name of the history block.
- **absorberName** – The name of the particle absorber at which to record.
- **ptclQty** – The quantity to be recorded. Particle quantities include time, position, velocity, weight, energy, and current.
- **component** – The component of the particle quantity to record. Only position, velocity, and weight require a component.

addParticleAbsorberCurrentHistory (*name*, *absorberName*)
Add particle history block that records the current located at a particle absorber.

Parameters

- **name** – The name of the history block.
- **absorberName** – The name of the particle absorber at which to record.

addParticleAbsorberEnergyHistory (*name*, *absorberName*)
Add particle history block that records the energy of particles located at a particle absorber.

Parameters

- **name** – The name of the history block.
- **absorberName** – The name of the particle absorber at which to record.

addParticleEmitterCurrentHistory (*name*, *emitterName*)
Add particle history block that records the current located at a particle emitter.

Parameters

- **name** – The name of the history block.
- **emitterName** – The name of the particle emitter at which to record.

addMacroParticleCountHistory (*name*, *speciesName*)
Add particle history block that records the number of macro particles in a simulation.

:param name The name of the history block.

:param speciesName The name of the species to be recorded.

addPhysicalParticleCountHistory (*name*, *speciesName*)
Add particle history block that records the number of physical particles in a simulation.

:param name The name of the history block.

:param speciesName The name of the species to be recorded.

addParticleEnergyHistory (*name*, *speciesName*)
Add particle history block that records energy of a particle species.

Parameters

- **name** – The name of the history block.
- **speciesName** – The name of the species to be recorded.

addEMFieldEnergyHistory (*name*, *region*)
Add field history block that records energy of the EM field. Can specify the entire simulation region or the region inside a user defined shape.

Parameters

- **`name`** – The name of the history block.
- **`region`** – A string representing the region in which to record. Can either be the entire simulation domain or a regiondefined by an imported geometry or created through CSG.

`addEMFieldEnergyHistory`(*name*, *lb*, *ub*)
Add field history block that records energy of the EM field. Can specify any region in the simulation domain.

Parameters

- **`name`** – The name of the history block.
- **`lb`** – An integer three-vector. The lower bounds on which to record in grid indices.
- **`ub`** – An integer three-vector. The upper bounds on which to record in grid indices.

`addFieldEnergyAtPositionHistory`(*name*, *location*, *field*)
Add field history block that records energy of the specified field. Records field energy at specific point in the simulation domain.

Parameters

- **`name`** – The name of the history block.
- **`location`** – A real valued three-vector. The point at which to record.
- **`field`** – The field which to record.

`addElectricFieldEnergyHistory`(*name*, *simRgn*)
Add field history block that records energy of the electric field inside the whole simulation domain.

Parameters

- **`name`** – The name of the history block.
- **`simRgn`** – A string representing the entire simulation region.

`addElectricFieldEnergyHistory`(*name*, *lb*, *ub*)
Add field history block that records energy of the electric field. Can specify any region in the simulation domain.

Parameters

- **`name`** – The name of the history block.
- **`lb`** – An integer three-vector. The lower bounds on which to record in grid indices.
- **`ub`** – An integer three-vector. The upper bounds on which to record in grid indices.

`addMagneticFieldEnergyHistory`(*name*, *simRgn*)
Add field history block that records energy of the magnetic field inside the whole simulation domain.

Parameters

- **`name`** – The name of the history block.
- **`simRgn`** – A string representing the entire simulation region.

`addMagneticFieldEnergyHistory`(*name*, *lb*, *ub*)
Add field history block that records energy of the magnetic field. Can specify any region in the simulation domain.

Parameters

- **`name`** – The name of the history block.
- **`lb`** – An integer three-vector. The lower bounds on which to record, in grid indices.

- **ub** – An integer three-vector. The upper bounds on which to record, in grid indices.

addPseudoPotentialHistory (*name*, *refPoint*, *measPoint*)
Add history block that calculates the pseudo-potential difference, in volts, between the reference point and the measure point. See the pseudoPotential documentation for details.

Parameters

- **name** – The name of the history block.
- **refPoint** – An integer three-vector. The grid indices of the reference point.
- **measPoint** – An integer three-vector. The grid indices of the measurement point.

addBinaryCombinationHistory (*name*, *coeff1*, *coeff2*, *op*, *history1*, *history2*)
Add history block that performs an element-wise binary operation on two other histories.

Parameters

- **name** – The name of the history block.
- **coeff1** – A real valued coefficient for history1.
- **coeff2** – A real valued coefficient for history2.
- **op** – The operation to perform on the two histories.
- **history1** – The name of the first history.
- **history2** – The name of the second history.

addPoyntingHistory (*name*, *lb*, *ub*)
Add history block that records the poynting flux on some plane within the simulation domain.

Parameters

- **name** – The name of the history block.
- **lb** – A real valued three-vector. The lower bounds on which to record, in coordinates.
- **coeff2** – A real valued three-vector. The upper bounds on which to record, in coordinates.

addFarFieldHistory (*name*, *timeDelay*, *duration*, *numTimePts*, *kirchRad*, *numPolarAng*, *numAzimAng*, *numLinInts*, *comp*, *kirchCenter*, *field*)
Add history block that records the far field radiation pattern of an EM source.

Parameters

- **name** – The name of the history block.
- **timeDelay** – The simulation time at which to begin recording.
- **duration** – The length of time during which the history records.
- **numTimePts** – The number of sampling points on the time interval.
- **kirchRad** – The radius of the Kirchhoff sphere.
- **numPolarAng** – The number of polar angles represented on the Kirchhoff sphere.
- **numAzimAng** – The number of azimuthal angles represented on the Kirchhoff sphere.
- **numLinInts** – The number of integration segments around the Kirchhoff sphere.
- **kirchCenter** – The center of the Kirchhoff sphere.

addFarFieldBoxHistory (*name*, *startTime*, *endTime*, *lc*, *uc*)
This history macro adds a set of histories that save fields on a 2-cell thick box, for use in the Kirchhoff Box Far Field post-processing.

Parameters

- **name** – The name of the history block.
- **startTime** – The simulation time at which the history begins recording.
- **endTime** – The simulation time at which the history stops recording.
- **lc** – The lower coordinates of the box (or slab).
- **uc** – The upper coordinates of the box (or slab).

kirchhoff.mac

This macro file can be imported to an input file with

```
$ import kirchhoff
```

It is imported at the top of all simulation runs.

This macro file is available to all packages.

This macro file defines the macro for computing the far field radiation pattern for a simulation.

Public Macros

addKirchhoffSurfaceIntegral()

Adds the Field, FieldUpdater and History blocks necessary to calculate the Kirchhoff Surface Integral to give the far field radiation pattern.

matrices.mac

This macro file can be imported to an input file with

```
$ import matrices
```

It is imported by VSimEs.mac.

This macro file is available to all packages.

This macro file defines macros for matrices. None of the macros are considered public, i.e., with stable APIs.

Public Macros

StencilElementMacro(*name*, *value*, *minDim*, *offset*, *rowFldIndx*, *colFldIndx*)

Insert a constant-value matrices StencilElement.

Parameters

- **name** – The name to give the stencil element.
- **value** – The value of the stencil element.
- **minDim** – The minimum dimensionality to apply this stencil element.
- **offset** – The offset (in index space) of the column cell from the row cel.
- **rowFldIndx** – The index in the matrices for the row. This corresponds to the writefield for multiply, to the readfield for solve.

- **colFldIndx** – The index in the matrices for the column. This corresponds to the readfield for multiply, to the writefield for solve.

STFuncStencilElementMacro (*name*, *value*, *minDim*, *cellOffset*, *funcOffset*, *rowFldIndx*, *colFldIndx*)
Insert a function-value matrices StencilElement.

Parameters

- **name** – The name to give the stencil element.
- **value** – The value of the stencil element.
- **minDim** – The minimum dimensionality to apply this stencil element.
- **cellOffset** – The offset (in index space) of the column cell from the row cel.
- **funcOffset** – The offset (in real space, as a multiple of the grid spacing) of the function evaluation point from the row cell.
- **rowFldIndx** – The index in the matrices for the row. This corresponds to the writefield for multiply, to the readfield for solve.
- **colFldIndx** – The index in the matrices for the column. This corresponds to the readfield for multiply, to the writefield for solve.

CoordProdSTFuncStencilElementMacro (*name*, *value*, *minDim*, *cellOffset*, *funcOffset*, *gridDir*, *gridOffset*, *rowFldIndx*, *colFldIndx*)
Insert a function-value matrices StencilElement in coordProd grid.

Parameters

- **name** – The name to give the stencil element.
- **value** – The value of the stencil element.
- **minDim** – The minimum dimensionality to apply this stencil element.
- **cellOffset** – The offset (in index space) of the column cell from the row cel.
- **funcOffset** – The offset (in real space, as a multiple of the grid spacing) of the function evaluation point from the row cell.
- **gridDir** – The direction in which differencing is being done for the stencil. Determines which grid spacing and face area to use.
- **gridOffset** – The offset (in index space) from the row cell at which the grid spacing and face area are taken.
- **rowFldIndx** – The index in the matrices for the row. This corresponds to the writefield for multiply, to the readfield for solve.
- **colFldIndx** – The index in the matrices for the column. This corresponds to the readfield for multiply, to the writefield for solve.

multifields.mac

This macro file can be imported to an input file with

```
$ import multifields
```

It is imported by VSim in all cases (electromagnetic and electrostatic).

This macro file is available to all packages.

This macro file contains many macros which add initial conditions to fields. It also contains macros to setup common operations with fields such as needed to make nodal fields, add applied (external) magnetic fields, copy perimieter fields, and perform binary operations with fields. A master macro, `writeMultiField()`, writes the <MultiField> block with all of the specified field operations.

Public Macros

`addInitialCondition` (*fieldName*, *numComponents*, *comp0*, *comp1*, *comp2*)
Add initial conditions to field blocks.

Parameters

- **`fieldName`** – The name of the field into which the initial condition block is written. Determines the name of the InitialCondition block.
- **`numComponents`** – The number of components in the initial condition.
- **`comp0`** – Expression (or None) for the component 0 of the initial condition.
- **`comp1`** – Expression (or None) for the component 1 of the initial condition.
- **`comp2`** – Expression (or None) for the component 2 of the initial condition.

`addInitialCondition` (*fieldName*, *numComponents*, *comp0*, *comp1*, *comp2*, *lb*, *ub*)
Add initial conditions to field blocks.

Parameters

- **`fieldName`** – The name of the field into which the initial condition block is written. Determines the name of the InitialCondition block.
- **`numComponents`** – The number of components in the initial condition.
- **`comp0`** – Expression (or None) for the component 0 of the initial condition.
- **`comp1`** – Expression (or None) for the component 1 of the initial condition.
- **`comp2`** – Expression (or None) for the component 2 of the initial condition.
- **`lb`** – The grid index (integer) lowerBounds of the initial condition.
- **`ub`** – The grid index (integer) upperBounds of the initial condition.

`addFieldInitialCondition` (*fieldName*, *component*, *function*)
Add an initial update step for initial conditions.

Parameters

- **`fieldName`** – fieldName The name of the field being initialized.
- **`component`** – The (integer) component of the field to initialize.
- **`function`** – The function to which the field will be initialized to.

particles.mac

This macro file can be imported to an input file with

```
$ import particles
```

It is imported by VSim if `VPM_INCLUDE_PARTICLES` is true.

This macro file is available to all packages.

This macro file defines the `finalizeParticles()` macro, specific to particle simulations. It contains the macros for defining particle species. It also contains macros for <ParticleSources> whcih load the particles into the simulation, for <ParticleSinks> which remove particles from a simulation.

Public Macros

addParticleSpecies (*speciesName*, *kind*, *charge*, *mass*, *nominalDensity*, *weighting*, *weightSetting*, *ppmorc*)

Add a particle species to a simulation by adding its data to the accumulation variables. Other accumulation variables, like those for sources and sinks, are added to by the other "add" methods. finalizeParticles() writes out the species with their sources and sinks.

Parameters

- **speciesName** – The name of the particle species.
- **kind** – The kind of species, e.g. relBoris, relBorisCyl, etc.
- **charge** – The charge of a physical particle of the species, in Coulombs.
- **mass** – The mass of a physcial particle of the species, in Kg.
- **nominalDensity** – A convenient (anticipated) density for the species, in 1/meters^3. Together with nomPtclsPerCell, it can be used to set the wieght of a constantWeight macroparticles, or the normalization of variableWeight macroparticles.
- **weighting** – The weighting for this species, either "constantWeights" or "variableWeights".
- **weightSetting** – How the weight is set, either explicitlySetWeights, or computedWeights from nominalDensity.
- **ppmorc** – For computedWeights, the number of macro particles in a cell, when the density in that cell is equal to nominal density. For explicitlySetWeights, the number of physical particles represented by one macro particle.

addParticleSpeciesLoader (*sourceName*, *owningSpecies*, *initOrRepeat*, *denTyp*, *denFunc*, *posGenKind*, *vbar*, *vsig*, *lb*, *ub*)

Add a particle loader to a particle species given the mean and standard deviation of the velocity distribution of the loaded particles. Loading is only on initialization.

Parameters

- **sourceName** – The name of the particle loader.
- **owningSpecies** – The name of the species to be loaded.
- **initOrRepeat** – A string denoting whether the loading is initialize only or to repeat.
- **denTyp** – Either physics load density or relative (to nominal) load density.
- **denFunc** – The function to use for loading.
- **posGenKind** – The kind of position generator (grid, bit-rev, . . .).
- **vbar** – A three-vector of mean velocities.
- **vsig** – A three-vector of rms velocities.
- **lb** – An NDIM vector of loower coordinates.

- **ub** – An NDIM vector of upper coordinates.

addParticleSpeciesLoader (*sourceName*, *owningSpecies*, *initOrRepeat*, *tmin*, *tmax*, *denTyp*, *denFunc*, *posGenKind*, *vbar*, *vsig*, *lb*, *ub*)

Add a particle loader to a particle species given the mean and standard deviation of the velocity distribution of the loaded particles. Loading is only on initialization. Delegates to addParticleSpeciesLoader(14 args).

Parameters

- **sourceName** – The name of the particle loader.
- **owningSpecies** – The name of the species to be loaded.
- **initOrRepeat** – A string denoting whether the loading is initialize only or to repeat.
- **tmin** – Tthe time to start loading.
- **tmax** – The time to stop loading (inclusive).
- **denTyp** – Either physics load density or relative (to nominal) load density.
- **denFunc** – The function to use for loading.
- **posGenKind** – The kind of position generator (grid, bit-rev, . . .).
- **vbar** – A three-vector of mean velocities.
- **vsig** – A three-vector of rms velocities.
- **lb** – An NDIM vector of loower coordinates.
- **ub** – An NDIM vector of upper coordinates.

addParticleSpeciesSink (*sinkName*, *owningSpecies*, *sinkKind*, *volType*, *lb*, *ub*)

Add a particle species sink to a particle species given the lower and upper bounds.

Parameters

- **sinkName** – The name of the particle sink.
- **owningSpecies** – The name of the species that contains this source block.
- **sinkKind** – The kind of sink. E.g. absAndSav, absorber, specularBndry, absAndSavCutCell.
- **volType** – The type of volume, either slab or location.
- **lb** – The lower bounds. An integer array of length VPM_NDIM.
- **ub** – The upper bounds. An integer array of length VPM_NDIM.

addParticleSpeciesSink (*sinkName*, *owningSpecies*, *sinkKind*, *location*)

Add a particle species sink to a particle species given the location.

Parameters

- **sinkName** – The name of the particle sink.
- **owningSpecies** – The name of the species that contains this source block.
- **sinkKind** – The kind of sink. E.g. absAndSav, absorber, specularBndry, absAndSavCutCell.
- **location** – location The location of the boundary condition, one of [lower|upper][X|Y|Z].

addParticleSpeciesSink (*sinkName*, *owningSpecies*, *sinkKind*, *volType*, *lb*, *ub*, *location*)

Add a particle species sink to a particle species. Either location or lb and ub should be None, as they are mutually exclusive. Done by adding to the string arrays describing the sink.

Parameters

- **`sinkName`** – The name of the particle sink.
- **`owningSpecie`** – The name of the species that contains this source block.
- **`sinkKind`** – The kind of sink. E.g. absAndSav, absorber, specularBndry, absAndSavCut-Cell.
- **`volType`** – The type of volume, either slab or location.
- **`lb`** – The lower bounds. An integer array of length VPM_NDIM.
- **`ub`** – The upper bounds. An integer array of length VPM_NDIM.
- **`location`** – location The location of the boundary condition, one of [lower|upper][X|Y|Z].

`finalizeParticles`()

Writing the particle Species blocks. Inserts Sinks and Sources (loaders and emitters) into the block. Writes depositor fields if electrostatic simulation.

physsetemits.mac

This macro file can be imported to an input file with

```
$ import physsetemits
```

It is imported at the top of all simulation runs.

This macro file is available to all packages.

This macro file defines the macros for setting up and writing physics-set-flux emitters into particle species definition.

Public Macros

`addPhysicsSetFluxEmitterSource`(*sourceName*, *sourceKind*, *owningSpecies*, *start*, *stop*, *vbar*, *vsig*, *workFunc*, *fldEvalOff*, *temp*, *fldEnhance*, *fluxMult*, *A*, *B*, *Cv*, *Cy*, *emOff*, *maskFunction*, *object*, *location*)

Adds a particle emitter source to a particle species. This macro handles the case of emission from a shape geometry.

Parameters

- **`sourceName`** – The name of the source block.
- **`sourceKind`** – The kind of emission model being used.
- **`owningSpecies`** – The name of the species that contains this source block.
- **`star`** – Time to start emission.
- **`stop`** – Time to end emission.
- **`vbar`** – The mean velocity of the distribution of emitted particles.
- **`vsig`** – The standard deviation of the velocity distribution of emitted particles.
- **`workFunc`** – The work function. The minimum energy required to emit an electron (eV).
- **`fldEvalOff`** – Field evaluation offset. Presumably the offset from the surface where the field resulting from the particle is evaluated. No use of this in the regression tests.
- **`temp`** – Temperature of the emitting surface (K).

- **`fldEnhance`** – Field enhancement. Multiplies the measured electric field by this amount.
- **`fluxMult`** – Flux multiplier. Multiplies the resulting output current by this amount.
- **`A`** – Coefficient A of the Fowler-Nordheim emission model.
- **`B`** – Coefficient B of the Fowler-Nordheim emission model.
- **`Cv`** – Coefficient Cv of the Fowler-Nordheim emission model.
- **`Cy`** – Coefficient Cy of the Fowler-Nordheim emission model.
- **`emOff`** – The distance away from the object that emission begins, a computational contrivance.
- **`maskFunction`** – The masking function to tailor emission as desired.
- **`object`** – The shape geometry to emit off of.
- **`location`** – The plane from which to emit.

`addChildLangmuirEmitter` (*sourceName*, *owningSpecies*, *start*, *stop*, *vbar*, *vsig*, *emOff*, *maskFunction*, *object*)
Adds a particle emitter source to a particle species with Child Langmuir properties. Child Langmuir specifies the mean and standard deviation of a velocity distribution.

`addRichardsonDushmanEmitter` (*sourceName*, *owningSpecies*, *start*, *stop*, *fldEnhance*, *fldEvalOff*, *fluxMult*, *temp*, *workFunc*, *emOff*, *maskFunction*, *object*)
Adds a particle emitter source to a particle species with Reichardson Dushman properties.

`addFowlerNordheimEmitter` (*sourceName*, *owningSpecies*, *start*, *stop*, *A*, *B*, *Cv*, *Cy*, *fldEnhance*, *workFunc*, *emOff*, *maskFunction*, *object*)
Adds a particle emitter source to a particle species with Fowler Nordheim properties.

secondemits.mac

This macro file can be imported to an input file with

```
$ import secondemits
```

It is imported by VSim whenever a simulation contains particles.

This macro file is available to all packages.

This macro file defines macros for adding secondary emitters for the various kinds of secondary emission supported by vorpal.

Public Macros

`addSecondaryElectronEmitter` (*secElecSpeciesName*, *secElecElectronSpeciesName*, *start*, *stop*, *material*, *suppressEnergy*, *secElecParticleSinkName*)
Add a secondary electron particle source block to the simulation. Emission is from a primary particle source's absorber which can either be a shape (imported or created through CSG) or a boundary of the simulation domain. Emission takes into account material properties. Delegates to addSecondaryElectrons(10 args).

Parameters

- **`secElecSpeciesName`** – The name of the secondary electron particle source block.
- **`secElecElectronSpeciesName`** – The name of the emitted electron species.
- **`start`** – The simulation time at which emission begins.

- **`stop`** – The simulation time at which emission ends.
- **`material`** – A material that determines secondary electron yield. Can be either "copper" or "stainless".
- **`suppressEnergy`** – A real number, units eV, denoting the suppressing energy of the local electric field. If the local field is of sign that would immediately push the electron back into the surface, then emission does not occur. If emission is desired regardless than this number is set to a very large number. Default is 1e20.
- **`secElecParticleSinkName`** – The name of the particle sink that the secondary electrons are to be emitted from. Must be a Boundary Absorb and Save or a Cut Cell Absorb and Save.

`addInteriorSecondaryElectronEmitter` (*secElecSpeciesName*, *secElecElectronSpeciesName*, *start*, *stop*, *material*, *suppressEnergy*, *emissionAxis*, *emissionDir*, *emissionCoord*, *secElecParticleSinkName*)

Add a secondary electron particle source block to the simulation. Emission is from a primary particle source's absorber which is interior to the simulation domain. Emission takes into account material properties. Emission direction and offset can be defined as desired. Delegates to addSecondaryElectrons(10 args).

Parameters

- **`secElecSpeciesName`** – The name of the secondary electron particle source block.
- **`secElecElectronSpeciesName`** – The name of the emitted electron species.
- **`start`** – The simulation time at which emission begins.
- **`stop`** – The simulation time at which emission ends.
- **`material`** – A material that determines secondary electron yield. Can be either "copper" or "stainless".
- **`suppressEnergy`** – A real number, units eV, denoting the suppressing energy of the local electric field. If the local field is of sign that would immediately push the electron back into the surface, then emission does not occur. If emission is desired regardless than this number is set to a very large number. Default is 1e20.
- **`emissionAxis`** – An integer that defines the axis of emission (e.g. x,y, or z).
- **`emissionDir`** – A string: "positive" or "negative". Defines which direction for the secondary electrons to travel after emission.
- **`emissionCoord`** – A real valued coordinate that defines the emission offset, or the distance away from the absorber to emit.
- **`secElecParticleSinkName`** – The name of the particle sink that the secondary electrons are to be emitted from. Must be a Interior Absorb and Save.

`addSimpleSecondaryElectronEmitter` (*simpleSecSpeciesName*, *simpleSecElectronSpeciesName*, *start*, *stop*, *suppressEnergy*, *emittedEnergy*, *SEY*, *simpleSecParticleSinkName*)

Add a secondary electron particle source block to the simulation. The secondary electron is of the same species as the primary (absorbed) electron. Emission is from a primary electron's absorber which can either be a shape (imported or created through CSG) or a boundary of the simulation domain. The emitted electron's energy can be specified along with the secondary electron yield.

Parameters

- **`simpleSecSpeciesName`** – The name of the secondary electron particle source block.
- **`simpleSecElectronSpeciesName`** – The name of the emitted electron species.

- **start** – The simulation time at which emission begins.
- **stop** – The simulation time at which emission ends.
- **suppressEnergy** – A real number, units eV, denoting the suppressing energy of the local electric field. If the local field is of sign that would immediately push the electron back into the surface, then emission does not occur. If emission is desired regardless than this number is set to a very large number. Default is 1e20.
- **emittedEnergy** – A real number denoting the energy, in Joules, of the emitted electrons.
- **SEY** – A function of energy, in eV, that determines the secondary electron yield curve.
- **simpleSecParticleSinkName** – The name of the particle sink that the secondary electrons are to be emitted from. Must be a Boundary Absorb and Save or a Cut Cell Absorb and Save.

addConstantProbabilitySecondaryElectronEmitter (*constProbSecSpeciesName*, *constProbSecElectronSpeciesName*, *start*, *stop*, *emissionProb*, *constProbSecParticleSinkName*)

Add a secondary electron particle source block to the simulation. Emission is from the primary species' absorber which can either be a shape (imported or created through CSG) or a boundary of the simulation domain. The probability for emission can be explicitly set.

Parameters

- **constProbSecSpeciesName** – The name of the secondary electron particle source block.
- **constProbSecElectronSpeciesName** – The name of the emitted electron species.
- **start** – The simulation time at which emission begins.
- **stop** – The simulation time at which emission ends.
- **emittedProb** – A real valued number on [0.0, 1.0] denoting the probability of emitting a secondary electron.
- **constProbSecParticleSinkName** – The name of the particle sink that the secondary electrons are to be emitted from. Must be a Boundary Absorb and Save or a Cut Cell Absorb and Save.

addSputterEmitter (*sputterSpeciesName*, *sputterElectronSpeciesName*, *start*, *stop*, *sputterEmittedSpeciesType*, *vsig*, *sputterParticleSinkName*)

Add a secondary neutral particle source block to the simulation. Emission is from the primary species' absorber which can either be a shape (imported or created through CSG) or a boundary of the simulation domain. The type of atom to be sputtered can be selected along with the standard deviation of the velocity of the emitted particles. Delegates to addSputterer(10 args).

Parameters

- **sputterSpeciesName** – The name of the secondary electron particle source block.
- **sputterElectronSpeciesName** – The name of the emitted electron species.
- **start** – The simulation time at which emission begins.
- **stop** – The simulation time at which emission ends.
- **sputterEmittedSpeciesType** – The type of atom that is to be sputtered. See the sputter emitter documentation for a list of valid types.
- **vsig** – A three-vector of rms velocities.

- **sputterParticleSinkName** – The name of the particle sink that the secondary particles are to be emitted from. Must be a Boundary Absorb and Save or a Cut Cell Absorb and Save.

addSputterEmitter (*sputterSpeciesName*, *sputterElectronSpeciesName*, *start*, *stop*, *sputterEmittedSpeciesType*, *vsig*, *emissionAxis*, *emissionDir*, *emissionCoord*, *sputterParticleSinkName*)

Add a secondary neutral particle source block to the simulation. Emission is from the primary species' absorber which can either be a shape (imported or created through CSG) or a boundary of the simulation domain. The type of atom to be sputtered can be selected along with the standard deviation of the velocity of the emitted particles. Emission direction and offset can be defined as desired. Delegates to addSputterer(10 args).

Parameters

- **sputterSpeciesName** – The name of the secondary electron particle source block.
- **sputterElectronSpeciesName** – The name of the emitted electron species.
- **start** – The simulation time at which emission begins.
- **stop** – The simulation time at which emission ends.
- **sputterEmittedSpeciesType** – The type of atom that is to be sputtered. See the sputter emitter

documentation for a list of valid types.

Parameters

- **vsig** – A three-vector of rms velocities.
- **emissionAxis** – An integer that defines the axis of emission (e.g. x,y, or z).
- **emissionDir** – A string: "positive" or "negative". Defines which direction for the secondary electrons to travel after emission.
- **emissionCoord** – A real valued coordinate that defines the emission offset, or the distance away from the absorber to emit.
- **sputterParticleSinkName** – The name of the particle sink that the secondary particles are to be emitted from. Must be a Boundary Absorb and Save or a Cut Cell Absorb and Save.

shapes.mac

This macro file can be imported to an input file with

```
$ import shapes
```

It is always imported by VSim at the top level.

This macro file is available to all packages.

This macro file defines the macros for adding shapes of different materials.

Public Macros

addShape (*cadBoundaryName*, *cadFileName*, *material*)

Add a shape by writing the GridBoundary block to the shapes accumulation variable. If PEC, set `VPM_CONDUCTING_GRID_BOUNDARY_NAME` to this name Else If dielectric, add to the dielectric shapes, etc.

Parameters

- **`cadBoundaryName`** – The name of the GridBoundary.
- **`cadFileName`** – The name of the (stl) file that contains the geometry.
- **`material`** – The name of the material that the boundary is made of

`addConductor` (*materialName*, *resistance*)
Add a conducting material.

Parameters

- **`materialName`** – The name of the material.
- **`resistance`** – The resistance of the material.

`addDielectric` (*materialName*, *relativePermittivity*, *conductivity*)
Add a dielectric material.

Parameters

- **`materialName`** – The name of the material.
- **`relativePermittivity`** – The relative permittivity of the material.
- **`conductivity`** – The conductivity of the dielectric object.

solvers.mac

This macro file can be imported to an input file with

```
$ import solvers
```

It is imported by VSimEs.mac.

This macro file is available to all packages.

This macro file provides macros for Poisson solvers, which are described below.

Public Macros

`addBaseSolver` (*name*, *krySize*, *orthog*)
Add the base solver block for an iterative solver to the accumulation variable, **`VPM_MATRIX_SOLVER`**, which will be put into the simulation.

Parameters

- **`name`** – Name for the updater.
- **`krySize`** – The size of the Krylov subspace. For gmres only.
- **`orthog`** – The orthogonality type for the Krylov subspace. For gmres only.

stfuncs.mac

This macro file can be imported to an input file with

```
$ import stfuncs
```

It is always imported by VSim at the top level.

This macro file is available to all packages.

This macro file defines the macros for adding shapes of different materials.

Public Macros

addGaussianPulseSTFuncBlock(*stName*, *omega*, *k0*, *k1*, *k2*, *amplitude*, *phase*, *origin0*, *origin1*, *origin2*, *widths0*, *widths1*, *widths2*, *vg*, *waistDisplacement*, *L_fwhm*, *keepon*)

This macro adds a block with all of the parameters and arguments necessary for a built-in Gaussian pulse. See *gaussianPulse* for explanation of the options.

Parameters **blah** – The blah param.

addChirpWavePulseSTFuncBlock(*stName*, *chirp*, *k0*, *k1*, *k2*, *amplitude*, *phase*, *origin0*, *origin1*, *origin2*, *widths0*, *widths1*, *widths2*, *vg*, *waistDisplacement*, *skewness*, *keepon*)

This macro adds a block with all of the parameters and arguments necessary for a built-in Chirp Wave pulse. See *chirpWavePulse* for explanation of the options.

addCosineFlattopSTFuncBlock(*stName*, *startPosition*, *startFlattop*, *endPosition*, *endFlattop*, *startAmplitude*, *endAmplitude*, *flattopAmplitude*, *direction*)

This macro adds a block with all of the parameters and arguments necessary for a built-in Cosine Flattop function. See *cosineFlattop* for explanation of the options.

addCosineRampSTFuncBlock(*stName*, *startPosition*, *endPosition*, *startAmplitude*, *endAmplitude*, *direction*)

This macro adds a block with all of the parameters and arguments necessary for a built-in Cosine Ramp function. See *cosineRamp* for explanation of the options.

addFeedbackSTFuncBlock(*stName*, *stFuncExpression*, *historyName*, *historyGoal*, *timeConstant*)

This macro adds a block with all of the parameters and arguments necessary for a built-in Feedback function. See *feedbackSTFunc* for explanation of the options.

addGaussianSTFuncBlock(*stName*, *amplitude*, *origin0*, *origin1*, *origin2*, *widths0*, *widths1*, *widths2*, *velocity0*, *velocity1*, *velocity2*)

This macro adds a block with all of the parameters and arguments necessary for a built-in Gaussian function. See *gaussian* for explanation of the options.

addHalfSinePulseSTFuncBlock(*stName*, *omega*, *k0*, *k1*, *k2*, *amplitude*, *phase*, *origin0*, *origin1*, *origin2*, *widths0*, *widths1*, *widths2*, *vg*, *waistDisplacement*, *skewness*, *keepon*)

This macro adds a block with all of the parameters and arguments necessary for a built-in Half Sine pulse. See *halfSinePulse* for explanation of the options.

addLeakyChannelSTFuncBlock(*stName*, *channelPosition0*, *channelPosition1*, *channelPosition2*, *maxRadius*, *maxParabRadius*, *centerAmplitude*, *quadCoef*, *expanFunc*, *direction*)

This macro adds a block with all of the parameters and arguments necessary for a built-in Leaky Channel function. See *leakychannel* for explanation of the options.

addPlaneWavePulseSTFuncBlock(*stName*, *omega*, *k0*, *k1*, *k2*, *amplitude*, *phase*, *origin0*, *origin1*, *origin2*, *widths0*, *widths1*, *widths2*, *vg*, *skewness*, *keepon*)

This macro adds a block with all of the parameters and arguments necessary for a built-in Plane Wave pulse. See *planeWavePulse* for explanation of the options.

addRadialCosineChannelSTFuncBlock(*stName*, *channelPosition*, *startRadius*, *endRadius*, *startAmplitude*, *endAmplitude*, *direction*)

This macro adds a block with all of the parameters and arguments necessary for a built-in Radial Cosine Channel function. See *radialCosChannel* for explanation of the options.

addSinePlaneWaveSTFuncBlock(*stName*, *omega*, *k0*, *k1*, *k2*, *amplitude*, *phase*)
This macro adds a block with all of the parameters and arguments necessary for a built-in Sine Plane Wave function. See *sinePlaneWave* for explanation of the options.

addSumFunctionSTFuncBlock(*stName*, *sumName1*, *sumName2*)
This macro adds a block with all of the parameters and arguments necessary for a built-in Sum function. See *sumFunc* for explanation of the options.

addProductFunctionSTFuncBlock(*stName*, *prodName1*, *prodName2*)
This macro adds two blocks with all of the parameters and arguments necessary for a built-in Multifunc product function. See *multFunc* for explanation of the options.

VSim.mac

This macro file can be imported to an input file with

```
$ import VSim
```

It is imported at the top of all simulation runs.

This macro file is available to all packages.

This macro file defines the `finalize()` macro, which writes out the blocks defined by accumulation variables. This file also defines many of the global accumulation variables.

Public Macros

finalize()
Write out all the previously defined blocks in the correct order and with the correct nesting.

VSimEm.mac

This macro file can be imported to an input file with

```
$ import VSimEm
```

It is imported by VSim if `VPM_SIMULATION_TYPE` is electromagnetic.

This macro file is available to all packages.

This macro file defines the `finalizeUpdaters()` macro, specific to electromagnetic simulations, and it also defines many macros for adding EM update steps.

Public Macros

writeMovingWindow(*shiftPos*, *shiftSpd*)

Write a moving window in the X-Direction block to the top level of the .in file.

Parameters

- **shiftPos** – The fractional amount of the domain for the speed below to cross before shifting.

- **shiftSpd** – The fraction of light speed to move the window by.

writeMovingWindow(*shiftDir*, *shiftPos*, *shiftSpd*)
Write a moving window to the top level of the .in file with specifiable direction.

Parameters

- **shiftDir** – The capital character describing the (Cartesian) direction to shift the window. One of X,Y,Z.
- **shiftPos** – The fractional amount of the domain for the speed below to cross before shifting.
- **shiftSpd** – The fraction of light speed to move the window by.

addCurrentDistribution(*name*, *j0*, *j1*, *j2*, *lc*, *uc*)
Add a current distribution into a slab.

Parameters

- **name** – The name of this current distribution.
- **j0** – Function giving component 0 (x) of the current distribution.
- **j1** – Function giving component 1 of the current distribution.
- **j2** – Function giving component 2 of the current distribution.
- **lc** – Lower limits of the coordinate slab over which to apply this current distribution
- **uc** – Upper limits of the coordinate slab over which to apply this current distribution

addDipoleCurrentDistribution(*name*, *comp*, *function*, *coordinate*)
Add a dipole current at a coordinate.

Parameters

- **name** – The name of this dipole current distribution.
- **comp** – The component of j that is being set.
- **function** – Function that the component of j will be set to.
- **coordinate** – The location for this dipole current.

finalizeUpdaters()
Create all of the updaters needed for an EM simulation. Writing them occurs in writeMultiField, which is called by finalize().

VSimEs.mac

This macro file can be imported to an input file with.

```
$ import VSimEs
```

It is imported by VSim if `VPM_SIMULATION_TYPE` is electrostatic.

This macro file is available to all packages.

This macro file defines the `finalizeUpdaters()` macro, specific to electrostatic simulations, and it also defines many private macros for adding ES update steps.

Public Macros

finalizeUpdaters()

Noindex

Add in all of the updaters needed for an ES simulation. Writing them occurs in writeMultiField, which is called by finalize().

VSimMs.mac

This macro file can be imported to an input file with

```
$ import VSimMs
```

This macro file is available to all packages.

This macro file defines the macros for magnetostatic simulations.

Public Macros

There are no public macros for the file.

VSimPf.mac

This macro file can be imported to an input file with.

```
$ import VSimPf
```

This macro file is available to all packages.

This macro file defines macros for the updaters of EM fields to provide high resolution data for calculating secondary electron production and multipacting effects Will only work with PEC grid boundaries

This macro file defines the following public macros:

Public Macros

addTimeDependence(*name*, *function*)

Sets the global variable VPM_MODE_FUNCTION. This is the time dependent mode function of a cavity.

Parameters **name** – The name of the cavity in which to add the

time-dependent mode function.

Parameters **function** – The time-dependent function of the mode.

3.19.5 Deprecated Macros

Macros from VSim 8 and earlier version have been deprecated. We include the previous documentation of these macros in the list below until they are removed. Deprecated macros also include: antennasGPU.mac, expressions.mac, dielectricGPU.mac, farFieldsGPU.mac, geometryGPU.mac, gpuPml.mac, malGPU.mac, scatteringBoxGPU.mac, usegpu.mac, yee.mac, yeeGPU.mac, yeeMacro.mac, version.mac, and yeeNew.mac.

DrudeDebyeLorentzDielectric.mac

This macro file can be imported to an input file with `$ import DrudeDebyeLorentzDielectric`.

This macro file is available with the VSimEM package.

This macro file contains macros to assist in the creation of a frequency dependant dielectric following the Drude, Debye Relaxation, and/or Lorentz Resonant Models.

In these models, the total displacement and conduction current will be the following

$$J_{total} = J_{\infty} + J_{DrudeDebye} + \Sigma J_{Lorentz}$$

$$[-i\omega\epsilon(\omega) + \sigma(\omega)]E = [-i\omega\epsilon_{R\infty}\epsilon_0 + \sigma_{\infty} + \frac{(\sigma_{Static}-\sigma_{\infty})}{(1-i\omega t_{collision})} + \Sigma_L \frac{-i\omega\alpha_L}{(\omega_L^2-\omega^2-i\omega\Gamma_L)}]E$$

There are 10 macros, 3 are mandatory (`createRegion`, `initialize`, `update`), and 7 are optionally used to control the material properties (`setInfLimits`, `SetAnalyticBackground`, `setDrudeByCarrier`, `setDrudeByConductance`, `setDebye`, `setLorentzByCarrier`, `setLorentzByConductanceRate`). Their calling sequence is listed below, with respect to each other, and the other blocks in the input file. More detail on each of the calls follows in the discussion.

Input file structure

```
<Multifield ...>

    <Field ...>s and <FieldUpdater ...>s for other physics

  createRegion_DDLD(edgeEname,objectName,nxbgn,nybgn,nzbgn,nxend,nyend,nzend,timeStep)

  ### OPTIONAL, NEITHER, EITHER, OR BOTH ###
  setAnalyticBackground_DDLD(epsRelExpression,sigmaExpression)
  setInfLimits_DDLD(epsRelInf,sigmaInf)

  ### OPTIONAL, AT MOST ONE OF THESE ###
  setDrudeByCarrier_DDLD(collisionTimeScale,carrierDensity,carrierCharge,carrierMass)
  setDrudeByConductance_DDLD(collisionTimeScale,sigmaStatic)
  setDebye_DDLD(relaxTime,epsRelStatic)

  ### OPTIONAL, MULTIPLE CALLS ALLOWED ###
  addLorentzByCarrier_DDLD(freqL,gammaL,carrierDensity,carrierCharge,carrierMass)
  addLorentzByConductanceRate_DDLD(freqL,gammaL,alphaL)

  initialize_DDLD()

  more <Fields ...>s and <FieldUpdater ...>s representing other physics

    <UpdateStep ...>s for other physics, up to and including
    <UpdateStep Ampere> and any E field Boundary Conditions

  update_DDLD()

    more <UpdateStep ...>s for other physics

</Multifield>
```

Create DDLD Region Macro

`createRegion_DDLD` (*edgeEname*, *objectName*, *nxbgn*, *nybgn*, *nzbgn*, *nxend*, *nyend*, *nzend*, *timeStep*)

Specifies that DDLD updaters will be applied in the region specified.

To get started, the user must call the `createRegion_DDLD` macro somewhere near the beginning of the `<MultiField ...>` block. This call requires the name of the edge electric `<Field ...>` used in the Ampere update, the name of the dielectric, as constructed in a `<GridBoundary ...>` object. It also requires the grid index limits over which to apply this model, and the time step.

Parameters

- **`edgeEname`** – Name of the (edge centred) E field to be used by the updater.
- **`objectName`** – Name of the `<GridBoundary ...>` object representing the DDLD.
- **`nxbgn`** – The index of the lower x extent for the DDLD updates.
- **`nybgn`** – The index of the lower y extent for the DDLD updates.
- **`nzbgn`** – The index of the lower z extent for the DDLD updates.
- **`nxend`** – The index of the upper x extent for the DDLD updates.
- **`nyend`** – The index of the upper y extent for the DDLD updates.
- **`nzend`** – The index of the upper z extent for the DDLD updates.
- **`timeStep`** – The timestep being used by the Yee algorithm.

Set Analytic Background Material Macro (Optional)

`setAnalyticBackground_DDLD` (*epsRelExpression*, *sigmaExpression*)

This algorithm does not treat or alter dielectric or conductance outside the user-provided grid index range, and it is required that the Ampere update within the user-provided index range is that of vacuum, e.g., $\epsilon_r = 1(epsRel = 1)$ and $\sigma = 0$ (sigma=0), whether or not that is actually the case. In particular, if in reality, the DDLD material is embedded in a material other than vacuum, this is treated with a call to `setAnalyticBackground_DDLD`, which creates a frequency-independent material within the user-specified index range, surrounding the DDLD's `<GridBoundary>` object. An analytic function of space is used to specify this background material's epsRel and sigma. The call to `setAnalyticBackground_DDLD` should be made immediately after the call to `createRegion_DDLD` call, and before any other set or add calls..

Parameters

- **`epsRelExpression`** – Relative permittivity. If using a function, then specify the function name as `functionName(x,y,z)`, or expression string.
- **`sigmaExpression`** – Conductivity. If using a function, then specify the function name as `functionName(x,y,z)`, or expression string, in 1/(ohms*meters).

Set Infinite Frequency Limit Macro (Optional)

`setInfLimits_DDLD` (*epsRelInf*, *sigmaInf*)

As a precaution to ensure stable behavior, the model must have well defined high-frequency limits of the dielectric and conductance, $\epsilon_{r\infty}$ (`epsRelInf`) and σ_∞ (`sigmaInf`). The defaults for these are vacuum, $\epsilon_{r\infty} = 1$ and $\sigma_\infty = 0$. But if residual high frequency dielectric or conductance is desired, in order to better fit the desired behavior in the frequency range of the simulation, call `setInfLimits_DDLD` to override the defaults. The call to `setInfLimits_DDLD` should be made immediately after the call to `createRegion_DDLD` and possibly `setAnalyticBackground_DDLD`, and before any other set or add calls. If a subsequent call to `setDebye_DDLD` is made, the value of sigmaInf will be overwritten.

Parameters

- **`epsRelInf`** – Relative permittivity limit at high frequencies. If using a function, then specify the function name as `functionName(x,y,z)`, or expression string.
- **`sigmaInf`** – Conductance limit at high frequencies, in 1/(ohms*meters).

Set Zero Frequency Limit - the Primary Options

In general, the low frequency behavior of the dielectric is different from the high freuqency limit, and is specificed with either a Drude model, or a Debye Relaxation Model, or neither, but not both. The Drude model focuses on the zero frequency limit of conductance, while the Debye Relaxation focuses on the zero frequency limit of dielectric. In the formula above, if we assume no zero-frequency Lorentz resonances, the zero frequency limit of conductance is indeed, sigmaStatic, from the Drude term. The zerofrequency limit of dielectric has contriubtions for all terms, however, and is

epsRelStatic = epsRelInf - (sigmaStatic-sigmaInf)*collisionTime/eps0 + SumL{ alphaL/ω_L^2 /eps0 }

$\epsilon_{rstatic} = \epsilon_{r\infty} - (\sigma_{static} - \sigma_{\infty}) \cdot t_{collision}/\epsilon_0 + \Sigma_L \frac{\alpha_L}{\omega_L^2 \epsilon_0}$

Drude Model, by carrier macro

`setDrudeByCarrier_DDLD`(*collisionTimeScale*, *carrierDensity*, *carrierCharge*, *carrierMass*)

The Drude model assumes unbound charge carriers which undergo collisions, resulting in a frequency dependant conduction current. sigmaStatic is the static (zero-frequency) limit of the conductance in this model. A call to `setDrudeByCarrier_DDLD` will set up the drude model parameters.

When this call is made, the zero frequency limit of conductance in the Drude Model is computed, according to the formula

sigmaStatic = collisionTime*(carrierDensity*carrierCharge^2/carrierMass)

param collisionTime Mean collision time in seconds.

param carrierDensity Number density of charge carriers in 1/meters^3.

param carrierCharge Unitless multiplier of the electron charge, e.

param carrierMass Unitless multiplier of the electron mass, m_e.

Drude Model, by conductance macro

`setDrudeByConductance_DDLD`(*collisionTimeScale*, *sigmaStatic*)

Alternately, the user may choose to setup the drude model directly with the resultant conductance, thus avoiding the need to know information about the carriers. A call to `setDrudeByConductance_DDLD` will do this.

Parameters

- **`collisionTime`** – Mean collision time, in seconds.
- **`sigmaStatic`** – 'Static' conductance in 1/(ohms*meters).

Debye Model

`setDebye_DDLD` (*relaxTime*, *epsRelStatic*)

The Debye Relaxation Model assumes bound charge carriers whose dielectric response relaxes at higher frequencies. Since bound charges cannot create conduction current, sigmaStatic=0 will be enforced by this call. The user provides epsRelStatic, the static (zero-frequency) limit of the dielectric, in the absence of any Lorentz resonances. the user also specifies relaxTime, the time scale at which dielectric response relaxes from epsRelStatic to epsRelInf. A call to `setDebye_DDLD` will set up the Debye Relaxation Model.

This model will reset collisionTime = relaxTime, and sigmaInf=(epsRelStatic-epsRelInf)/relaxTime, in order to convert the Drude term to the Debye Relaxation Model. This formulation then allows the combination of Infinity-Limit and Drude Terms to be recast as

[-i* ω *eps(ω) + sigma(ω)]*E = -i * ω *[epsRelInf + (epsRelStatic-epsRelInf)/(1-i* ω *relaxTime)]*E

$[-i\omega\epsilon(\omega) + \sigma(\omega)]E = -i\omega[\epsilon_{R\infty} + (\epsilon_{RStatic} - \epsilon_{R\infty})/(1 - i\omega t_r)]E$

Parameters

- **`relaxTime`** – Debye relaxation time, in seconds.
- **`epsRelStatic`** – Unitless relative dielectric permitivity in the static limit.

Intermediate Frequency Behavior (Optional)

The intermediate frequency behavior can optionally include one or more Lorentz Resonannces. The unusual case of a zero frequency Lorentz resonance, $f_L = 0$, would be equivalent to a Drude Model, and so would normally be handled using that model. However, it is allowed to produce multiple Drude terms in such a manner if such unusual behavior is actually desired. A zero line width, $\Gamma_L = 0$, is also permitted, although such a circumstance would also be considered unusual.

Lorentz Model, by Carrier Macro

`addLorentzByCarrier_DDLD` (*freqL*, *gammaL*, *carrierDensity*, *carrierCharge*, *carrierMass*)

The Lorentz Model assumes bound charge carriers whose response is resonant at some finite frequency, freqL, with a line width of the resonance given by gammaL. A Lorentz resonance will be added to the infinite and zero frequency limits with a call to `addLorentzByCarrier_DDLD`.

When this call is made, these parameters are used to produce the conductance rate in the Lorentz Current, according to

α_L = carrierDensity*carrierCharge^2/carrierMass

Parameters

- **`freqL`** – Lorentz frequency in Hertz, $= \omega_L/(2\pi)$.
- **`gammaL`** – Lorentz gamma in 1/seconds.
- **`carrierDensity`** – Number density of charge carriers in 1/meters^3.
- **`carrierCharge`** – Unitless multiplier of the electron charge, e.
- **`carrierMass`** – Unitless multiplier of the electron mass, m_e.

Lorentz Model, by Conductance Rate

addLorentzByConductanceRate_DDLD (*freqL*, *gammaL*, *alphaL*)

Alternately, the user may choose to setup the Loretnz resonance directly with the resultant conductance rate, thus avoiding the need to know information about the carriers. A call to `addLorentzByConductanceRate_DDLD` will do this.

Parameters

- **freqL** – Lorentz Frequency (f_L) in Hertz, $= \frac{\omega_L}{(2\pi)}$.
- **gammaL** – Lorentz Gamma (Γ_L) in 1/seconds.
- **alphaL** – Lorentz Alpha (α_L) in 1/(ohms*meters*seconds).

Initialization (Required)

initialize_DDLD()

Immediately after create, set, and add calls, the model must instantiate and initialize its internal fields. This is done with a call to `initialize_DDLD()`, which has no arguments.

If the input file is making use of the `updateStepOrder` attribute, instead of order of appearence in the input file, then the single Initial update step, `initializeDDLD`, must be inserted in the proper location within the list, before any of the `<UpdateStep>` s. The particular placement with respect to any other InitialUpdateSteps is not important.

Update Macro (Required)

update_DDLD()

The actual update of the DDLD is done with the update_DDLD() call, which has no arguments. The call must be placed just folowing the Ampere `<UpdateStep ...>` and any additional `<UpdateStep ...>` s which act as boundary conditions for the electric field update.

If the input file is making use of the `updateStepOrder` attribute, instead of order of appearence in the input file, then the single update step, `updateDDLD()`, must be inserted in the proper location within the list, after the Ampere and electric field boundary condition update steps.

Communication and Parallelization

The DDLD materials are a point model, so no communication of the internal fields is necessary across parallel processes. Thus while there are many `<FieldUpdater ...>` s involved in the initialization and update, they are all placed together in single steps, which have empty messageFields attributes.

Stability

The algorithm is implicit and so no additional Courant conditions are introduced on the time step. However, instabiity is still possible if the conductivity at some frequency between zero and 2/dt were to be negative. Some limiters are in place to help prevent such a circumstance should the user attempt to input negative conductance, for example. Similarly, there are limiters to help prevent relative dielectric below unity. Thus, these macros are not capable of treating negative index of refraction, for example, and should not be attempted to be used for such purposes.

absorbingBox.mac

This macro file can be imported to an input file with `$ import DrudeDebyeLorentzDielectric`.

This macro file is available to all packages.

This collection of macros will set up absorbers on all sides of the simulation domain. Also there are macros to set absorbers correctly when periodic boundaries are present. This macro is encrypted and available with the VSimBase license. It can be imported to an input file with *$ import absorbingBox* For Cartesian simulations, or Cylindrical simulations that exclude the axis region, the absorbingBox macro is appropriate. For cylindrical simulations in 2D, it may be more appropriate to use the *absorbingBoxCylAxis* macro, which enforces periodicity in the third dimension, normally *phi*, ϕ, and excludes absorbtion on the $r = 0$ axis. Additionally there is the *absSavBoxCylAxis*, which uses absAndSav particle sinks in place of absorbers.

The labelling of the faces, edges and corners is consistent across all macros contained in this macro file. See *Labels used for faces, edges and corners by the absorbing box macro.*.

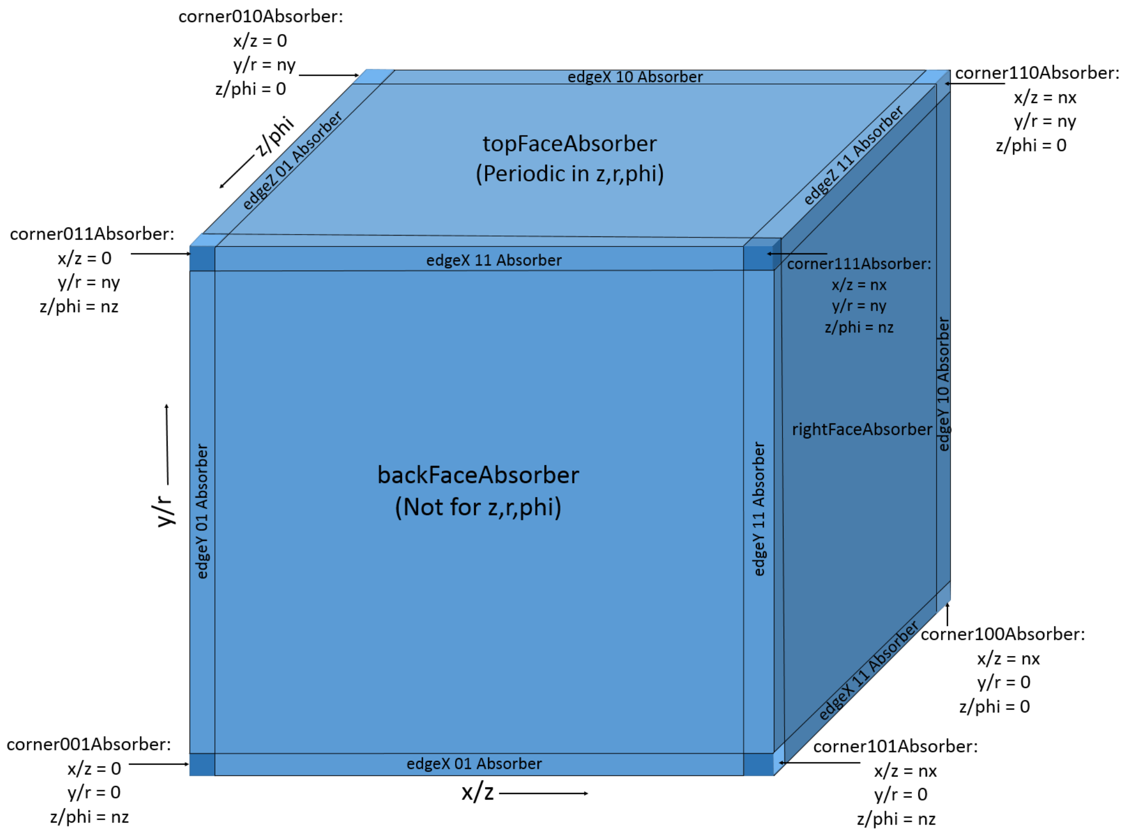


Fig. 3.4: Labels used for faces, edges and corners by the absorbing box macro.

absorbingBox Macro

absorbingBox (*nx*, *ny*, *nz*, *per_x*, *per_y*, *per_z*)
: Sets up fully absorbing boxes in any dimension. If per_x/y/z are specified, the faces that are periodic will not be absorbing.

absorbingBox (*nx*, *ny*, *nz*)
: Sets up fully absorbing boxes in any dimension, with no periodic boundaries.

Parameters

- **nx** – Number of cells in x direction.
- **ny** – Number of cells in y direction.
- **nz** – Number of cells in z direction.
- **per_x** – Is the simulation periodic in x (True/False).
- **per_y** – Is the simulation periodic in y (True/False).
- **per_z** – Is the simulation periodic in z (True/False).

absAndEmitBox Macro

absAndEmitBox (*nx*, *ny*, *nz*, *per_x*, *per_y*, *per_z*)

Sets up absorbing and reemitting boxes in any dimension. If per_x/y/z are specified the faces that are periodic will not be absorbing/remitting.

Parameters

- **nx** – Number of cells in x direction.
- **ny** – Number of cells in y direction.
- **nz** – Number of cells in z direction.
- **per_x** – Is the simulation periodic in x (True/False).
- **per_y** – Is the simulation periodic in y (True/False).
- **per_z** – Is the simulation periodic in z (True/False).

absorbingBoxCylAxis Macro

absorbingBoxCylAxis (*nz*, *nr*, *nphi*)

Sets up absorbing boxes with axis excluded, with phi periodic.

Parameters

- **nz** – Number of cells in z direction (often *NX*).
- **nr** – Number of cells in r direction.
- **nphi** – Number of cells in ϕ direction.

absSavBoxCylAxis Macro

absSavBoxCylAxis (*nz*, *nr*, *nphi*)

Sets up absorbing boxes that save particle information for reuse with axis excluded and with phi periodic.

Parameters

- **nz** – Number of cells in z direction (often *NX*).
- **nr** – Number of cells in r direction.
- **nphi** – Number of cells in ϕ direction.

adimacros.mac

This macro file can be imported to an input file with `$ import adimacros`.

This macro file is available to all packages.

`adimacros`()
Alternating Direction Implicit (ADI) macros are used with the implicit Solver. The `adimacros.mac` macros are for use with the VSim Multifield feature. Macros from adimacros.mac that VSim users may find helpful include:

Parameters

- **`adiUpdaters`** – For full ADI update.
- **`dmAdiUpdaters`** – For full ADI update with domains requiring GridBoundary.

adiUpdaters

`adiUpdaters`(*name*, *efld*, *bfld*)
Defines all of the operators for the full ADI update. The adiUpdaters macro is located in the file adimacros.mac.

Names start with `p` or `m` corresponding to P matrix and M matrix, respectively. Names end with the direction of the ADI. mult or solv refers to application or inversion of the matrix. The divergence preserving update applies these in the order:

- msolv
- pmult
- (add in current)
- psolv
- mmult

The curl steady state update applies these in the order:

- mmult
- pmult
- (add in current)
- psolv
- msolv

adiUpdaters macro parameters include:

- name: name assigned to the collection.
- efld: electric field
- bfld: magnetic field

dmAdiUpdaters

`dmAdiUpdaters`(*name*, *efld*, *bfld*, *bndry*, *inttype*)
Defines all of the operators for the full ADI update for domains requiring GridBoundaries. This allows users to simulation EM with implicit methods for domains that are circles, spheres, or cavities, for example. The dmAdiUpdaters macro is located in the file adimacros.mac.

Names start with `p` or `m` corresponding to P matrix or M matrix, respectively. Names end with the direction of the ADI. mult or solv refers to application or inversion of the matrix. The divergence preserving update applies these in the order:

- msolv
- pmult
- (add in current)
- psolv
- mmult

The curl steady state update applies these in the order:

- mmult
- pmult
- (add in current)
- psolv
- msolv

param name Name assigned to the collection.

param efld Electric field.

param bfld Magnetic field.

param bndry Name of the grid boundary.

param inttype The interiorness type; this should usually be deymittra.

antennas.mac

This macro file can be imported to an input file with `$ import antennas`.

This macro file is available with the VSimEM package.

This macro collections contains combinations of shape primitives to generate horn and parabolic antennnas, and add a history that will record far field data. There are two versions of this macro, one antennas is for use when doing computations on a CPU, the other antennasGPU is used for computations on a GPU. They retain the exact same functionality, however GPU and CPU macros cannot be mixed.

Horn Antenna Macro

`hornAntenna` (*name*, *dir*, *ofx*, *ofy*, *ofz*, *ww*, *wh*, *wl*, *al*, *ah*, *aw*, *mt*)
Defines a horn antenna with a rectangular aperture.

hornAntenna Macro Parameters

param name Name of the horn antenna.

param dir Axial direction of the horn antenna.

param ofx Offset in X (m).

param ofy Offset in Y (m).

param ofz Offset in Z (m).

param ww Width of waveguide (m).

param wh Height of waveguide (m).

param wh Height of waveguide (m).

param wl Length of waveguide (m).

param al Length of aperture (m).

param ah Height of aperture (m).

param aw Width of aperture (m).

param mt Thickness of horn antnenna (m).

Parabolic Antenna Macro

`parabolicAntenna` (*name*, *dir*, *ofx*, *ofy*, *ofz*, *dp*, *rd*, *mt*)

This macro defines a parabolic dish (no feed specified).

Parameters

- **`name`** – Name of the parabolic dish.
- **`dir`** – Axial direction of the parabolic dish.
- **`ofx`** – Offset in X (m).
- **`ofy`** – Offset in Y (m).
- **`ofz`** – Offset in Z (m).
- **`dp`** – Depth of the paraboloid (m).
- **`rd`** – Radius of the paraboloid at the aperture (m).
- **`mt`** – Thickness of horn antnenna (m).

Add Far Field History Macro

`addFarFieldHistory` (*historyName*, *NUM_THETA*, *NUM_PHI*, *RS*, *NS*, *HIST_DELAY*, *FREQUENCY*)

Adds a history to compute the far field radiation pattern. Far fields must be added with the addFarFields macro found in the farFields macro.

Parameters

- **`historyName`** – Name to give the history dataset.
- **`NUM_THETA`** – The number of theta points in the far field.
- **`NUM_PHI`** – The number of phi points in the far field.
- **`RS`** – The Radius of the Kirchhoff sphere–must be larger than the geometry and any scatteringBox, but smaller than any boundary conditions (most MALs protrude 1 wavelength into the simulation domain).
- **`NS`** – The number of the points in the discretization of the line integrals around the Kirchhoff sphere.
- **`HIST_DELAY`** – Delay time before far field transformation begins.

- **FREQUENCY** – The frequency at which the test device is operating.

addFarFieldHistory Output

The history output will be in the form of .h5 files, which will have titles `dataset_farField.vsh5`.In the *Visualization* window, you will be able to see a field called `historyName` as an option under the E field in *Scalar Data*. Here you will find the far field radiation pattern.

Add Far Field History Power User Macro

addFarFieldHistoryPowerUser(*historyName*, *emFieldName*, *NUM_THETA*, *NUM_PHI*, *RS*, *NS*, *HIST_DELAY*, *FREQUENCY*, *DFF*, *TFFMIN*, *TFFMAX*, *NTFAR*, *ANTENNA_X*, *ANTENNA_Y*, *ANTENNA_Z*)

This macro is intended for finding the radiation patterns of large scale structures. All parameters in computing the history are available to the user to facilitate this. Far fields must be added with the addFarFields macro found in the farFields macro.

This macro is encrypted and only available with the VSimEM license.

Parameters

- **historyName** – Name of the history.
- **emFieldName** – Name of the multifield.
- **NUM_THETA** – The number of theta points in the far field.
- **NUM_PHI** – The number of phi points in the far field.
- **RS** – The Radius of the Kirchhoff sphere–must be larger than the geometry and any scatteringBox, but smaller than any boundary conditions (most MALs protrude 1 wavelength into the simulation domain).
- **NS** – The number of the points in the discretization of the line integrals around the Kirchhoff sphere.
- **HIST_DELAY** – Delay time before far field transformation begins.
- **FREQUENCY** – The frequency at which the test device is operating.
- **DFF** – Distance to the far field (normally set to 10 m).
- **TFFMIN** – Time at which the far field starts being taken, at the far field. Normally (DFF+RS)/LIGHTSPEED + HIST_DELAY.
- **TFFMAX** – Time at which the far field stops being taken, at the far field. Normally TFFMIN + 1/FREQUENCY.
- **NTFAR** – Number of time points the far field is taken at. Normally 1.
- **ANTENNA_X** – Center of the Kirchhoff sphere on the x axis, normally 0.
- **ANTENNA_Y** – Center of the Kirchhoff sphere on the y axis, normally 0.
- **ANTENNA_Z** – Center of the Kirchhoff sphere on the z axis, normally 0.

average.mac

This macro file can be imported to an input file with `$ import average`.

This macro file is available to all packages.

This collection of macros will calculate the trailing average or window average of any 1D or 3D field.

averageFieldWindow Macro

averageFieldWindow(*FieldName*, *FieldOffset*, *Dimensionality*, *DUMP_PERIOD*, *DT*, *WinWidthSteps*, *WinWidthSec*, *lx*, *ly*, *lz*, *ux*, *uy*, *uz*)

Adds a field that is the window average of the field *FieldName*. A window average is an unweighted mean of the last n timesteps. The window's width is specified in units of timesteps (WinWidthSteps) or seconds (WinWidthSec). The unused window width argument must be set to *none*. Care must be taken with windows that are longer than the dump period; the average field will only dump when it has completed, which will not necessarily be every dump. After setting up all average fields, make a call to the appendAveSteps macro, below, to include the average fields in the updateStepOrder.

Parameters

- **FieldName** – Name of the field to be averaged.
- **FieldOffset** – Offset of the field to be averaged. Can be *none*, *edge*, *center*, *face*, or *edge4v*.
- **Dimensionality** – Integer number of components in field, 1 or 3.
- **DUMP_PERIOD** – Number of steps between dumps.
- **DT** – Length of timestep.
- **WinWidthSteps** – Time length of window in decimal timesteps; may be float or integer (e.g. 19.9); Set to `none` if using WinWidthSec.
- **WinWidthSec** – Time length of window in seconds; Set to `none` if using WinWidthStep.
- **lx** – Lower x integer cell number.
- **ly** – Lower y integer cell number.
- **lz** – Lower z integer cell number
- **ux** – Upper x integer cell number.
- **uy** – Upper y integer cell number.
- **uz** – Upper z integer cell number.

averageField Macro

averageField(*FieldName*, *FieldOffset*, *Dimensionality*, *DT*, *tau*, *lx*, *ly*, *lz*, *ux*, *uy*, *uz*)

Adds a field that is the trailing average of the field *FieldName*. A trailing average is the mean of all previous timesteps but weighted exponentially toward more recent values. It is also known as a Kalman filter, or low-pass filter. The trailing average is given by the equation,

$$\bar{n}_i = \left(1 - \frac{\delta t}{\tau}\right) n_{i-1} + \frac{\delta t}{\tau} n_i$$

where i is the current timestep, n_i is the field value at the current timestep, and δt is DT, the length of a timestep. The trailing factor, τ, is specified by the user. After all average fields are set up, make a call to the appendAveSteps macro, below, to include the average fields in the updateStepOrder.

Parameters

- **`FieldName`** – Name of the field to be averaged.
- **`FieldOffset`** – Offset of the field to be averaged.
- **`Dimensionality`** – Integer number of components in field, 1 or 3.
- **`DT`** – Length of timestep.
- **`tau`** – Trailing factor in seconds (e.g. 1.9e-10).
- **`lx`** – Lower x integer cell number.
- **`ly`** – Lower y integer cell number.
- **`lz`** – Lower z integer cell number.
- **`ux`** – Upper x integer cell number.
- **`uy`** – Upper y integer cell number.
- **`uz`** – Upper z integer cell number.

appendAveSteps Macro

`appendAveSteps`(*updateStepList*)

This macro helps solve the problem of the updateStepOrder variable, which can optionally be hard coded by the user to specify an order of update steps that differs from the natural order in the PRE code. If the user has specified the updateStepOrder, then the averaging fields must be added to the list or they will never update. The averageField UpdateStep must be appended to updateStepOrder after the updateStep for whatever field you are averaging. Call this macro from within the <MultiField> block.

To use this macro,

1. Assign your updateStepOrder to the variable updateStepList; make the list without brackets. If you have set up your multifield with macros like yee, this will have been done automatically. Example: updateStepList = step1 step2 step3
2. Call the macro: $ updateStepList = appendAveSteps(updateStepList)
3. Put updateStepList into updateStepOrder: updateStepOrder = [updateStepList]

Parameters **`updateStepList`** – List of pre-existing update steps without brackets.

Example Usage of Average Macros

```
averageField(testField1D,none,1,DT,8.e-11,0,0,0,NX,NY,NZ)
averageFieldWindow(testField1D,none,1,DUMP_PERIOD,DT,19.9,none,0,0,0,NX,NY,NZ)
averageField(testField,edge,3,DT,9.e-11,0,0,0,NX,NY,NZ)
averageFieldWindow(testField,edge,3,DUMP_PERIOD,DT,none,1.8e-10,0,0,0,NX,NY,NZ)



# Update steps
<UpdateStep step1>
        toDtFrac = 1.
```

(continues on next page)

(continued from previous page)

```
        updaters = [testUpdate]
        messageFields = [testField]
</UpdateStep>

<UpdateStep step2>
        toDtFrac = 1.
        updaters = [testUpdate1D]
        messageFields = [testField1D]
</UpdateStep>

$ updateStepList = step1 step2
$ updateStepList = appendAveSteps(updateStepList)
updateStepOrder = [updateStepList]

</MultiField>
```

basicEM.mac

This macro file can be imported to an input file with `$ import antennas`.

This macro file is available to all packages.

This collection of macros is meant to handle basic EM fields. It will establish the updaters and update steps to run the yee algorithm. The user must define the electric and magnetic fields themselves.

basicEM Macro

basicEM(*elecfield*, *magfield*, *nx*, *ny*, *nz*)
Define the fields, updaters and update steps for the regular update method on a Yee mesh. Designed to replace the old kind = yeeEmField. This requires the user to define the electric and magnetic fields so initial and boundary can be defined. This macro must set the update step order.

basicEM(*elecfield*, *magfield*, *gridboundary*, *nx*, *ny*, *nz*)
Version of macro for use with *GridBoundary*.

basicEM(*elecfield*, *magfield*, *nx*, *ny*, *nz*, *plx*, *ply*, *plz*, *pux*, *puy*, *puz*)
Version of macro for use with the *PmlRegion*.

Parameters

- **elecfield** – Name of the electric field.
- **magfield** – Name of the magnetic field.
- **nx** – Number of cells in the X-direction.
- **ny** – Number of cells in the Y-direction.
- **nz** – Number of cells in the Z-direction.
- **gridboundary** – The name of the grid boundary.
- **plx** – Number of pml cells in lower X.
- **ply** – Number of pml cells in lower Y.
- **pyz** – Number of pml cells in lower Z.
- **pux** – Number of pml cells in upper X.

- **puy** – Number of pml cells in upper Y.
- **puz** – Number of pml cells in upper Z.

Example of basicEM Macro

```
<EmField yeeEmField>
  kind = emMultiField

  <Field YeeElecField>
    numComponents = 3
    offset = edge
  </Field>

  <Field YeeMagField>
    numComponents = 3
    offset = face
  </Field>

  basicEM(YeeElecField,YeeMagField,sphere,NX,NY,NZ)
</EmField>
```

cnmacros.mac

This macro file can be imported to an input file with `$ import cnmacros`.

This macro file is available to all packages.

This collection of macros is used for using the Crank-Nicholson implicit solve for advancing the electromagnetic fields.

Crank-Nicholson Matrix Macro

cnMatrix(*name*, *sign*)

Matrix for a Crank-Nicholson solver for EM. This does not work well at large values of implicitness for the standard reasons when solving hyperbolic equations.

Parameters

- **name** – Name of the matrix.
- **sign** – 1 for the explicit or application part, -1 for the implicit or solve part.

Crank-Nicholson Epetra Updaters Macro

cnEpetraUpdaters()

Define the explicit and implicit updaters needed for the Crank-Nicolson update of the EM field. Assumes that the fields being updated are edgeElec and faceMag.

Crank-Nicholson EM Field Macro

`cnEmFieldAlgorithm`(*withNodalFields*)

This macros inserts the Crank-Nicholson update. It assumes that edgeElec and faceMag have been defined. It defines any fields needed for just this algorithm.

Parameters **`withNodalFields`** – Whether to create and update nodalE and nodalB. This is set to nonzero to be used with particles. With large implicitness, one may need to set maxcellxing for the particles.

deymittra.mac

This macro file can be imported to an input file with `$ import deymittra`.

This macro file is available to to the VSimMD, VSimEM or VSimPD packages.

This is a general macro file that contains multiple macros.

dmYee Macro

`dMYee`(*elecfield*, *magfield*, *gridbndry*, *nx*, *ny*, *nz*)

Defines the fields, updaters and update steps for the Dey-Mittra cut cell method on a Yee mesh. This macro also sets up interpolation to use the Yee field (edge for E and face for B). The dMYee macro is located in the file `deymittra.mac`.

Since this macro defines updates, be aware of the order in case boundary updaters are to be added:

- `step1.0`: the first half B update.
- `step2.0`: the full E update
- `step3.0`: the second half B update

The dMYee macro is compatible with the .. function:: applyPML macro‘ from `pml.mac`.

Dey Mittra Yee Macro

`deyMittraYee`(*elecfield*, *magfield*, *rhojfield*, *gridbndry*, *nx*, *ny*, *nz*)

Define the fields, updaters and update steps for the Dey-Mittra cut cell method on a Yee mesh. This macro also sets up interpolation to use the Yee field (edge for E and face for B) Since this macro defines updates the user be aware of the order in case they with to add boundary updaters.

step1 - the first half B update step2 - the full E update step3 - the second half B update

Parameters

- **`elecfield`** – Name of the electric field.
- **`magfield`** – Name of the magnetic field.
- **`rhojfield`** – Name of the charge density and current field.
- **`gridbndry`** – Name of the grid boundary.
- **`nx`** – Number of cells in x direction.
- **`ny`** – Number of cells in y direction.
- **`nz`** – Number of cells in z direction.

deyMittraYeeInterpol Macro

`deyMittraYeeInterpol` (*elecfield*, *magfield*, *rhojfield*, *gridbndry*, *nx*, *ny*, *nz*)

Defines the fields, updaters and update steps for the Dey-Mittra cut cell method on a Yee mesh, including the new constrained fields in the conductors needed for improved interpolation. The dMittraYeeInterpol macro is located in the file `deymittra.mac`.

Since this macro defines updates, be aware of the order in case boundary updaters are to be added:

`step1`: First half B update

`step2`: Full E update

`step2.5`: Constrained field updater

`step3`: Second half B update

dielectric.mac

This macro file can be imported to an input file with `$ import dielectric`.

This macro file is available to the VSimEM package.

This collection of macros can be used to define a dielectric with a user defined profile and permittivity, or lossy dielectric, with a user defined profile, permittivity and conductivity. These macros are only compatible with macro generated multifields (e.g. with the yee macro) There are two versions of this macro, dielectric and dielectricGPU. They have the same functionality, however dielectric is meant for simulations run on a CPU and dielectricGPU for simulations run on a GPU. GPU and CPU macros cannot be mixed

Add Dielectric Macro

`addDielectric` (*dielectricProfile*, *name*)

Adds a dielectric with the profile specified.

Parameters

- **`dielectricProfile`** – Spatial profile of relative permittivity of the dielectric as a function of x,y,z.
- **`name`** – Name of the dielectric.

Add Lossy Dielectric Macro

`addLossyDielectric` (*name*, *PERMITTIVITY*, *CONDUCTIVITY*, *lx*, *ly*, *lz*, *ux*, *uy*, *uz*)

This macro defines a lossy dielectric in the region defined.

Parameters

- **`name`** – Name of the lossy dielectric.
- **`PERMITIVITY`** – Permittivity of the dielectric.
- **`CONDUCTIVITY`** – Conductivity of the lossy dielectric.
- **`lx`** – Lower bound of the dielectric in the X direction.
- **`ly`** – Lower bound of the dielectric in the Y direction.
- **`lz`** – Lower bound of the dielectric in the Z direction.

- **ux** – Upper bound of the dielectric in the X direction.
- **uy** – Upper bound of the dielectric in the Y direction.
- **uz** – Upper bound of the dielectric in the Z direction.

Example usage:

```
addDielectric(boxDielectric, 1 + (EPSILON_R-1)*geoBoxP(x,y,z,5*DX,5*DY,5*DZ))
# The geoBoxP() evaluates to one where the point x,y,z is inside the box geometry
and 0 outside, so "1+" ensures vacuum relative permittivity outside the box..
```

```
$ EPSILON_SALT_WATER = 32.
$ SIGMA_SALT_WATER = 0.1
<function someGeometry(x,y,z)>
   H(x-x1)*H(x2-x)*H(y-y1)*H(y2-y)
</function>
<function epsilonrelative(x,y,z)>
   1+(EPSILON_SALT_WATER-1)*someGeometry(x,y,z)
</function>
<function sigma(x,y,z)>
   SIGMA_SALT_WATER*someGeometry(x,y,z)
</function>
addLossyDielectric(lossyDiel,epsilonrelative(x,y,z),sigma(x,y,z),XBGN,YBGN,ZBGN,XEND,
↪YEND,ZEND)
```

em.mac

This macro file can be imported to an input file with `$ import em`.

This macro file is available to all packages.

This collection of macros consists of some algorithms to help with electromagnetic simulations.

computeCurlDt Macro

computeCurlDt (*name*, *restorefield*, *readfield*, *writefield*, *multfac*)

A macro to compute the curl of a vector times dt for doing an EM update. Updates over the entire computational region.

This macro requires the variables NX, NY, NZ, DT, DX, DY and DZ.

Parameters

- **name** – Name of the updater to compute the curl from.
- **restoreField** – Name of field from which to restore updater time upon restart.
- **readField** – The field to copy from.
- **writefield** – The field to copy to.
- **multfac** – The factor by which to multiply dt before updating. This is ordinarily set to one.

constCurlDt Macro

constCurlDt (*name*, *restorefield*, *readfield*, *writefield*, *multfac*)

Compute the curl of a vector times a factor. The factor is set to the appropriate fraction of DT.

This macro requires the variables NX, NY, NZ, DT, DX, DY and DZ.

Parameters

- **name** – Name of the updater to compute the curl from.
- **restoreField** – Name of field from which to restore updater time upon restart.
- **readField** – The field to copy from.
- **writefield** – The field to copy to.
- **multfac** – The factor by which to multiply dt before updating. This is ordinarily set to one.

Yee Macro

Yee (*elecfield*, *magfield*, *nx*, *ny*, *nz*)

Define the fields, updaters and update steps for the regular update method on a Yee mesh. This macro also sets up interpolation to use the Yee field (edge for E and face for B). Since this macro defines updates the user be aware of the order in case they with to add boundary updaters.

step1.0 - the first half B update

step2.0 - the full E update

step3.0 - the second half B update

step4.1 - the nodal E update

step4.2 - the nodal B update

init_step1.1 - the nodal E restore

init_step1.2 - the nodal B restore

Parameters

- **elecfield** – Name of the electric field.
- **magfield** – Name of the magnetic field.
- **nx** – The number of cells in the X direction.
- **ny** – The number of cells in the Y direction.
- **nz** – The number of cells in the Z direction.

esGridBoundary.mac

This macro file can be imported to an input file with `$ import esGridBoundary`.

This macro file is available to the VSimEM, VSimSD, or VSimPD packages.

This collection of macros consists of tools for solving Dirichlet electrostatic boundaries and computing node-centered Laplacians.

nodeLaplacian Macro

nodeLaplacian(*name*, *gb*, *md*, *lbx*, *lby*, *lbz*, *ubx*, *uby*, *ubz*, *comp*, *fld*, *fcomp*, *ri*, *ci*, *dcoeff*, *xcoeff*, *ycoeff*, *zcoeff*)

Macro for node-centered Laplacian with grid boundaries.

Parameters

- **name** – The base name of the boundary condition.
- **gb** – The grid boundary name.
- **md** – The minimum dimension needed for this boundary condition.
- **lbx** – Lower bound in the x direction.
- **lby** – Lower bound in the y direction.
- **lbz** – Lower bound in the z direction.
- **ubx** – Upper bound in the x direction.
- **uby** – Upper bound in the y direction.
- **ubz** – Upper bound in the z direction.
- **comp** – The component (row of the matrix) being filled.
- **fld** – The name of the field used to fill the associated source (RHS) vector.
- **fcomp** – The component of the field.
- **ri** – The interiorosity of points whose corresponding rows in the matrix are to be filled.
- **ci** – The interiorosity of points whose corresponding columns in the matrix can be filled.
- **dcoeff** – Diagonal coefficient.
- **xcoeff** – Coefficient for x offset stencils.
- **ycoeff** – Coefficient for y offset stencils.
- **zcoeff** – Coefficient for z offset stencils.

coordProdNodeLaplacian Macro

coordProdNodeLaplacian(*name*, *gb*, *md*, *lbx*, *lby*, *lbz*, *ubx*, *uby*, *ubz*, *comp*, *fld*, *fcomp*, *ri*, *ci*, *scl*, *func*)

Macro for node-centered Laplacian with grid boundaries in a coordinate product (coordProd) grid.

Parameters

- **name** – The base name of the boundary condition.
- **gb** – The grid boundary name.
- **md** – The minimum dimension needed for this boundary condition.
- **lbx** – Lower bound in the x direction.
- **lby** – Lower bound in the y direction.
- **lbz** – Lower bound in the z direction.
- **ubx** – Upper bound in the x direction.
- **uby** – Upper bound in the y direction.

- **ubz** – Upper bound in the z direction.
- **comp** – The component (row of the matrix) being filled.
- **fld** – The name of the field used to fill the associated source (RHS) vector.
- **fcomp** – The component of the field.
- **ri** – The interiorosity of points whose corresponding rows in the matrix are to be filled.
- **ci** – The interiorosity of points whose corresponding columns in the matrix can be filled.
- **scl** – The scaling factor by which all matrix elements will be multiplied.
- **func** – The expression for the function defining the value.

nodeFuncLaplacian Macro

nodeFuncLaplacian (*name*, *gb*, *md*, *lbx*, *lby*, *lbz*, *ubx*, *uby*, *ubz*, *comp*, *fld*, *fcomp*, *ri*, *ci*, *xcoeff*, *ycoeff*, *zcoeff*, *func*)

Macro for node-centered Laplacian with grid boundaries using a STFunc for the coefficient.

Parameters

- **name** – The base name of the boundary condition.
- **gb** – The grid boundary name.
- **md** – The minimum dimension needed for this boundary condition.
- **lbx** – Lower bound in the x direction.
- **lby** – Lower bound in the y direction.
- **lbz** – Lower bound in the z direction.
- **ubx** – Upper bound in the x direction.
- **uby** – Upper bound in the y direction.
- **ubz** – Upper bound in the z direction.
- **comp** – The component (row of the matrix) being filled.
- **fld** – The name of the field used to fill the associated source (RHS) vector.
- **fcomp** – The component of the field.
- **ri** – The interiorosity of points whose corresponding rows in the matrix are to be filled.
- **ci** – The interiorosity of points whose corresponding columns in the matrix can be filled.
- **xcoeff** – Coefficient for x offset stencils.
- **ycoeff** – Coefficient for y offset stencils.
- **zcoeff** – Coefficient for z offset stencils.
- **func** – The expression for the STFunc defining the value of the coefficient.

gridBoundaryDirichletBC Macro

gridBoundaryDirichletBC (*name*, *gb*, *md*, *lbx*, *lby*, *lbz*, *ubx*, *uby*, *ubz*, *comp*, *exp*, *scl*)

Macro for setting Dirichlet boundary conditions on a grid boundary in electrostatic solves with node-centered fields.

Parameters

- **name** – The base name of the boundary condition.
- **gb** – The grid boundary name.
- **md** – The minimum dimension needed for this boundary condition.
- **lbx** – Lower bound in the x direction.
- **lby** – Lower bound in the y direction.
- **lbz** – Lower bound in the z direction.
- **ubx** – Upper bound in the x direction.
- **uby** – Upper bound in the y direction.
- **ubz** – Upper bound in the z direction.
- **comp** – The component (row of the matrix) being filled.
- **exp** – The expression for the function defining the value.
- **scl** – The scaling applied to all matrix and vector entries.

exteriorGridBoundaryDirichletBC Macro

exteriorGridBoundaryDirichletBC(*name*, *gb*, *md*, *lbx*, *lby*, *lbz*, *ubx*, *uby*, *ubz*, *comp*, *exp*, *scl*)

Macro for solving points exterior to a grid boundary defined Dirichlet boundary condition in electrostatic solves with node-centered fields.

Parameters

- **name** – The base name of the boundary condition.
- **gb** – The grid boundary name.
- **md** – The minimum dimension needed for this boundary condition.
- **lbx** – Lower bound in the x direction.
- **lby** – Lower bound in the y direction.
- **lbz** – Lower bound in the z direction.
- **ubx** – Upper bound in the x direction.
- **uby** – Upper bound in the y direction.
- **ubz** – Upper bound in the z direction.
- **comp** – The component (row of the matrix) being filled.
- **exp** – The expression for the function defining the value.
- **scl** – The scaling applied to all matrix and vector entries.

esSolveOpenBdry.mac

This macro file can be imported to an input file with `$ import esSolveOpenBdry`.

This macro file is available to all packages.

This macro file consists of tools for solving open electrostatic boundary conditions.

surfaceCharge Macro

surfaceCharge (*name*, *md*, *lbx*, *lby*, *lbz*, *ubx*, *uby*, *ubz*, *mcomp*, *rfld*, *rcomp*, *wfld*, *wcomp*, *coeff*, *offx*, *offy*, *offz*)

Macro to calculate the boundary screening charge.

Parameters

- **name** – The base name of the field updater.
- **md** – The minimum dimension needed for this boundary condition.
- **lbx** – Lower bound in the x direction.
- **lby** – Lower bound in the y direction.
- **lbz** – Lower bound in the z direction.
- **ubx** – Upper bound in the x direction.
- **uby** – Upper bound in the y direction.
- **ubz** – Upper bound in the z direction.
- **comp** – The component (row of the matrix) being filled.
- **rfld** – The name of the input field from which to calculate the screening charge.
- **rcomp** – The component of the read field.
- **wfld** – The name of the input field from which to write the screening charge.
- **wcomp** – The component of the write field.
- **coeff** – Diagonal coefficient.
- **offx** – X-offset of the input vector.
- **offy** – Y-offset of the input vector.
- **offz** – Z-offset of the input vector.

surfacePotential Macro

surfacePotential (*name*, *md*, *lbx*, *lby*, *lbz*, *ubx*, *uby*, *ubz*, *srcLbx*, *srcLby*, *srcLbz*, *srcUbx*, *srcUby*, *srcUbz*, *dn*, *rfld*, *rcomp*, *wfld*, *wcomp*, *rTFld*, *alpha*)

Macro to calculate the potential due to the boundary screening charge.

Parameters

- **name** – The base name of the field updater.
- **md** – The minimum dimension needed for this boundary condition.
- **lbx** – Lower bound in the x direction over which this is applied.
- **lby** – Lower bound in the y direction over which this is applied .
- **lbz** – Lower bound in the z direction over which this is applied.
- **ubx** – Upper bound in the x direction over which this is applied.
- **uby** – Upper bound in the y direction over which this is applied.
- **ubz** – Upper bound in the z direction over which this is applied.

- **srcLbx** – Lower bound in the x direction where the source is located.
- **srcLby** – Lower bound in the y direction where the source is located.
- **srcLbz** – Lower bound in the z direction where the source is located.
- **srcUbx** – Upper bound in the x direction where the source is located.
- **srcUby** – Upper bound in the y direction where the source is located.
- **srcUbz** – Upper bound in the z direction where the source is located.
- **dn** – Direction.
- **rfld** – The name of the input field from which to calculate the potential.
- **rcomp** – The component of the read field.
- **wfld** – The name of the field where to write the potential.
- **wcomp** – The component of the write field.
- **coeff** – Diagonal coefficient.
- **rTFld** – The field from which to restore time.
- **alpha** – The multiplying factor when calculating the potential.

linearSolver Macro

linearSolver (*name*)

Macro for the linear solver block. Uses a gmres solver.

Parameters **name** – The name of the LinearSolver.

esSolveDirichlet Macro

esSolveDirichlet (*NXLO*, *NYLO*, *NZLO*, *NXHI*, *NYHI*, *NZHI*, *srcFld*, *fld*, *comp*, *rstTimeFld*, *dcoeff*, *xcoeff*, *ycoeff*, *zcoeff*, *scl*)

Macro to solve with 0.0 at the boundary.

Parameters

- **NXLO** – Lower bound in the x direction (grid cell).
- **NYLO** – Lower bound in the y direction (grid cell).
- **NZLO** – Lower bound in the z direction (grid cell).
- **NXHI** – Upper bound in the x direction (grid cell).
- **NYHI** – Upper bound in the y direction (grid cell).
- **NZHI** – Upper bound in the z direction (grid cell).
- **srcFld** – Source field for the solve.
- **fld** – The field to write the results of the solve to.
- **comp** – The component of the read/write field.
- **rstTimeFld** – The field from which to restore time.
- **dcoeff** – Diagonal coefficient for the laplacian.
- **xcoeff** – X-coefficient for the laplacian.

- **`ycoeff`** – Y-coefficient for the laplacian.
- **`zcoeff`** – Z-coefficient for the laplacian.
- **`scl`** – The scaling applied to all matrix and vector entries for BCs.

esSolveDirichletFromField Macro

`esSolveDirichletFromField`(*NXLO*, *NYLO*, *NZLO*, *NXHI*, *NYHI*, *NZHI*, *srcFld*, *fld*, *comp*, *rstTimeFld*, *dcoeff*, *xcoeff*, *ycoeff*, *zcoeff*, *scl*)

Macro to solve with boundary conditions set from the result field.

Parameters

- **`NXLO`** – Lower bound in the x direction (grid cell).
- **`NYLO`** – Lower bound in the y direction (grid cell).
- **`NZLO`** – Lower bound in the z direction (grid cell).
- **`NXHI`** – Upper bound in the x direction (grid cell).
- **`NYHI`** – Upper bound in the y direction (grid cell).
- **`NZHI`** – Upper bound in the z direction (grid cell).
- **`srcFld`** – Source field for the solve.
- **`fld`** – The field to write the results of the solve to.
- **`comp`** – The component of the read/write field.
- **`rstTimeFld`** – The field from which to restore time.
- **`dcoeff`** – Diagonal coefficient for the laplacian.
- **`xcoeff`** – X-coefficient for the laplacian.
- **`ycoeff`** – Y-coefficient for the laplacian.
- **`zcoeff`** – Z-coefficient for the laplacian.
- **`scl`** – The scaling applied to all matrix and vector entries for BCs.

farFields.mac

This macro file can be imported to an input file with `$ import farFields`.

This macro file is available to the VSimEM package.

This macro file enables the calculation of the electromagnetic fields far from the computational domain using Green's functions.

There are two versions of this macro, `addFarFields` and `addFarFieldsGPU`. They have the same functionality, however they are meant to be used for simulations run on a CPU and GPU, respectively.

`addFarFields`()

This macro defines the fields, updaters and update steps to be able to calculate far fields. It is dependent on macro generated multifields, and is incompatible with user generated multifields. The `addFarFields` macro is located in the file `farFields.mac`.

It also is also dependent on the antennas macro, to call a history that will record the far field data. This history is titled `addFarFieldHistory` and is located in the file `antennas.mac`

For more information, you can visit the *Add Far Field History Macro* page where the parameters and output are listed.

filters.mac

This macro file can be imported to an input file with `$ import filters`.

This macro file is available to the VSimMD and VSimPA packages.

This macro file provides some useful filters for diagnostic purposes.

`filter` (*elecfield*, *workingmagfield*, *boundaryname*, *maxeigenval*, *filterval*, *nxl*, *nyl*, *nzl*, *nxu*, *nyu*, *nzu*, *updatestep*)

Parameters

- **`elecfield`** – Name of the dynamic electric field.
- **`workingmagfield`** – Name of a working magnetic field, e.g, NOT the dynamic magnetic field
- **`boundaryname`** – Name of the electromagnetic GridBoundary, or 0, if none.
- **`maxeigneval`** – Maximum Maxwell eigenvalue, e.g., = 4 * c^2 * dt^2 * (1/dx^2 + 1/dy^2 + 1/dz^2) / (dmfrac^2).
- **`filterval`** – Filter value, e.g., = 1, for a single stage filter, see macro notes for more.
- **`nxl`** – Lower bound in X.
- **`nyl`** – Lower bound in Y.
- **`nzl`** – Lower bound in Z.
- **`nxu`** – Upper bound in X.
- **`nyu`** – Upper bound in Y.
- **`nzu`** – Upper bound in Z.
- **`updatestep`** – Update step for filter sequence.

This macro applies an EM noise filter, especially for Numerical Cerenkov.

Several filters in sequence can be used. The following are suggested:

filterval = 1 … one stage filter, medium damping, P(x) = 1-x

filterval = 1, 1 … two stage filter, strong damping, P(x) = (1-x)*(1-x)

filterval = -1, 1 … two stage filter, weaker damping, P(x) = (1-x*x)

filterval = 1, -2, 1 … three stage filter, weaker damping at long wavelengths, strong damping at short wavelengths, P(x) = 1-3*x*x+2*x*x*x

In general, if x = eigenvalue/maxeigenval, then that mode will be damped according to the value of the polynomial,(1-filterval_1*x) *(1-filterval_2*x)* with the polynomial shown in the three examples above.

Example of use:

```
<Field WorkMagField>
  numComponents = 3
  offset = face
</Field>
```

(continues on next page)

(continued from previous page)

```
<UpdateStep NormalAmpereStep1.0>
  toDtFrac      = 1.0
  updaters      = [yeeAmpere]
  messageFields = [YeeElecField]
</UpdateStep>

$ MAX_EIGENVAL = SPD_LIGHT*SPD_LIGHT*DT*DT*(1./DX**2 + 1./DY**2)/(CFL_FACTOR*CFL_
↪FACTOR)
filter(YeeElecField, WorkMagField, myBoundary, MAX_EIGENVAL,  1.0, 0, 0, 0, NX, NY,␣
↪NZ, 2.0)
filter(YeeElecField, WorkMagField, myBoundary, MAX_EIGENVAL, -2.0, 0, 0, 0, NX, NY,␣
↪NZ, 2.2)
filter(YeeElecField, WorkMagField, myBoundary, MAX_EIGENVAL,  1.0, 0, 0, 0, NX, NY,␣
↪NZ, 2.4)

<UpdateStep NormalFaradayStep3.0>
  toDtFrac      = 1.0
  updaters      = [yeeFaraday deyMittraFaraday]
  messageFields = [YeeMagField]
</UpdateStep>
```

If there are unusual boundary conditions applied to the electric field interior to the domain, they should probably be re-imposed after each filter step. This macro is encrypted and is available with the VSimMD License.

geometry.mac

This macro file can be imported to an input file with `$ import geometry`.

This macro file is available to the VSimEM, VSimMD, and VSimPD packages.

This macro file provides some useful functions for creating geometries for simulations. There are two versions of this macro, one, geometry, is for use when doing computations on a CPU, the other, geometryGPU, is used for computations on a GPU. They retain the exact same functionality, however GPU and CPU macros cannot be mixed. The GPU version only works with the VSimEM package.

Geometry Macro Introduction

A very powerful tool exists in VSim to create complex shapes for simulation. Simply import the geometry.mac file and a large number of primitive shapes are available to combine into complex shapes. This file also contains macros to import .stl and Python-defined shapes.

Basic Filling/Voiding

The default way to begin forming geometry is to void the universe and then fill it with shapes one at a time. To ensure a void starting environment use the command, resetGeoToVoid().

Alternatively you can begin forming geometry by filling the entire region with metal and then void shapes, that is, carve out hollow spaces. The filled starting environment is created with resetGeoToFill(universe). A name, in this case *universe*, must be given for a filled environment.

Filling and Voiding operations act on the previously defined operations. This means that the order of shape filling and voiding is important. This is expanded upon further in *Advanced Filling and Voiding* and is demonstrated in the ridgedWaveguide macro, below.

Grid Boundaries

To finalize the geometry as a grid boundary, the command

```
saveGeoToGridBoundary(gridBoundaryName,CFL_FACTOR)
```

must be used. If a reusable complex shape is desired, the shape may be constructed as a macro, with the argument of the macro listed as the shape name. An example is given below,

```
<macro ridgedWaveguide()>
  voidGeoExpression(waveguideBox,geoBoxP,x,y,z,.5,.4,.15))
  fillGeoExpression(ridge_1,geoBoxP(x-.25,y    ,z,.1,.15,.15))
  fillGeoExpression(ridge_2,geoBoxP(x-.25,y-.25,z,.1,.15,.15))
  fillGeoExpression(ridge_3,geoBoxP(x    ,y-.25,z,.1,.15,.15))
  fillGeoExpression(ridge_4,geoBoxP(x    ,y    ,z,.1,.15,.15))
</macro>
ridgedWaveguide()
saveGeoToGridBoundary(waveguide,CFL_FACTOR)
```

There is also a version of this macro that allows the saving of the volumes:

```
saveGeoToGridBoundary(gridBoundaryName,CFL_FACTOR,calcVol)
```

An example is

```
<macro ridgedWaveguide()>
  voidGeoExpression(waveguideBox,geoBoxP,x,y,z,.5,.4,.15))
  fillGeoExpression(ridge_1,geoBoxP(x-.25,y    ,z,.1,.15,.15))
  fillGeoExpression(ridge_2,geoBoxP(x-.25,y-.25,z,.1,.15,.15))
  fillGeoExpression(ridge_3,geoBoxP(x    ,y-.25,z,.1,.15,.15))
  fillGeoExpression(ridge_4,geoBoxP(x    ,y    ,z,.1,.15,.15))
</macro>
ridgedWaveguide()
saveGeoToGridBoundary(waveguide,CFL_FACTOR,true)
```

Shape Primitives

geoBoxP (*x*, *y*, *z*, *LXO*, *LYO*, *LZO*)

Parameters

- **LXO** – Length of box in X (m).
- **LXO** – Length of box in Y (m).
- **LXO** – Length of box in Z (m).

geoQuadrilateralSlabXP (*x*, *y*, *z*, *LXO*, *ya*, *za*, *yb*, *zb*, *yc*, *zc*, *yd*, *zd*)

Parameters

- **LXO** – Length of box in X (m).
- **ya** – Y-coordinate of first corner in quadrilateral, measured clockwise (m).
- **za** – Z-coordinate of first corner in quadrilateral, measured clockwise (m).
- **yb** – Y-coordinate of second corner in quadrilateral, measured clockwise (m).
- **zb** – Z-coordinate of second corner in quadrilateral, measured clockwise (m).

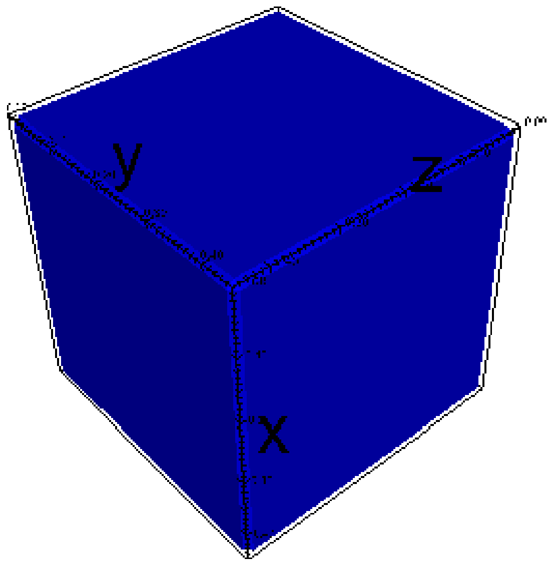


Fig. 3.5: Box (x,y,z,LXO,LYO,LZO)

- **yc** – Y-coordinate of third corner in quadrilateral, measured clockwise (m).
- **zc** – Z-coordinate of third corner in quadrilateral, measured clockwise (m).
- **yd** – Y-coordinate of fourth corner in quadrilateral, measured clockwise (m).
- **zd** – Z-coordinate of fourth corner in quadrilateral, measured clockwise (m).

geoTriangleSlabXP (*x*, *y*, *z*, *LXO*, *ya*, *za*, *yb*, *zb*, *yc*, *zc*)

Parameters

- **LXO** – Length of box in X (m).
- **ya** – Y-coordinate of first corner in triangle, measured clockwise (m).
- **za** – Z-coordinate of first corner in triangle, measured clockwise (m).
- **yb** – Y-coordinate of second corner in triangle, measured clockwise (m).
- **zb** – Z-coordinate of second corner in triangle, measured clockwise (m).
- **yc** – Y-coordinate of third corner in triangle, measured clockwise (m).
- **zc** – Z-coordinate of third corner in triangle, measured clockwise (m).

geoBiParabolicSlabXP (*x*, *y*, *z*, *LXO*, *yVertexIn*, *yVertexOut*, *halfHeightZIn*, *halfHeightZOut*)

Parameters

- **LXO** – Length of biparabolic slab in X (m).
- **yVertexIn** – Y-coordinate of the inner vertex (m).
- **yVertexOut** – Y-coordinate of the outer vertex (m).
- **halfHeightZIn** – Z-coordinate of the inner height(m).
- **halfHeightZOut** – Z-coordinate of the outer height (m).

geoCylinderXP (*x*, *y*, *z*, *RADIUS*, *LXO*)

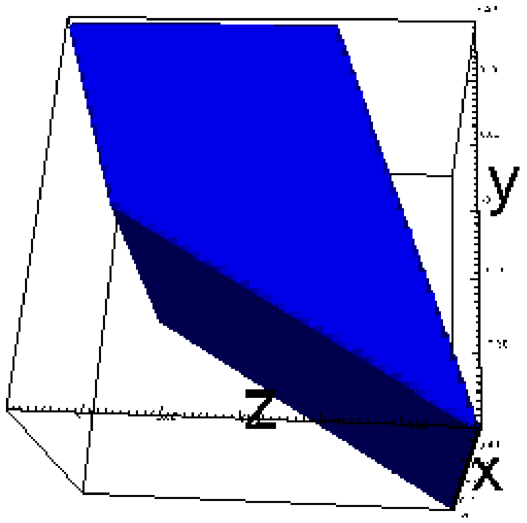


Fig. 3.6: Quadrilateral Slab (x,y,z,LXO,ya,za,yb,zb,yc,zc,yd,zd)

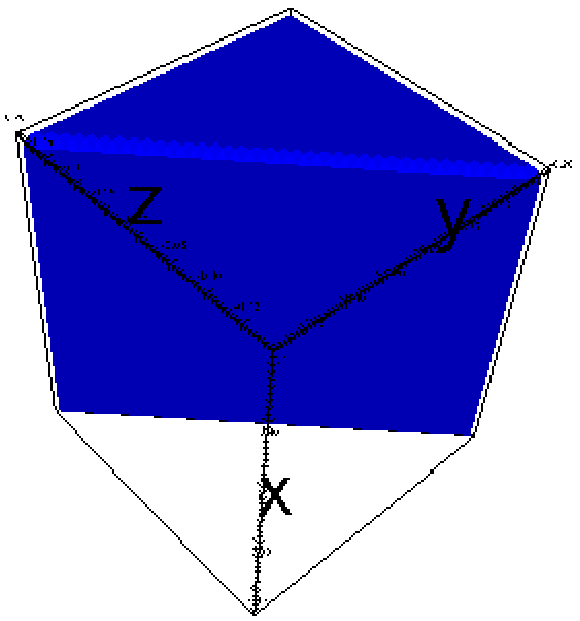


Fig. 3.7: Triangle Slab (x,y,z,LXO,ya,za,yb,zb,yc,zc)

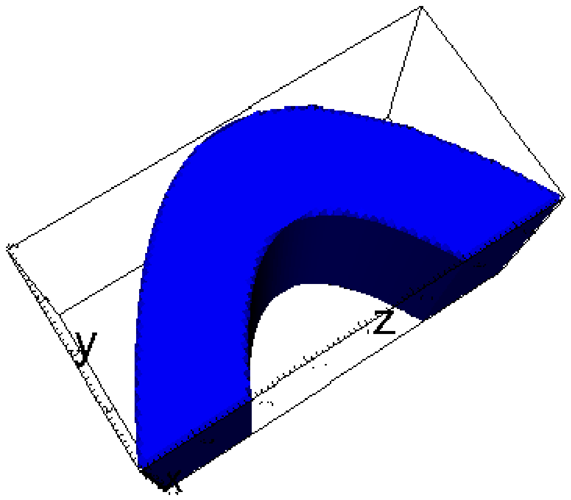


Fig. 3.8: BiParabolic Slab (x,y,z,LXO,yVertexIn,yVertexOut,halfHeightZIn,halfHeightZOut)

Parameters

- **`RADIUS`** – Radius of the cylinder (m).
- **`LXO`** – Length of cylinder in X (axial direction) (m).

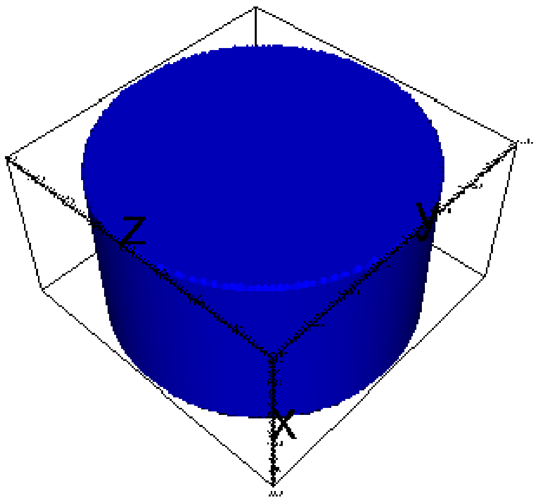


Fig. 3.9: Cylinder (x,y,z,RADIUS,LXO)

`geoPipeXP` (*x*, *y*, *z*, *INNER_RADIUS*, *RADIUS*, *LXO*)

Parameters

- **`INNER_RADIUS`** – Inner radius of the pipe (m).
- **`RADIUS`** – Outer radius of the pipe (m).
- **`LXO`** – Length of pipe in X (axial direction) (m).

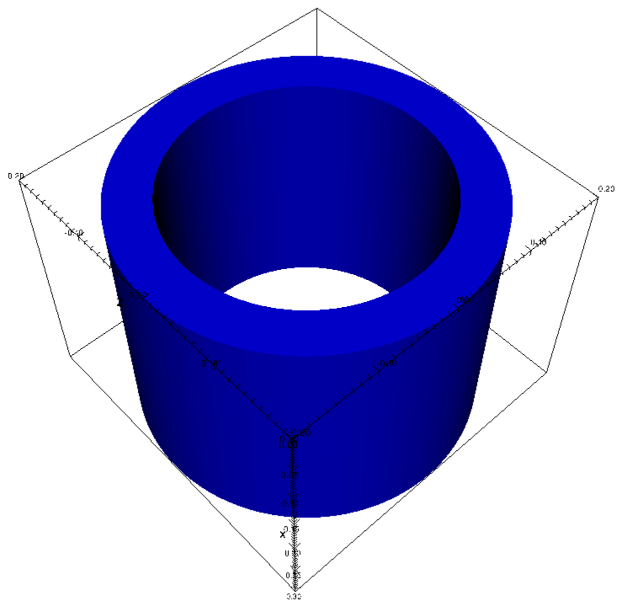


Fig. 3.10: Pipe (x,y,z,INNER_RADIUS,RADIUS,LXO)

geoConeXP (*x*, *y*, *z*, *MINOR_RADIUS*, *MAJOR_RADIUS*, *LXO*)

Parameters

- **INNER_RADIUS** – Minor radius of the pipe (m).
- **MAJOR_RADIUS** – Major radius of the cone (m).
- **LXO** – Length of cone in X (axial direction) (m).

Spherical Shape Primitives

geoHemiSphereXP (*x*, *y*, *z*, *RADIUS*)

Parameters **RADIUS** – Radius of the hemisphere (m).

geoSphere (*x*, *y*, *z*, *RADIUS*)

Parameters **RADIUS** – Radius of the sphere (m).

geoTorusX (*x*, *y*, *z*, *MINOR_RADIUS*, *MAJOR_RADIUS*)

Parameters

- **MINOR_RADIUS** – Radius of the physical torus (m).
- **MAJOR_RADIUS** – Distance between the origin and the middle of the torus (m).

geoParaboloidXP (*x*, *y*, *z*, *xVertex*, *RADIUS*)

geoRndCylinderXP (*x*, *y*, *z*, *RADIUS*, *LXO*, *RND_RADIUS*)

Parameters

- **RADIUS** – Radius of the cylinder (m).
- **LXO** – Length of the cylinder, axial direction is x by default (m).
- **RND_RADIUS** – Radius of the rounded edge of the cylinder (m).

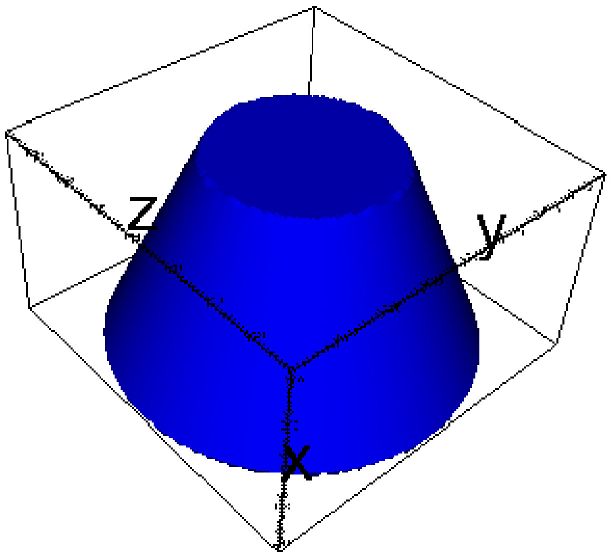


Fig. 3.11: Cone (x,y,z,MINOR_RADIUS,MAJOR_RADIUS,LXO)

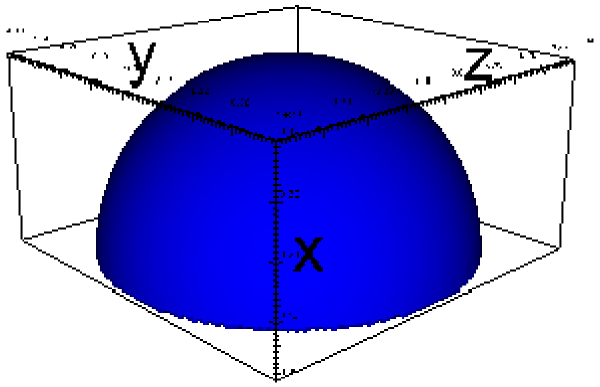


Fig. 3.12: Hemisphere (x,y,z,RADIUS)

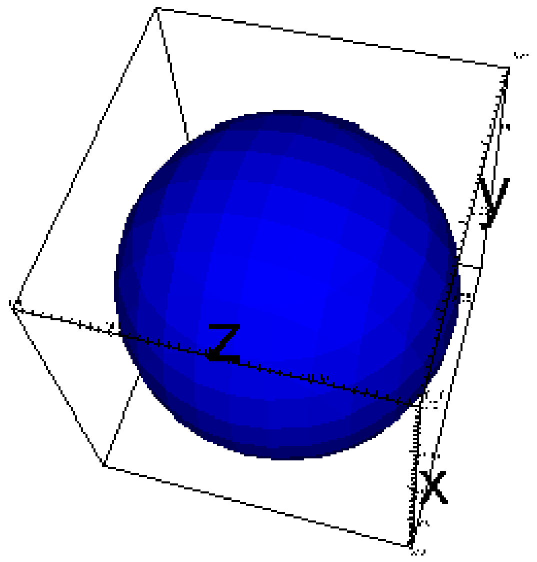


Fig. 3.13: Sphere (x,y,z,RADIUS)

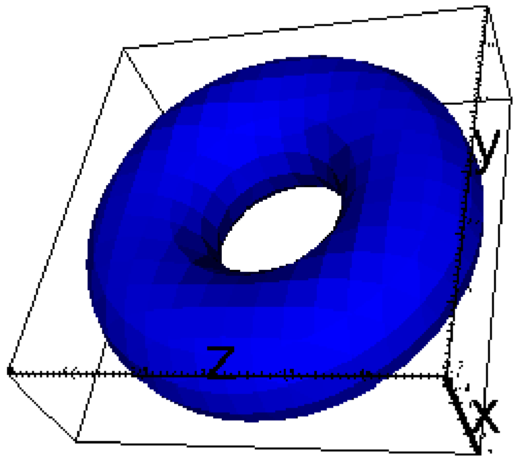


Fig. 3.14: Torus (x,y,z,MINOR_RADIUS,MAJOR_RADIUS)

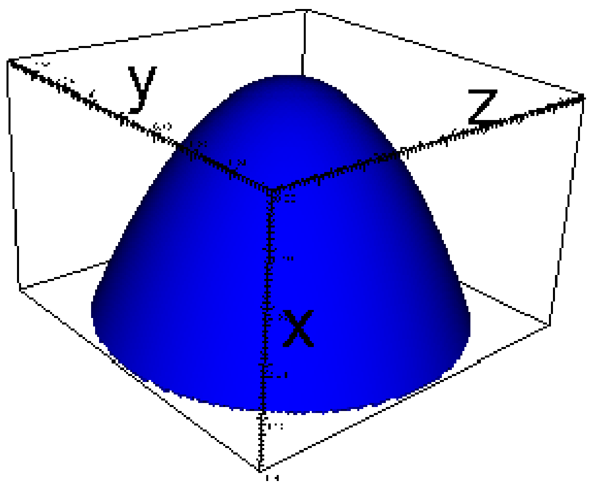


Fig. 3.15: Paraboloid (x,y,z,xVertex,RADIUS)

geoRndRectangleSlabXP (*x*, *y*, *z*, *LZO*, *LYO*, *LZO*, *RND_RADIUS*)

Parameters

- **LXO** – Length of box in X (m).
- **LXO** – Length of box in Y (m).
- **LXO** – Length of box in Z (m).
- **RND_RADIUS** – Radius of the rounded edge of the box (m).

geoEllipsoid (*x*, *y*, *z*, *A*, *B*, *C*)

Parameters

- **A** – Eccentricity of the ellipsoid in X.
- **B** – Eccentricity of the ellipsoid in Y.
- **C** – Eccentricity of the ellipsoid in Z.

geoHemiEllipsoidXP (*x*, *y*, *z*, *A*, *B*, *C*)

Parameters

- **A** – Eccentricity of the hemiellipsoid in X.
- **B** – Eccentricity of the hemiellipsoid in Y.
- **C** – Eccentricity of the hemiellipsoid in Z.

geoEllipticalCylinderXP (*x*, *y*, *z*, *B*, *C*, *LXO*)

Parameters

- **B** – Eccentricity of the elliptical cylinder in Y.
- **C** – Eccentricity of the elliptical cylinder in Z.

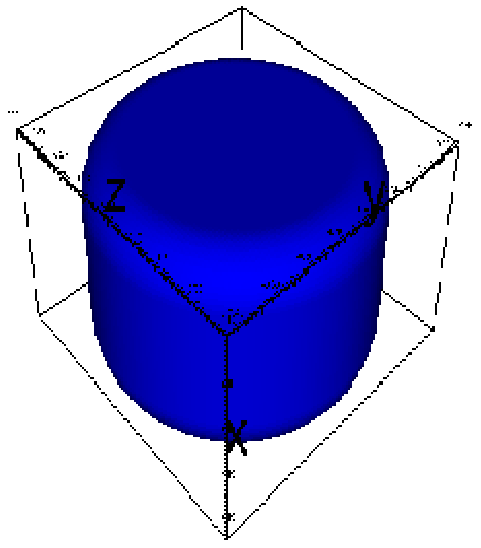


Fig. 3.16: Rounded Cylinder (x,y,z,RADIUS,LXO,RND_RADIUS)

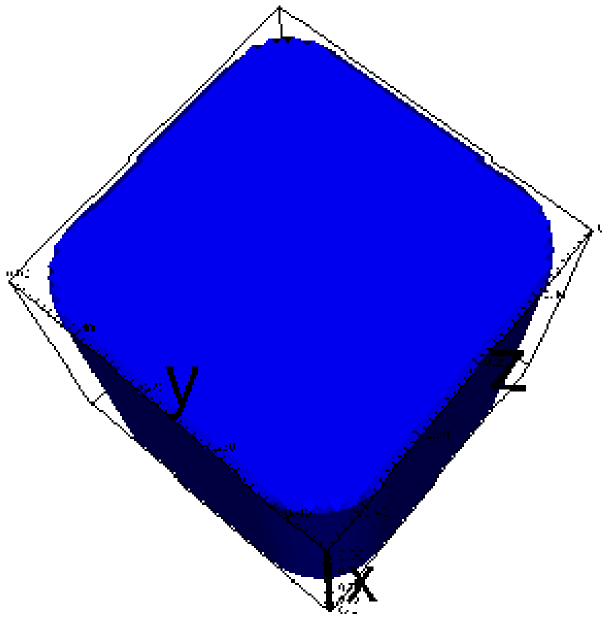


Fig. 3.17: Rounded Rectangular Slab (x,y,z,LZO,LYO,LZO,RND_RADIUS)

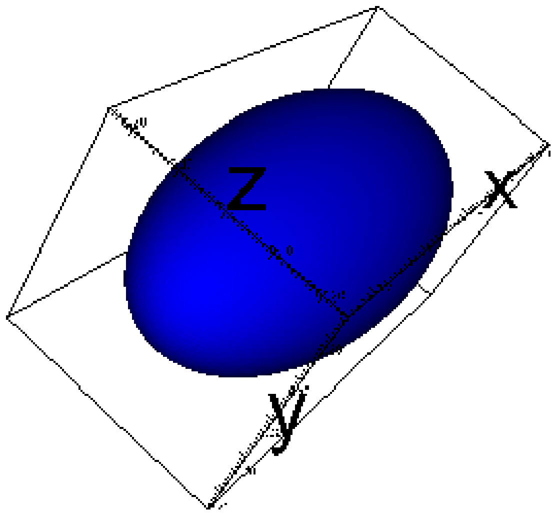


Fig. 3.18: Ellipsoid (x,y,z,A,B,C)

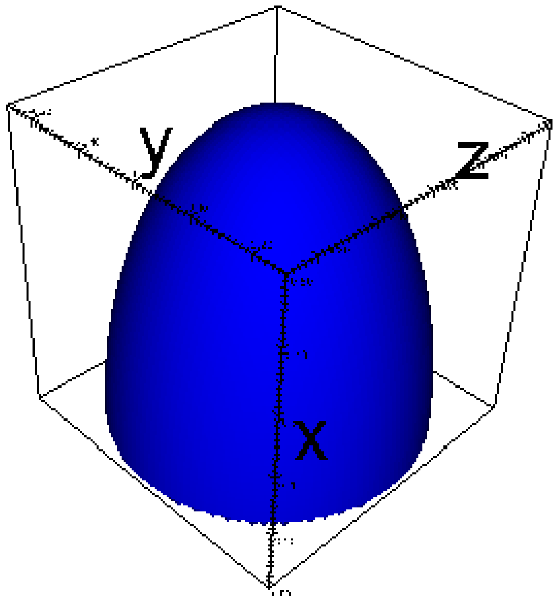


Fig. 3.19: Hemiellipsoid (x,y,z,A,B,C)

- **LXO** – Length of the elliptical cylinder in the X direction.

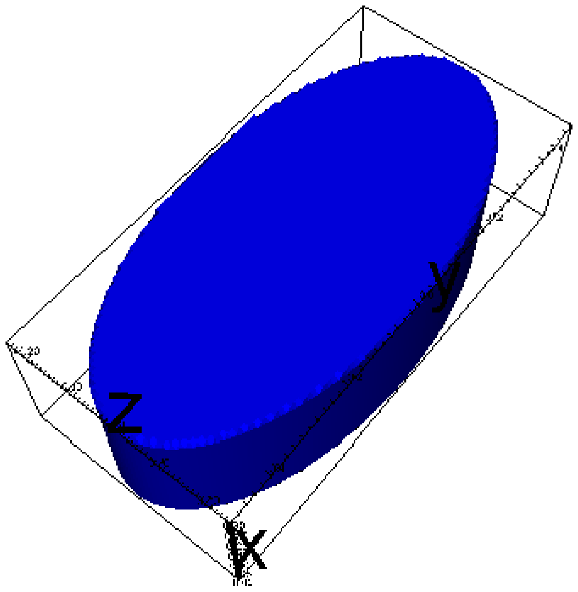


Fig. 3.20: Elliptical Cylinder (x,y,z,B,C,LXO)

geoEllipticalConeXP (*x*, *y*, *z*, *B*, *C*, *lowerScale*, *upperScale*, *LXO*)

Parameters

- **B** – Eccentricity of the elliptical cone in Y.
- **C** – Eccentricity of the elliptical cone in Z.
- **lowerScale** – Scale of the bottom of the cone.
- **upperScale** – Scale of the top of the cone.
- **LXO** – Length of the elliptical cone in the X direction.

In general, shapes in the geometry macro are created in the Y-Z plane, and then extruded through the X axis. For example a cylinder is circular in the Y-Z plane, and has its length in the X direction.

Moving Shapes

In general, the primitive shapes place points of rotation symmetry, such as the center of a circle at the origin. Rectilinear shapes begin at the origin and extend in the X-direction.

To move a shape in the X,Y and Z direction, the arguments of the function must be adjusted. For example to move a box 3 meters in the positive X direction the input would be geoBox(x-3,y,z,5,5,5). It would remain 5 meters long in the X direction. Adding instead of subtracting will cause a shift in the negative X direction.

Rotating Shapes

It is common to need to rotate a shape's orientation. This is most common in shapes such as cylinders, or tori. All shapes in the geometry macro have their axial direction in the X axis. To set the Y axis as the axial direction, simply exchange the variables Y and X. For example geoCylinder(y,x,z,.3,.8)

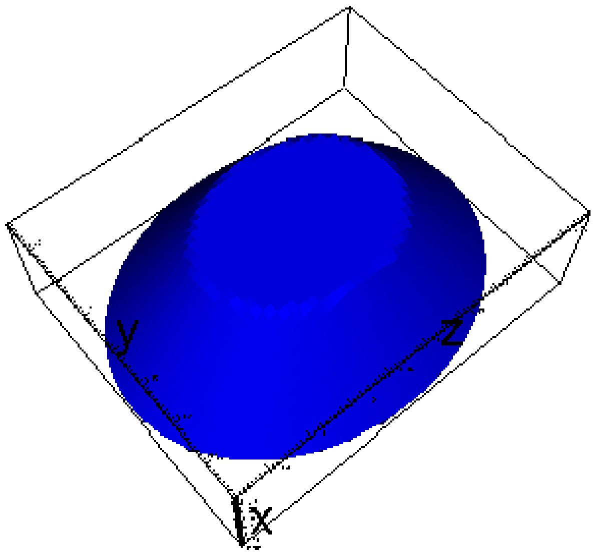


Fig. 3.21: Elliptical Cone (x,y,z,B,C,lowerScale,upperScale,LXO)

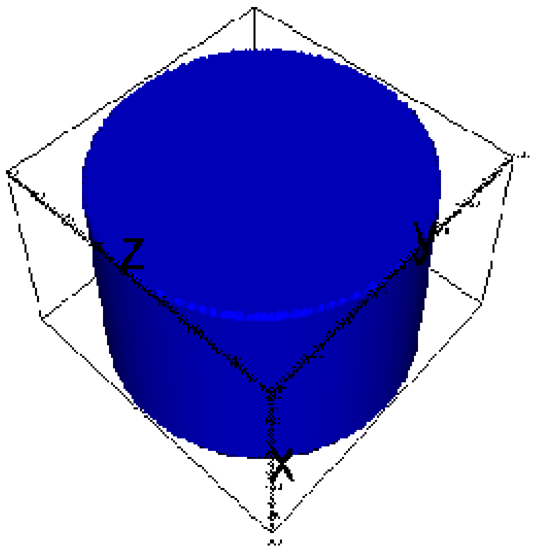


Fig. 3.22: Default Rotation Cylinder

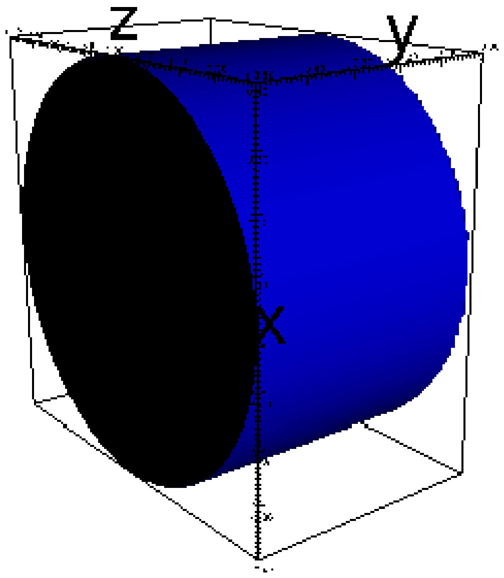


Fig. 3.23: Rotated Cylinder

Advanced Filling and Voiding

By using a sequence of fills and voids, complex shapes can be quickly created and tested. Here we illustrate this with the ridged waveguide. We start with a filled environment. Next we void out a rectangular waveguide, and then using fills to create the ridges. This is demonstrated in the example file.

Importing Objects From STL Files

The geometry macro can import CAD generated geometry directly from an STL file. (STEP files may also be converted to STL through the TXGML setup interface.) This is handled in the same manner as creating a rudimentary shape.

`voidGeoCad`(*objectName*, *stlFileName*, *shapeComplement*, *scaling*, *translation*)

`fillGeoCad`(*objectName*, *stlFileName*, *shapeComplement*, *scaling*, *translation*)

`fillGeoFastCAD`(*objectName*, *stlFileName*, *dmfrac*, *shapeComplement*, *scaling*)

Parameters

- **`objectName`** – Name of the geometry object.
- **`stlFileName`** – Name of the STL file to be imported.
- **`shapeComplement`** – Binary True or False. Reverses the meaning of inside/outside. Normally set to False to insert the shape into the simulation domain. Set to True if you want to import the interior of the shape and remove everything exterior to it from your domain.
- **`scaling`** – Scaling factor to be applied to STL shape. Most commonly used to adjust units of STL file to meters.
- **`translation`** – Vector describing any translations applied to STL shape; is defined in terms of the units of the STL file.

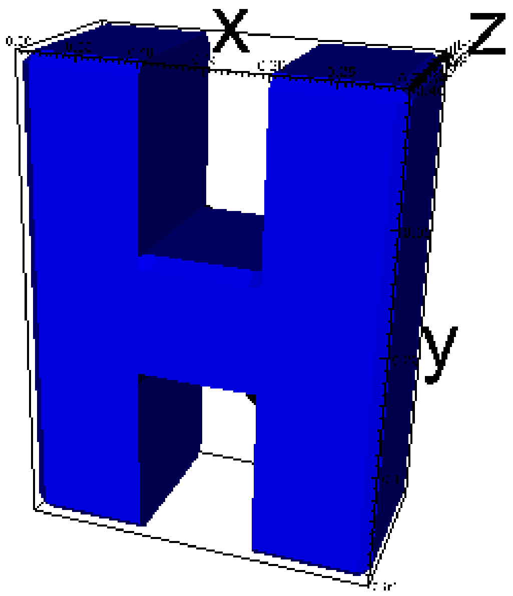


Fig. 3.24: Ridged waveguide made of "fill" and "void" regions

CAD shapes can be used together with primitive shapes. When using the shapeComplement feature, it is important that the .stl file have a single solid, as is the common usage, and that the solid be connected. If there are more than one disconnected shapes, then multiple .stl files should be used.

In VSim 7, the *fillGeoFastCAD* macro makes use of the **VMesh** fast meshing capabilities and is an alternative to *fillGeoCad*.

Note: Unlike *fillGeoCad* or any other geometry declaration, a *fillGeoFastCAD* macro call ought not be followed up by a *saveGeoToGridBoundary* call.

param objectName Name of the geometry object.

param stlFileName Name of the STL file to be imported.

param dmfrac Value of Dey-Mittra fraction.

param shapeComplement Binary: True or False. Reverses the meaning of inside/outside. Normally set to False to insert the shape into the simulation domain. Set to True if you want to import the interior of the shape and remove everything exterior to it from your domain.

param scaling Scaling factor to be applied to the STL shape. Most commonly used to adjust units of STL file to meters.

Importing a Python Script

The geometry macro contains the ability to import a Python-defined shape into the simulation environment. This is done with the command shown below.

`voidGeoPython` (*objectName*, *functionName*)

`fillGeoPython` (*objectName*, *functionName*)

Parameters

- **`objectName`** – Name of the geometry object.
- **`functionName`** – Name of the function within the Python file.

The Python file itself must have the same name as the input file name (excluding the file extension), and contain the specified function. The function should evaluate to unity inside the shape and zero outside the shape.

Shape Primitive Creation

Sometimes it may be desired to create a new shape primitive. This technique is most useful if creating an input file with a repetitive, complex geometry. The basis of all shape primitives is the Heaviside function.

In the code block given below, the macro hollowSphere specifies a sphere in which the sphere only extends between a specified inner and outer radius.

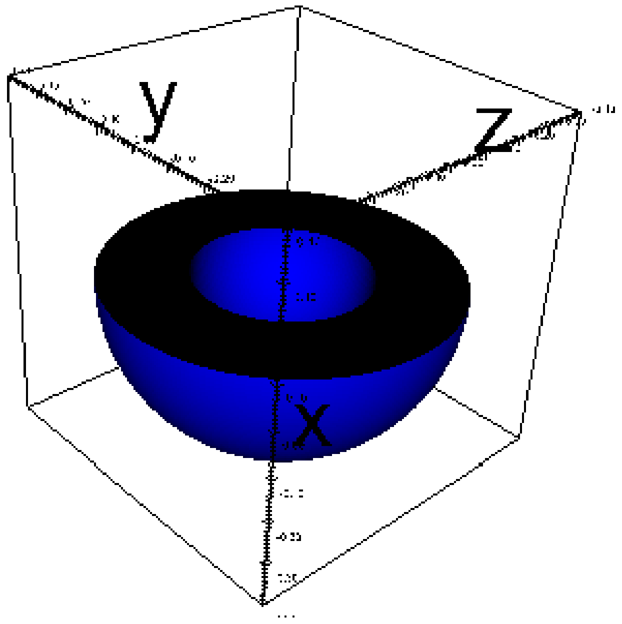


Fig. 3.25: hollowSphere (x,y,z,rInner,rOuter)

```
<macro hollowSphere(x,y,z,rInner,rOuter)>
 ( H(rOuter^2 - x^2 - y^2 - z^2)-H(rInner^2 - x^2 - y^2 - z^2) )
</macro>
```

Tips/Tricks

When building complex shapes it tends to be easier to initially center the shapes around the origin, and then move them into place. This way mirroring or rotating a shape around any of the axis is easy and understandable. It is also recommended when starting out to create a large domain to create shapes in. This way if an error in the translation or size is made it is possible that it will still appear in the domain, making the error easier to spot. Increasing the number of cells helps sharpen corners and smooths rounded shapes.

hist.mac

This macro file can be imported to an input file with `$ import history`.

This macro file is available to all packages.

The hist macro file contains many of the histories used in simulations, for a detailed description of each history and how to call it please see *VSim Reference*. This macro can be imported to an input file with $ import hist. This macro is unencrypted and is available with a VSimBase license.

saveHistories Macro

`saveHistories`()

This macro saves the histories generated via macros.

addFieldAtCoordBlock Macro

`addFieldAtCoordHist`(*name*, *fieldName*, *comp*, *px*, *py*, *pz*)

This history records the value of a field in the components specified at the point specified. More information is available at *fieldAtCoords*

Parameters

- **`name`** – The name of the history.
- **`fieldName`** – The name of the field used in the hstory.
- **`comp`** – Components to be used in the history.
- **`px`** – X-coordinate of the history.
- **`py`** – Y-coordinate of the history.
- **`pz`** – Z-coordinate of the history.

addPoyntingHist Macro

`addPoyntingHist`(*name*, *elecfield*, *magfield*, *lx*, *ly*, *lz*, *ux*, *uy*, *uz*)

Parameters

- **`name`** – The name of the history.
- **`elecfield`** – The name of the electric field used in the hstory.
- **`magfield`** – The name of the magnetic field used in the hstory.
- **`lx`** – Lower X-coordinate of the history.
- **`ly`** – Lower Y-coordinate of the history.
- **`lz`** – Lower Z-coordinate of the history.
- **`ux`** – Upper X-coordinate of the history.
- **`uy`** – Upper Y-coordinate of the history.
- **`uz`** – Upper Z-coordinate of the history.

addFieldSqrVolOpHist Macro

addFieldSqrVolOpHist (*histName*, *scope*, *fieldName*, *operation*, *multiplyByDV*, *recordFieldValue*, *recordPosition*, *lbs*, *ubs*, *gridName*, *gridBndry*, *inside*, *order*, *applyStep*, *multFactor*, *label*)

A macro to add a history that calculates the volume integral of a field-squared within a given volume V, i.e., the integral over V of ||F||^2 dV (times multFactor). N.B. this history may take a relatively long time.

Parameters

- **hisName** – The name of the history.
- **scope** – The name of the MultiField or EmField containing the field (can be the empty string "", and the field name can then be qualified with the MultiField or EmField).
- **fieldName** – The name of the field used in the hstory.
- **operation** – sumOverCells, maxOverCells, or minOverCells.
- **multiplyByDV** – Whether to multiply by dV, either 0 or 1 (false or true).
- **recordFieldValue** – (for min or max, not sum): if 1, the first components will be the value of the field (with dA if requested, without multFactor) at min/max.
- **recordPosition** – If 1, the coordinates of the min/max field will be added (after field values if desired).
- **lbs** – The lower bounds (can be "", for whole sim domain).
- **ubs** – The upper bounds (can be "", for whole sim domain).
- **gridName** – The name of the grid.
- **gridBndry** – The name of the grid boundary, can be "" for no boundary.
- **inside** – If 1, integrate inside gridBndry, if 0, outside.
- **order** – The order of interpolation to use (N.B. this volume integration has second-order error, so using order > 1 is probably not really useful).
- **applyStep** – The integer time step at which to calculate the history.
- **multFactor** – A factor by which to multiply the field energy.
- **label** – A string-vec label for history file (can be "" for generic label).

addTwoFieldSqrVolOpHist Macro

addTwoFieldSqrVolOpHist (*histName*, *scope*, *fieldName1*, *fieldName2*, *operation*, *multiplyByDV*, *recordFieldValue*, *recordPosition*, *lbs*, *ubs*, *gridName*, *gridBndry*, *inside*, *order*, *applyStep*, *multFactor*, *multFactor2*, *label*)

A macro to add a history that calculates the desired operation (min, max, sum) of a field-squared within a given volume V, i.e., the integral over V of a(||F||^2 + b||G||^2) dV (where a = multFactor, b = multFactor2, and F and G are the fields)

Parameters

- **histName** – The name of the history.
- **scope** – The name of the MultiField or EmField containing the field (can be the empty string "", and the field name can then be qualified with the MultiField or EmField).
- **fieldName1** – The name of the first field used in the hstory.

- **`fieldName2`** – The name of the second field used in the hstory.
- **`operation`** – sumOverCells, maxOverCells, or minOverCells.
- **`multiplyByDV`** – Whether to multiply by dV, either 0 or 1 (false or true).
- **`recordFieldValue`** – (for min or max, not sum): If 1, the first components will be the value of the field (with dA if requested, without multFactor) at min/max.
- **`recordPosition`** – If 1, the coordinates of the min/max field will be added (after field values if desired).
- **`lbs`** – The lower bounds (can be "", for whole sim domain).
- **`ubs`** – The upper bounds (can be "", for whole sim domain).
- **`gridName`** – The name of the grid.
- **`gridBndry`** – The name of the grid boundary, can be "" for no boundary.
- **`inside`** – If 1, integrate inside gridBndry, if 0, outside.
- **`order`** – The order of interpolation to use (N.B. this volume integration has second-order error, so using order > 1 is probably not really useful).
- **`applyStep`** – The integer time step at which to calculate the history.
- **`multFactor`** – A factor by which to multiply the field energy.
- **`multFactor2`** – Second factor by which to multiply the field energy.
- **`label`** – A string-vec label for history file (can be "" for generic label).

addFieldSqrVolIntegralHist Macro

`addFieldSqrVolIntegralHist` (*histName*, *scope*, *fieldName*, *lbs*, *ubs*, *gridName*, *gridBndry*, *inside*, *order*, *applyStep*, *multFactor*, *label*)

A macro to add a history that calculates the volume integral of a field-squared within a given volume V, i.e., the integral over V of ||F||^2 dV (times multFactor)

Parameters

- **`histName`** – The name of the history.
- **`scope`** – The name of the MultiField or EmField containing the field (can be the empty string "", and the field name can then be qualified with the MultiField or EmField).
- **`fieldName`** – The name of the field used in the hstory.
- **`lbs`** – The lower bounds (can be "", for whole sim domain).
- **`ubs`** – The upper bounds (can be "", for whole sim domain).
- **`gridName`** – The name of the grid.
- **`gridBndry`** – The name of the grid boundary, can be "" for no boundary.
- **`inside`** – If 1, integrate inside gridBndry, if 0, outside.
- **`order`** – The order of interpolation to use (N.B. this volume integration has second-order error, so using order > 1 is probably not really useful).
- **`applyStep`** – The integer time step at which to calculate the history.
- **`multFactor`** – A factor by which to multiply the field energy.
- **`label`** – A string-vec label for history file (can be "" for generic label).

addFieldSqrVolMaxHist Macro

addFieldSqrVolMaxHist (*histName*, *scope*, *fieldName*, *recordFieldValue*, *recordPosition*, *lbs*, *ubs*, *gridName*, *gridBndry*, *inside*, *order*, *applyStep*, *multFactor*, *labels*)

A macro to add a history that calculates the volume integral of a field-squared within a given volume V, i.e., the integral over V of ||F||^2 dV (times multFactor)

Parameters

- **histName** – The name of the history.
- **scope** – The name of the MultiField or EmField containing the field (can be the empty string "", and the field name can then be qualified with the MultiField or EmField).
- **fieldName** – The name of the field used in the hstory.
- **recordFieldValue** – (for min or max, not sum): if 1, the first components will be the value of the field (with dA if requested, without multFactor) at min/max.
- **recordPosition** – If 1, the coordinates of the min/max field will be added (after field values if desired).
- **lbs** – The lower bounds (can be "", for whole sim domain).
- **ubs** – The upper bounds (can be "", for whole sim domain).
- **gridName** – The name of the grid.
- **gridBndry** – The name of the grid boundary, can be "" for no boundary.
- **inside** – If 1, integrate inside gridBndry, if 0, outside.
- **order** – The order of interpolation to use (N.B. this volume integration has second-order error, so using order > 1 is probably not really useful).
- **applyStep** – The integer time step at which to calculate the history.
- **multFactor** – A factor by which to multiply the field energy.
- **label** – A string-vec label for history file (can be "" for generic label).

addTwoFieldSqrVolMaxHist Macro

addTwoFieldSqrVolMaxHist (*histName*, *scope*, *fieldName1*, *fieldName2*, *recordFieldValue*, *recordPosition*, *lbs*, *ubs*, *gridName*, *gridBndry*, *inside*, *order*, *applyStep*, *multFactor*, *multFactor2*, *labels*)

A macro to add a history that calculates the max of the sum of two field-squared within a given volume V, e.g. max over V of the quantity: a(||F||^2 + b||G||^2) (where a = multFactor, b = multFactor2)

Parameters

- **histName** – The name of the history.
- **scope** – The name of the MultiField or EmField containing the field (can be the empty string "", and the field name can then be qualified with the MultiField or EmField).
- **fieldName1** – The name of the first field used in the hstory.
- **fieldName2** – The name of the second field used in the hstory.
- **recordFieldValue** – (for min or max, not sum): if 1, the first components will be the value of the field (with dA if requested, without multFactor) at min/max.

- **recordPosition** – If 1, the coordinates of the min/max field will be added (after field values if desired).
- **lbs** – The lower bounds (can be "", for whole sim domain).
- **ubs** – The upper bounds (can be "", for whole sim domain).
- **gridName** – The name of the grid.
- **gridBndry** – The name of the grid boundary, can be "" for no boundary.
- **inside** – If 1, integrate inside gridBndry, if 0, outside.
- **order** – The order of interpolation to use (N.B. this volume integration has second-order error, so using order > 1 is probably not really useful).
- **applyStep** – The integer time step at which to calculate the history.
- **multFactor** – A factor by which to multiply the field energy.
- **multFactor2** – Second factor by which to multiply the field energy.
- **label** – A string-vec label for history file (can be "" for generic label).

add2FieldSqrVolIntegralHistMacro

add2FieldSqrVolIntegralHist (*histName*, *scope*, *fieldName1*, *fieldName2*, *lbs*, *ubs*, *gridName*, *gridBndry*, *inside*, *order*, *applyStep*, *multFactor1*, *multFactor2*)

A macro to add a history that calculates the max of the sum of two field-squared within a given volume V, e.g. max over V of the quantity: a(||F||^2 + b||G||^2) (where a = multFactor, b = multFactor2)

Parameters

- **histName** – The name of the history.
- **scope** – The name of the MultiField or EmField containing the field (can be the empty string "", and the field name can then be qualified with the MultiField or EmField).
- **fieldName1** – The name of the field F.
- **fieldName2** – The name of the field G.
- **lbs** – The lower bounds (can be "", for whole sim domain).
- **ubs** – The upper bounds (can be "", for whole sim domain).
- **gridName** – The name of the grid.
- **gridBndry** – The name of the grid boundary, can be "" for no boundary.
- **inside** – If 1, integrate inside gridBndry, if 0, outside.
- **order** – The order of interpolation to use (N.B. this volume integration has second-order error, so using order > 1 is probably not really useful).
- **applyStep** – The integer time step at which to calculate the history.
- **multFactor1** – A factor by which to multiply ||F^2||.
- **multFactor2** – A factor by which to multiply ||G^2||.

addVolInnerProd3Hist Macro

addVolInnerProd3Hist (*histName*, *scope*, *fieldF*, *fieldG*, *lbs*, *ubs*, *gridName*, *gridBndry*, *inside*, *gridBndry2*, *inside2*, *order*, *applyStep*, *sMatrixDiags*, *sMatrixOffDiags multFactor*)

A macro to add a history that calculates the volume inner product two fields (which are 3-vector-valued) i.e., the integral over V of F.S.G dV (times multFactor) where S is a 3x3 symmetric matrix The volume V is the intersection of 2 regions, each defined by a gridBoundary

Parameters

- **histName** – The name of the history.
- **scope** – The name of the MultiField or EmField containing the field (can be the empty string "", and the field name can then be qualified with the MultiField or EmField).
- **fieldF** – The name of the field F.
- **fieldG** – The name of the field G.
- **lbs** – The lower bounds (can be "", for whole sim domain).
- **ubs** – The upper bounds (can be "", for whole sim domain).
- **gridName** – The name of the grid.
- **gridBndry** – The name of a gridBoundary within (or without) which the energy will be calculated (can be "" for no gridBndry).
- **inside** – If 1, integrate inside gridBndry, if 0, outside.
- **gridBndry2** – The name of a gridBoundary within (or without) which the energy will be calculated (can be "" for no gridBndry).
- **inside2** – If 1, integrate inside gridBndry, if 0, outside.
- **order** – The order of interpolation to use (N.B. this volume integration has second-order error, so using order > 1 is probably not really useful).
- **applyStep** – The integer time step at which to calculate the history.
- **sMatrixDiags** – A list [s11 s22 s33] of the diagonal elements of the sMatrix.
- **sMatrixOffDiags** – A list [s23 s31 s12] of the off-diagonal elements of the (symmetric) sMatrix.
- **multFactor** – A factor by which to multiply the field energy.

addFieldSqrSurfOpHist Macro

addFieldSqrSurfOpHist (*histName*, *scope*, *fieldName*, *operation*, *multiplyByDA*, *recordFieldValue*, *recordPosition*, *lbs*, *ubs*, *gridName*, *gridBndry*, *inside*, *order*, *applyStep*, *multFactor*, *labels*)

A macro to add a history that calculates field-squared over a surface e.g., the integral over A of ||F||^2 dA (times multFactor) or the maximum over A of ||F||^2 (times multFactor) or the minimum over A of ||F||^2 (times multFactor) (will always be the last value in the history, if more than that value is requested via recordFieldValue or recordPosition)

Parameters

- **histName** – The name of the history.
- **scope** – The name of the MultiField or EmField containing the field (can be the empty string "", and the field name can then be qualified with the MultiField or EmField).

- **`fieldName`** – The name of the field used in the history.
- **`operation`** – Either sumOverCells, maxOverCells, or minOverCells.
- **`multiplyByDA`** – Whether to multiply by dA, either 0 or 1 (false or true).
- **`recordFieldValue`** – (for min or max, not sum): if 1, the first components will be the components of the field (not multiplied by dA or multFactor) at min/max.
- **`recordPosition`** – (for min or max, not sum): if 1, the coordinates of the min/max field will be added (after field values if desired).
- **`lbs`** – The lower bounds (can be "", for whole sim domain).
- **`ubs`** – The upper bounds (can be "", for whole sim domain).
- **`gridName`** – The name of the grid.
- **`gridBndry`** – The name of a gridBoundary within (or without) which the energy will be calculated (can be "" for no gridBndry).
- **`inside`** – If 1, integrate inside gridBndry, if 0, outside.
- **`order`** – The order of interpolation to use (N.B. this volume integration has second-order error, so using order > 1 is probably not really useful).
- **`applyStep`** – The integer time step at which to calculate the history.
- **`multFactor`** – A factor by which to multiply the field energy.
- **`labels`** – A string-vec of descriptions of the results for the hdf5 file (can be "").

addTwoFieldSqrSurfOpHist Macro

`addTwoFieldSqrSurfOpHist` (*histName*, *scope*, *fieldName1*, *fieldName2 operation*, *multiplyByDA*, *recordFieldValue*, *recordPosition*, *lbs*, *ubs*, *gridName*, *gridBndry*, *inside*, *order*, *applyStep*, *multFactor*, *multFactor2*, *labels*)

A macro to add a history that calculates field-squared over a surface for two fields e.g., the integral over A of ||F||^2 + g ||G||^2 dA (times multFactor) or the maximum over A of ||F||^2 + g ||G||^2 (times multFactor) or the minimum over A of ||F||^2 + g ||G||^2 (times multFactor) where f and g are constants. (this will always be the last value in the history, if more than that value is requested via recordFieldValue or recordPosition)

Parameters

- **`histName`** – The name of the history.
- **`scope`** – The name of the MultiField or EmField containing the field (can be the empty string "", and the field name can then be qualified with the MultiField or EmField).
- **`fieldName1`** – The name of the field F.
- **`fieldName2`** – The name of the field G.
- **`operation`** – Either sumOverCells, maxOverCells, or minOverCells.
- **`multiplyByDA`** – Whether to multiply by dA, either 0 or 1 (false or true).
- **`recordFieldValue`** – (for min or max, not sum): If 1, the first components will be the components of the field (not multiplied by dA or multFactor) at min/max.
- **`recordPosition`** – (for min or max, not sum): If 1, the coordinates of the min/max field will be added (after field values if desired).
- **`lbs`** – The lower bounds (can be "", for whole sim domain).

- **ubs** – The upper bounds (can be "", for whole sim domain).
- **gridName** – The name of the grid.
- **gridBndry** – The name of a gridBoundary within (or without) which the energy will be calculated (can be "" for no gridBndry).
- **inside** – If 1, integrate inside gridBndry, if 0, outside.
- **order** – The order of interpolation to use (N.B. this volume integration has second-order error, so using order > 1 is probably not really useful).
- **applyStep** – The integer time step at which to calculate the history.
- **multFactor** – A factor by which to multiply the surface integral.
- **multFactor2** – A factor by which to multiply G.
- **labels** – A string-vec of descriptions of the results for the hdf5 file (can be "").

addFieldSqrSurfIntegralHist Macro

addFieldSqrSurfIntegralHist (*histName*, *scope*, *fieldName*, *lbs*, *ubs*, *gridName*, *gridBndry*, *inside*, *order*, *applyStep*, *multFactor*, *label*)

A macro to add a history that calculates the maximum or minimum of a field-magnitude-squared the maximum or minimum over a surface A of ||F||^2 (times multFactor).

Parameters

- **histName** – The name of the history.
- **scope** – The name of the MultiField or EmField containing the field (can be the empty string "", and the field name can then be qualified with the MultiField or EmField).
- **fieldName** – The name of the field.
- **lbs** – The lower bounds (can be "", for whole sim domain).
- **ubs** – The upper bounds (can be "", for whole sim domain).
- **gridName** – The name of the grid.
- **gridBndry** – The name of a gridBoundary within (or without) which the energy will be calculated (can be "" for no gridBndry).
- **inside** – If 1, integrate inside gridBndry, if 0, outside.
- **order** – The order of interpolation to use (N.B. this volume integration has second-order error, so using order > 1 is probably not really useful).
- **applyStep** – The integer time step at which to calculate the history.
- **multFactor** – A factor by which to multiply the surface integral.
- **labels** – A string-vec of descriptions of the results for the hdf5 file (can be "").

addFieldSqrSurfMaxHist Macro

addFieldSqrSurfMaxHist (*histName*, *scope*, *fieldName*, *recordFieldValue*, *recordPosition*, *lbs*, *ubs*, *gridName*, *gridBndry*, *inside*, *order*, *applyStep*, *multFactor*, *labels*)

A macro to add a history that calculates the maximum or minimum of a field-magnitude-squared the maximum or minimum over a surface A of ||F||^2 (times multFactor).

Parameters

- **`histName`** – The name of the history.
- **`scope`** – The name of the MultiField or EmField containing the field (can be the empty string "", and the field name can then be qualified with the MultiField or EmField).
- **`fieldName`** – The name of the field.
- **`recordFieldValue`** – (for min or max, not sum): If 1, the first components will be the components of the field (not multiplied by dA or multFactor) at min/max.
- **`recordPosition`** – (for min or max, not sum): If 1, the coordinates of the min/max field will be added (after field values if desired).
- **`lbs`** – The lower bounds (can be "", for whole sim domain).
- **`ubs`** – The upper bounds (can be "", for whole sim domain).
- **`gridName`** – The name of the grid.
- **`gridBndry`** – The name of a gridBoundary within (or without) which the energy will be calculated (can be "" for no gridBndry).
- **`inside`** – If 1, integrate inside gridBndry, if 0, outside.
- **`order`** – The order of interpolation to use (N.B. this volume integration has second-order error, so using order > 1 is probably not really useful).
- **`applyStep`** – The integer time step at which to calculate the history.
- **`multFactor`** – A factor by which to multiply the surface integral.
- **`labels`** – A string-vec of descriptions of the results for the hdf5 file (can be "").

addTwoFieldSqrSurfMaxHist Macro

`addTwoFieldSqrSurfMaxHist` (*histName*, *scope*, *fieldName1*, *fieldName2*, *recordFieldValue*, *recordPosition*, *lbs*, *ubs*, *gridName*, *gridBndry*, *inside*, *order*, *applyStep*, *multFactor*, *multFactor2*, *labels*)

A macro to add a history that calculates the maximum of two fields squared (times multFactor) over a surface: i.e., of (multFactor) * (||F||^2 + g ||G||^2) where g is a constant, and dA is optional.

Parameters

- **`histName`** – The name of the history.
- **`scope`** – The name of the MultiField or EmField containing the field (can be the empty string "", and the field name can then be qualified with the MultiField or EmField).
- **`fieldName1`** – The name of the field F.
- **`fieldName2`** – The name of the field G.
- **`recordFieldValue`** – (for min or max, not sum): If 1, the first components will be the components of the field (not multiplied by dA or multFactor) at min/max.
- **`recordPosition`** – (for min or max, not sum): If 1, the coordinates of the min/max field will be added (after field values if desired).
- **`lbs`** – The lower bounds (can be "", for whole sim domain).
- **`ubs`** – The upper bounds (can be "", for whole sim domain).
- **`gridName`** – The name of the grid.

- **gridBndry** – The name of a gridBoundary within (or without) which the energy will be calculated (can be "" for no gridBndry).
- **inside** – If 1, integrate inside gridBndry, if 0, outside.
- **order** – The order of interpolation to use (N.B. this volume integration has second-order error, so using order > 1 is probably not really useful).
- **applyStep** – The integer time step at which to calculate the history.
- **multFactor** – A factor by which to multiply the result.
- **multFactor** – A factor by which to multiply the second field.
- **labels** – A string-vec of descriptions of the results for the hdf5 file (can be "").

addFieldSqrSurfMinHist Macro

addFieldSqrSurfMinHist (*histName*, *scope*, *fieldName*, *recordFieldValue*, *recordPosition*, *lbs*, *ubs*, *gridName*, *gridBndry*, *inside*, *order*, *applyStep*, *multFactor*, *labels*)

A macro to add a history that calculates the minimum of a field-squared (times multFactor) over a surfaceN.B. This doesn't work unless the minimum value is below 3e38.

Parameters

- **histName** – The name of the history.
- **scope** – The name of the MultiField or EmField containing the field (can be the empty string "", and the field name can then be qualified with the MultiField or EmField).
- **fieldName** – The name of the field.
- **recordFieldValue** – (for min or max, not sum): If 1, the first components will be the components of the field (not multiplied by dA or multFactor) at min/max.
- **recordPosition** – (for min or max, not sum): If 1, the coordinates of the min/max field will be added (after field values if desired).
- **lbs** – The lower bounds (can be "", for whole sim domain).
- **ubs** – The upper bounds (can be "", for whole sim domain).
- **gridName** – The name of the grid.
- **gridBndry** – The name of a gridBoundary within (or without) which the energy will be calculated (can be "" for no gridBndry).
- **inside** – If 1, integrate inside gridBndry, if 0, outside.
- **order** – The order of interpolation to use (N.B. this volume integration has second-order error, so using order > 1 is probably not really useful).
- **applyStep** – The integer time step at which to calculate the history.
- **multFactor** – A factor by which to multiply the result.
- **labels** – A string-vec of descriptions of the results for the hdf5 file (can be "").

addFieldSurfComplex3DCrossIntegralHist Macro

addFieldSurfComplex3DCrossIntegralHist(*histName*, *scope*, *field1Re*, *field1Im*, *field2Re*, *field2Im*, *lbs*, *ubs*, *gridName*, *gridBndry*, *inside*, *order*, *applyStep*, *multFactor*)

A macro to add a history that calculates the surface-flux integral of a complex field cross product over a surface i.e., 0.5 * integral over A of Re[F1^* x F2].n dA (times multFactor) where F1 and F2 are 3-vector-valued fields and where n is always the outward unit normal (regardless of whether the region inside or outside of the GridBndry is used). The prefactor of 1/2 is so the answer reflects the time-average of the integral over A of (Re[F1 exp(-i w t)]xRe[F2 exp(-i w t)]).n dA

Parameters

- **histName** – The name of the history.
- **scope** – The name of the MultiField or EmField containing the field (can be the empty string "", and the field name can then be qualified with the MultiField or EmField).
- **field1Re** – The name of the real part of field F1.
- **field1Im** – The name of the imaginary part of field F1.
- **field2Re** – The name of the real part of field F2.
- **field2Im** – The name of the imaginary part of field F2.
- **lbs** – The lower bounds (can be "", for whole sim domain).
- **ubs** – The upper bounds (can be "", for whole sim domain).
- **gridName** – The name of the grid.
- **gridBndry** – The name of a gridBoundary within (or without) which the energy will be calculated (can be "" for no gridBndry).
- **inside** – If 1, integrate inside gridBndry, if 0, outside.
- **order** – The order of interpolation to use (N.B. this volume integration has second-order error, so using order > 1 is probably not really useful).
- **applyStep** – The integer time step at which to calculate the history.
- **multFactor** – A factor by which to multiply the result.

addVoltageGainInOscElecFieldHistBase Macro

addVoltageGainInOscElecFieldHistBase(*histName*, *scope*, *eField*, *freq*, *freqIndex*, *speed*, *initialPhase*, *lbs*, *ubs*, *gridCurve*, *order*, *applyStep*, *multFactor*)

A macro to add a history that calculates the voltage gained by a charged particle following a curve at constant speed in an oscillating field with given magnitude. For example, the line integral over the curve of ds.E(x) exp(-2 pi i freq (s(x)/speed) - i initialPhase) (times multFactor) where E is the electric field at point x (on the curve), s(x) is the length (along the curve) from the beginning of the curve to point x, and f is the field's oscillation frequency speed is the particle's speed along the curve initialPhase is the phase of the (complex oscillating) field when the particles starts along the curve. Or for example, the history returns the real and imaginary part of the above, so that if initialPhase = 0, then the real part represents the voltage gain by a particle that starts along the curve when the given field is at its maximum, and the imaginary part represents the voltage gain by a particle that starts along the curve when the given field is zero and decreasing (if freq is positive…if freq is negative, then because of the minus sign in exp(-…), it's as if the field is increasing from zero). (if multFactor is 1., the result is in Volts)

Parameters

- **`histName`** – The name of the history.
- **`scope`** – The name of the MultiField or EmField containing the field (can be the empty string "", and the field name can then be qualified with the MultiField or EmField).
- **`eField`** – The name of the e-field.
- **`freq`** – The oscillation frequency (not angular frequency) of the e-field or the name of a history that calculates the frequency (in the latter case, that history must come before this macro is called in the input file).
- **`freqIndex`** – If param freq is simply a number, this should be -1;if param freq is the name of a history that calculates the frequency, this should be the index (e.g., [0] or [1 2]) of the frequency within the history data.
- **`speed`** – The speed of a charged particle following the curve.
- **`initialPhase`** – The initial phase (in radians) of the (complex, oscillating) field when the particle starts along the curve.
- **`lbs`** – The lower bounds (can be "", for whole sim domain).
- **`ubs`** – The upper bounds (can be "", for whole sim domain).
- **`gridCurve`** – The name of a GridCurve along which the field will be integrated.
- **`order`** – The order of interpolation to use (N.B. this method of integration has second-order error, so using order > 1 is probably not really useful).
- **`applyStep`** – The integer time step at which to calculate the history.
- **`multFactor`** – A factor by which to multiply the result.

addVoltageGainInOscElecFieldHist Macro

`addVoltageGainInOscElecFieldHist`(*histName*, *scope*, *eField*, *freq*, *speed*, *initialPhase*, *lbs*, *ubs*, *gridCurve*, *order*, *applyStep*, *multFactor*)

A macro to add a history that calculates the voltage gained by a charged particle following a curve at constant speed in an oscillating field with given magnitude, (with given oscillation frequency). For example, the line integral over the curve of ds.E(x) exp(-2 pi i freq (s(x)/speed) - i initialPhase) (times multFactor) where E is the electric field at point x (on the curve), s(x) is the length (along the curve) from the beginning of the curve to point x, and f is the field's oscillation frequency speed is the particle's speed along the curve initialPhase is the phase of the (complex oscillating) field when the particles starts along the curve. Or for example, the history returns the real and imaginary part of the above, so that if initialPhase = 0, then the real part represents the voltage gain by a particle that starts along the curve when the given field is at its maximum, and the imaginary part represents the voltage gain by a particle that starts along the curve when the given field is zero and decreasing (if freq is positive…if freq is negative, then because of the minus sign in exp(-…), it's as if the field is increasing from zero). (if multFactor is 1., the result is in Volts)

Parameters

- **`histName`** – The name of the history.
- **`scope`** – The name of the MultiField or EmField containing the field (can be the empty string "", and the field name can then be qualified with the MultiField or EmField).
- **`eField`** – The name of the e-field.
- **`initialPhase`** – The initial phase (in radians) of the (complex, oscillating) field when the particle starts along the curve.

- **`lbs`** – The lower bounds (can be “”, for whole sim domain).
- **`ubs`** – The upper bounds (can be “”, for whole sim domain).
- **`gridCurve`** – The name of a GridCurve along which the field will be integrated.
- **`order`** – The order of interpolation to use (N.B. this method of integration has second-order error, so using order > 1 is probably not really useful).
- **`applyStep`** – The integer time step at which to calculate the history.
- **`multFactor`** – A factor by which to multiply the result.

addVoltageGainInOscElecFieldHist Macro

`addVoltageGainInOscElecFieldHist` (*histName*, *scope*, *eField*, *freq*, *speed*, *initialPhase*, *lbs*, *ubs*, *gridCurve*, *order*, *applyStep*, *multFactor*)

A macro to add a history that calculates the voltage gained by a charged particle following a curve at constant speed in an oscillating field with given magnitude,(with oscillation frequency given by another History). For example, the line integral over the curve of ds.E(x) exp(-2 pi i freq (s(x)/speed) - i initialPhase) (times multFactor) where E is the electric field at point x (on the curve), s(x) is the length (along the curve) from the beginning of the curve to point x, and f is the field’s oscillation frequency speed is the particle’s speed along the curve initialPhase is the phase of the (complex oscillating) field when the particles starts along the curve. Or for example, the history returns the real and imaginary part of the above, so that if initialPhase = 0, then the real part represents the voltage gain by a particle that starts along the curve when the given field is at its maximum, and the imaginary part represents the voltage gain by a particle that starts along the curve when the given field is zero and decreasing (if freq is positive… if freq is negative, then because of the minus sign in exp(-…), it’s as if the field is increasing from zero). (if multFactor is 1., the result is in Volts)

param histName The name of the history.

param scope The name of the MultiField or EmField containing the field (can be the empty string “”, and the field name can then be qualified with the MultiField or EmField).

param eField The name of the e-field.

param freq The oscillation frequency (not angular frequency) of the e-field or the name of a history that calculates the frequency (in the latter case, that history must come before this macro is called in the input file).

param speed The speed of a charged particle following the curve.

param initialPhase The initial phase (in radians) of the (complex, oscillating) field when the particle starts along the curve.

param lbs The lower bounds (can be “”, for whole sim domain).

param ubs The upper bounds (can be “”, for whole sim domain).

param gridCurve The name of a GridCurve along which the field will be integrated.

param order The order of interpolation to use (N.B. this method of integration has second-order error, so using order > 1 is probably not really useful).

param applyStep The integer time step at which to calculate the history.

param multFactor A factor by which to multiply the result.

addEnergyInBHist Macro

addEnergyInBHist (*histName*, *scope*, *bField*, *lbs*, *ubs*, *gridName*, *gridBndry*, *inside*, *order*, *applyStep*, *multFactor*)

A macro to add a history that calculates the energy in the magnetic (B) field within a given volume V, i.e., the integral over V of ||B||^2/(2 mu_0) dV (if multFactor is 1., the result is the magnetic field energy in Joules, although in 2D you it's really Joules/m, and in 1D, Joules/m^2)

Parameters

- **histName** – The name of the history.
- **scope** – The name of the MultiField or EmField containing the field (can be the empty string "", and the field name can then be qualified with the MultiField or EmField).
- **bField** – The name of the b-field.
- **lbs** – The lower bounds (can be "", for whole sim domain).
- **ubs** – The upper bounds (can be "", for whole sim domain).
- **gridBndry** – The name of a gridBoundary within (or without) which the energy will be calculated (can be "" for no gridBndry).
- **inside** – If 1, calculate energy inside gridBndry, if 0, outside.
- **order** – The order of interpolation to use (N.B. this method of integration has second-order error, so using order > 1 is probably not really useful).
- **applyStep** – The integer time step at which to calculate the history.
- **multFactor** – A factor by which to multiply the result.

addEnergyInBHistComplex Macro

addEnergyInBHistComplex (*histName*, *scope*, *Breal*, *Bimag*, *lbs*, *ubs*, *gridName*, *gridBndry*, *inside*, *order*, *applyStep*, *multFactor*)

A macro to add a history that calculates the energy in the complex magnetic (B) field within a given volume V, i.e., (1/2) the integral over V of ||B||^2/(2 mu_0) dV (if multFactor is 1., the result has units of Joules, although in 2D you it's really Joules/m, and in 1D, Joules/m^2). The prefactor (1/2) reflects the time-average of the integral over V of Re[B exp(-i w t)]^2/(2 mu_0) dV

Parameters

- **histName** – The name of the history.
- **scope** – The name of the MultiField or EmField containing the field (can be the empty string "", and the field name can then be qualified with the MultiField or EmField).
- **Bimag** – The imaginary part of the b-field.
- **Breal** – The real part of the b-field.
- **lbs** – The lower bounds (can be "", for whole sim domain).
- **ubs** – The upper bounds (can be "", for whole sim domain).
- **gridName** – The name of the grid.
- **gridBndry** – The name of a gridBoundary within (or without) which the energy will be calculated (can be "" for no gridBndry).
- **inside** – If 1, calculate energy inside gridBndry, if 0, outside.

- **order** – The order of interpolation to use (N.B. this method of integration has second-order error, so using order > 1 is probably not really useful).
- **applyStep** – The integer time step at which to calculate the history.
- **multFactor** – A factor by which to multiply the result.

addEnergyInEHist Macro

addEnergyInEHist (*histName*, *scope*, *eField*, *lbs*, *ubs*, *gridName*, *gridBndry*, *inside*, *order*, *applyStep*, *multFactor*)

A macro to add a history that calculates the energy in the electric (E) field within a given volume V, i.e., the integral over V of ||B||^2/(2 mu_0) dV (if multFactor is 1., the result is the magnetic field energy in Joules, although in 2D you it's really Joules/m, and in 1D, Joules/m^2)

Parameters

- **histName** – The name of the history.
- **scope** – The name of the MultiField or EmField containing the field (can be the empty string "", and the field name can then be qualified with the MultiField or EmField).
- **eField** – The name of the e-field.
- **lbs** – The lower bounds (can be "", for whole sim domain).
- **ubs** – The upper bounds (can be "", for whole sim domain).
- **gridBndry** – The name of a gridBoundary within (or without) which the energy will be calculated (can be "" for no gridBndry).
- **inside** – If 1, calculate energy inside gridBndry, if 0, outside.
- **order** – The order of interpolation to use (N.B. this method of integration has second-order error, so using order > 1 is probably not really useful).
- **applyStep** – The integer time step at which to calculate the history.
- **multFactor** – A factor by which to multiply the result.

addEnergyInEHistComplex Macro

addEnergyInEHistComplex (*histName*, *scope*, *Ereal*, *Eimag*, *lbs*, *ubs*, *gridName*, *gridBndry*, *inside*, *order*, *applyStep*, *multFactor*)

A macro to add a history that calculates the energy in the complex electric (E) field within a given volume V, i.e., (1/2) the integral over V of ||B||^2/(2 mu_0) dV (if multFactor is 1., the result has units of Joules, although in 2D you it's really Joules/m, and in 1D, Joules/m^2). The prefactor (1/2) reflects the time-average of the integral over V of Re[B exp(-i w t)]^2/(2 mu_0) dV

Parameters

- **histName** – The name of the history.
- **scope** – The name of the MultiField or EmField containing the field (can be the empty string "", and the field name can then be qualified with the MultiField or EmField).
- **Eimag** – The imaginary part of the e-field.
- **Ereal** – The real part of the e-field.
- **lbs** – The lower bounds (can be "", for whole sim domain).

- **`ubs`** – The upper bounds (can be “”, for whole sim domain).
- **`gridName`** – The name of the grid.
- **`gridBndry`** – The name of a gridBoundary within (or without) which the energy will be calculated (can be “” for no gridBndry).
- **`inside`** – If 1, calculate energy inside gridBndry, if 0, outside.
- **`order`** – The order of interpolation to use (N.B. this method of integration has second-order error, so using order > 1 is probably not really useful).
- **`applyStep`** – The integer time step at which to calculate the history.
- **`multFactor`** – A factor by which to multiply the result.

addEnergyInDielectricFromEHist Macro

`addEnergyInDielectricFromEHist` (*histName*, *scope*, *fieldName*, *lbs*, *ubs*, *gridName*, *dielGridBndry*, *insideDiel*, *volumeGridBndry*, *insideVolume*, *order*, *applyStep*, *epsilonDiags*, *epsilonOffDiags*, *multFactor*)

A macro to add a history that calculates the energy in the electric (E) field within a region of uniform dielectric in a given volume V, i.e., the integral over intersection(V, diel. region) of D.E/2 dV (if multFactor is 1., the result has units of Joules, although in 2D you it’s really Joules/m, and in 1D, Joules/m^2)

Parameters

- **`histName`** – The name of the history.
- **`scope`** – The name of the MultiField or EmField containing the field (can be the empty string “”, and the field name can then be qualified with the MultiField or EmField).
- **`lbs`** – The lower bounds (can be “”, for whole sim domain).
- **`ubs`** – The upper bounds (can be “”, for whole sim domain).
- **`gridName`** – The name of the grid.
- **`dielGridBndry`** – The name of a gridBoundary that describes a region within (or without) which is uniform dielectric–a region in which the energy will be calculated (can be “” for no gridBndry).
- **`insideDiel`** – If 1, calculate energy inside dielGridBndry, if 0, outside.
- **`volumeGridBndry`** – The name of a gridBoundary within (or without) which the energy will be calculated – this has nothing to with dielectric, just a way to say: calculate the energy within volume V (volumeGridBndry) within the dielectric; the volumeGridBndry should not intersect the dielGridBndry/ (the surfaces should not even cut through the same cell)
- **`insideVolume`** – If 1, calculate energy inside volumeGridBndry, if 0, outside.
- **`order`** – The order of interpolation to use (N.B. this method of integration has second-order error, so using order > 1 is probably not really useful).
- **`applyStep`** – The integer time step at which to calculate the history.
- **`epsilonDiags`** – A list [eps_xx eps_yy eps_zz] of the diagonal elements of the relative epsilon matrix (e.g., [1 1 1] in vacuum).
- **`epsilonOffDiags`** – A list [eps_yz eps_zx eps_xy] of the off-diagonal elements of the (symmetric) relative epsilon matrix (e.g., [0 0 0] in vacuum, or any isotropic dielectric)
- **`multFactor`** – A factor by which to multiply the result.

addInstantWallDissipationHist Macro

addInstantWallDissipationHist (*histName*, *scope*, *fieldName*, *lbs*, *ubs*, *gridName*, *gridBndry*, *inside*, *order*, *applyStep*, *R_S*, *multFactor*, *label*)

A macro to add a history that calculates the instantaneous energy dissipation (power) in the metal wall containing given an RF magnetic field (B). i.e., the integral over A of R_s ||B/mu_0||^2 dA It is assumed that the B-field conforms to the metallic boundary conditions… if not, it may be better to add 'boundaryCondition = magneticAtPEC' to the <SpaceFunc B> block. (if multFactor is dimensionless, the result has units of Watts, although in 2D it's really W/m, and in 1D, W/m^2)

Parameters

- **histName** – The name of the history.
- **scope** – The name of the MultiField or EmField containing the field (can be the empty string "", and the field name can then be qualified with the MultiField or EmField).
- **lbs** – The lower bounds (can be "", for whole sim domain).
- **ubs** – The upper bounds (can be "", for whole sim domain).
- **gridName** – The name of the grid.
- **gridBndry** – The name of a gridBoundary describing the metal wall.
- **inside** – If 1, calculate energy inside gridBndry, if 0, outside.
- **order** – The order of interpolation to use (N.B. this method of integration has second-order error, so using order > 1 is probably not really useful).
- **applyStep** – The integer time step at which to calculate the history.
- **R_S** – The surface resistance of the metal (in Ohms)–this should be found from experiment (it depends on frequency), but in naive theory, R_S = rho / delta where rho is the metal's bulk resistivity, and delta is the skin depth at the frequency in question.
- **multFactor** – A factor by which to multiply the result.
- **label** – A description of the result for the hdf5 file; can be set to "defaultLabel" for default label.

addTimeAvgWallDissipationHist Macro

addInstantWallDissipationHist (*histName*, *scope*, *fieldName*, *lbs*, *ubs*, *gridName*, *gridBndry*, *inside*, *order*, *applyStep*, *R_S*, *multFactor*, *label*)

A macro to add a history that calculates the time-averaged energy dissipation (power) in the metal wall containing given an RF magnetic field oscillation amplitude (B). i.e., the integral over A of (1/2) R_s ||B/mu_0||^2 dA It is assumed that the B-field conforms to the metallic boundary conditions… if not, it may be better to add 'boundaryCondition = magneticAtPEC' to the <SpaceFunc B> block. (if multFactor is 1., the result has units of Watts.

Parameters

- **histName** – The name of the history.
- **scope** – The name of the MultiField or EmField containing the field (can be the empty string "", and the field name can then be qualified with the MultiField or EmField).
- **lbs** – The lower bounds (can be "", for whole sim domain).
- **ubs** – The upper bounds (can be "", for whole sim domain).

- **`gridName`** – The name of the grid.
- **`gridBndry`** – The name of a gridBoundary describing the metal wall.
- **`inside`** – If 1, calculate energy inside gridBndry, if 0, outside.
- **`order`** – The order of interpolation to use (N.B. this method of integration has second-order error, so using order > 1 is probably not really useful).
- **`applyStep`** – The integer time step at which to calculate the history.
- **`R_S`** – The surface resistance of the metal (in Ohms)–this should be found from experiment (it depends on frequency), but in naive theory, R_S = rho / delta where rho is the metal's bulk resistivity, and delta is the skin depth at the frequency in question.
- **`multFactor`** – A factor by which to multiply the result.
- **`label`** – A description of the result for the hdf5 file; can be set to "defaultLabel" for default label.

addComplexPoyntingSurfaceIntegralHist Macro

`addComplexPoyntingSurfaceIntegralHist` (*histName*, *scope*, *Er*, *Ei*, *Br*, *Bi*, *lbs*, *ubs*, *gridName*, *gridBndry*, *inside*, *order*, *applyStep*, *multFactor*)

A macro to add a history that calculates the Poynting-flux integral of complex EM fields over a surface i.e., 1/mu_0 times 1/2 integral over A of (E^* x B).n dA (times multFactor) (the 1/2 is so the result yields the time average power crossing the surface) where E and B are the electric and magnetic fields and where n is always the outward unit normal (regardless of whether the region inside or outside of the GridBndry is used) where the surface A is described by the GridBndry

Parameters

- **`histName`** – The name of the history.
- **`scope`** – The name of the MultiField or EmField containing the field (can be the empty string "", and the field name can then be qualified with the MultiField or EmField).
- **`Er`** – The real part of the e-field
- **`Ei`** – The imaginary part of the e-field.
- **`Br`** – The real part of the b-field.
- **`Bi`** – The imaginary part of the b-field.
- **`lbs`** – The lower bounds (can be "", for whole sim domain).
- **`ubs`** – The upper bounds (can be "", for whole sim domain).
- **`gridName`** – The name of the grid.
- **`gridBndry`** – The name of a gridBoundary describing the metal wall.
- **`inside`** – If 1, calculate energy inside gridBndry, if 0, outside.
- **`order`** – The order of interpolation to use (N.B. this method of integration has second-order error, so using order > 1 is probably not really useful).
- **`applyStep`** – The integer time step at which to calculate the history.
- **`multFactor`** – A factor by which to multiply the result.

initBeam.mac

This macro file can be imported to an input file with `$ import initBeam`.

This macro file is available to all packages, but is most useful for VSimPA cases.

This macro file can be used to initialized the electomagnetic fields of a beam initialized inside the simulation window. A static Poisson solve is performed in the frame of the electron beam, the E and B fields are transformed back in the laboratory frame and set up on the grid.

initBeam Macro

`initBeam`(*density*, *elecField*, *magField*, *gammav*, *xlo*, *ylo*, *zlo*, *xhi*, *yhi*, *zhi*, *dx*, *dy*, *dz*, *ndim*, *dumpFields*)

Define the fields, updaters and initial update steps initializing the electric and magnetic field of a charged particle beam on the grid, using Dirichlet boundary conditions.

`initBeamWithOpenBdry`(*density*, *elecField*, *magField*, *gammav*, *xlo*, *ylo*, *zlo*, *xhi*, *yhi*, *zhi*, *dx*, *dy*, *dz*, *ndim*, *dumpFields*)

Define the fields, updaters and initial update steps initializing the electric and magnetic field of a charged particle beam on the grid, using open boundary conditions.

Parameters

- **`density`** – Charge density of the beam (located at the nodes).
- **`elecField`** – Resulting electric field (beam self-field, located at the edges).
- **`magField`** – Resulting magnetic field (beam self-field, located on the face).
- **`gammav`** – Lorentz factor (gamma) of the charged particle beam in its direction of propagation. `gammav` is used to do the Lorentz transformation in the beam frame.
- **`nz`** – Number of cells in the Z-direction.
- **`xlo`** – Lower boundary cell in x to solve for the fields.
- **`ylo`** – Lower boundary cell in y to solve for the fields.
- **`zlo`** – Lower boundary cell in z to solve for the fields.
- **`xhi`** – Upper boundary cell in x to solve for the fields.
- **`yhi`** – Upper boundary cell in y to solve for the fields.
- **`zhi`** – Upper boundary cell in z to solve for the fields.
- **`dx`** – Grid size in the x direction.
- **`dy`** – Grid size in the y direction.
- **`dz`** – Grid size in the z direction.
- **`ndim`** – Dimension of the simulation.
- **`dumpFields`** – Whether to dump the intermediary fields used to calculate the beam self-fields. If set to 0, those fields will not be dumped.

Example of initBeam Macro

```
<MultiField emField>
   # field used to deposit the charge density of the charged particle beam
   # located at the nodes
   <Field rhoBeam>
     numComponents = 1
     offset = none
     kind = depField
   </Field>

   # Electric field on edges
   <Field edgeE>
     numComponents = 3
     offset = edge
     interpolation = esirk2ndOrder
   </Field>

    # Magnetic field on faces
   <Field faceB>
     numComponents = 3
     offset = face
     interpolation = esirk2ndOrder
   </Field>

   initBeam(rhoBeam, edgeE, faceB, BEAM_GAMMA, 0, 0, 0, NX, NY, NZ, DX, DY, DZ, NDIM,␣
↪0)

</MultiField>
```

mal.mac

This macro file can be imported to an input file with

```
$ import mal
```

This macro file is available to all packages.

This set of macros applies a MAL in the slab specified. A matched absorbing layer uses isotropic electric and magnetic damping profiles to absorb the incident wave. This is unlike a PML (Perfectly Matched Layer) which uses the same electric and magnetic damping profiles, but is anisotropic. MAL boundaries are more stable as an anisotropic boundary condition can become unstable when the incident wave has a non-zero imaginary part to its normal wavenumber (e.g., fringing fields from nearby structure, or particles entering the layer)

This set of macros consists of 5 macros, all of which accept nearly the same exact arguments.

There are two versions of this macro:

- mal is for computations on a CPU
- malGPU is used for computations on a GPU

They retain the exact same functionality, however GPU and CPU macros cannot be mixed.

MALprofileDump Macro

MALprofileDump(*malName*, *nxl*, *nyl*, *nzl*, *nxu*, *nyu*, *nzu*, *maldir*, *malsign*, *startval*, *delta*, *frac*, *pwr*)

OR

MALprofileDump (*malName*, *nxl*, *nyl*, *nzl*, *nxu*, *nyu*, *nzu*, *maldir*, *malsign*, *startval*, *delta*, *frac*, *pwr*, *stfunc*, *malverb*)

This macro is useful for debugging purposes, as it gives the MAL profile.

Parameters

- **malName** – Name of the MAL layer.
- **nxl** – Lower bound in the x direction.
- **nyl** – Lower bound in the y direction.
- **nzl** – Lower bound in the z direction.
- **nxu** – Upper bound in the x direction.
- **nyu** – Upper bound in the y direction.
- **nzu** – Upper bound in the z direction.
- **maldir** – 0, 1, or 2 for the x, y, or z direction of the MAL.
- **malsign** – The sign of the outgoing direction.
- **startval** – Any offset of the grid start position in the direction of the MAL (XSTART, YSTART, ZSTART).
- **delta** – The grid size in the MAL direction.
- **frac** – The peak damping amplitude: 0.5 is suggested, typical range is 0.125 to 2.0.
- **pwr** – The damping profile goes as x**pwer: 3.0 is suggested, typical range is 1.0 to 4.0.
- **stfunc** – An optional STFunc which may be used to mask the damping profile, for example, to avoid a region that will interact with a beam.
- **malverb** – The verbosity level that should be used by this macro, for example to assist with debugging.

MALdampB_beforeFaraday Macro

MALdampB_beforeFaraday (*bField*, *malName*, *nxl*, *nyl*, *nzl*, *nxu*, *nyu*, *nzu*, *maldir*, *malsign*, *startval*, *delta*, *frac*, *pwr*)

OR

MALdampB_beforeFaraday (*bField*, *malName*, *nxl*, *nyl*, *nzl*, *nxu*, *nyu*, *nzu*, *maldir*, *malsign*, *startval*, *delta*, *frac*, *pwr*, *stfunc*, *malverb*)

This macro must be called in the update step order before the first Faraday update step.

Parameters

- **bField** – Name of the magnetic field.
- **malName** – Name of the MAL layer.
- **nxl** – Lower bound in the x direction.
- **nyl** – Lower bound in the y direction.
- **nzl** – Lower bound in the z direction.
- **nxu** – Upper bound in the x direction.
- **nyu** – Upper bound in the y direction.
- **nzu** – Upper bound in the z direction.

- **maldir** – 0, 1, or 2 for the x, y, or z direction of the MAL.
- **malsign** – The sign of the outgoing direction.
- **startval** – Any offset of the grid start position in the direction of the MAL (XSTART, YSTART, ZSTART).
- **delta** – The grid size in the MAL direction.
- **frac** – The peak damping amplitude: 0.5 is suggested, typical range is 0.125 to 2.0.
- **pwr** – The damping profile goes as x**pwer: 3.0 is suggested, typical range is 1.0 to 4.0.
- **stfunc** – An optional STFunc which may be used to mask the damping profile, for example, to avoid a region that will interact with a beam.
- **malverb** – The verbosity level that should be used by this macro, for example to assist with debugging.

MALdampE_beforeAmpere Macro

MALdampE_beforeAmpere (*eFieId*, *malName*, *nxl*, *nyl*, *nzl*, *nxu*, *nyu*, *nzu*, *maldir*, *malsign*, *startval*, *delta*, *frac*, *pwr*)

OR

MALdampE_beforeAmpere (*eField*, *malName*, *nxl*, *nyl*, *nzl*, *nxu*, *nyu*, *nzu*, *maldir*, *malsign*, *startval*, *delta*, *frac*, *pwr*, *stfunc*, *malverb*)

This macro must be called in the update step order before the Ampere update step.

Parameters

- **eField** – Name of the electric field.
- **malName** – Name of the MAL layer.
- **nxl** – Lower bound in the x direction.
- **nyl** – Lower bound in the y direction.
- **nzl** – Lower bound in the z direction.
- **nxu** – Upper bound in the x direction.
- **nyu** – Upper bound in the y direction.
- **nzu** – Upper bound in the z direction.
- **maldir** – 0, 1, or 2 for the x, y, or z direction of the MAL.
- **malsign** – The sign of the outgoing direction.
- **startval** – Any offset of the grid start position in the direction of the MAL (XSTART, YSTART, ZSTART).
- **delta** – The grid size in the MAL direction.
- **frac** – The peak damping amplitude: 0.5 is suggested, typical range is 0.125 to 2.0.
- **pwr** – The damping profile goes as x**pwer: 3.0 is suggested, typical range is 1.0 to 4.0.
- **stfunc** – An optional STFunc which may be used to mask the damping profile, for example, to avoid a region that will interact with a beam.
- **malverb** – The verbosity level that should be used by this macro, for example to assist with debugging.

MALdampE_afterAmpere Macro

MALdampE_afterAmpere (*eField*, *malName*, *nxl*, *nyl*, *nzl*, *nxu*, *nyu*, *nzu*, *maldir*, *malsign*, *startval*, *delta*, *frac*, *pwr*)

OR

MALdampE_afterAmpere (*eField*, *malName*, *nxl*, *nyl*, *nzl*, *nxu*, *nyu*, *nzu*, *maldir*, *malsign*, *startval*, *delta*, *frac*, *pwr*, *stfunc*, *malverb*)

This macro must be called in the update step order after the Ampere update step.

Parameters

- **eField** – Name of the electric field.
- **malName** – Name of the MAL layer.
- **nxl** – Lower bound in the x direction.
- **nyl** – Lower bound in the y direction.
- **nzl** – Lower bound in the z direction.
- **nxu** – Upper bound in the x direction.
- **nyu** – Upper bound in the y direction.
- **nzu** – Upper bound in the z direction.
- **maldir** – 0, 1, or 2 for the x, y, or z direction of the MAL.
- **malsign** – The sign of the outgoing direction.
- **startval** – Any offset of the grid start position in the direction of the MAL (XSTART, YSTART, ZSTART).
- **delta** – The grid size in the MAL direction.
- **frac** – The peak damping amplitude: 0.5 is suggested, typical range is 0.125 to 2.0.
- **pwr** – The damping profile goes as x**pwer: 3.0 is suggested, typical range is 1.0 to 4.0.
- **stfunc** – An optional STFunc which may be used to mask the damping profile, for example, to avoid a region that will interact with a beam.
- **malverb** – The verbosity level that should be used by this macro, for example to assist with debugging.

MALdampB_afterFaraday Macro

MALdampB_afterFaraday (*bField*, *malName*, *nxl*, *nyl*, *nzl*, *nxu*, *nyu*, *nzu*, *maldir*, *malsign*, *startval*, *delta*, *frac*, *pwr*)

OR

MALdampB_afterFaraday (*bField*, *malName*, *nxl*, *nyl*, *nzl*, *nxu*, *nyu*, *nzu*, *maldir*, *malsign*, *startval*, *delta*, *frac*, *pwr*, *stfunc*, *malverb*)

This macro must be called in the update step order after the second Faraday update step.

Parameters

- **bField** – Name of the magnetic field.
- **malName** – Name of the MAL layer.
- **nxl** – Lower bound in the x direction.

- **nyl** – Lower bound in the y direction.
- **nzl** – Lower bound in the z direction.
- **nxu** – Upper bound in the x direction.
- **nyu** – Upper bound in the y direction.
- **nzu** – Upper bound in the z direction.
- **maldir** – 0, 1, or 2 for the x, y, or z direction of the MAL.
- **malsign** – The sign of the outgoing direction.
- **startval** – Any offset of the grid start position in the direction of the MAL (XSTART, YSTART, ZSTART).
- **delta** – The grid size in the MAL direction.
- **frac** – The peak damping amplitude: 0.5 is suggested, typical range is 0.125 to 2.0.
- **pwr** – The damping profile goes as x**pwer: 3.0 is suggested, typical range is 1.0 to 4.0.
- **stfunc** – An optional STFunc which may be used to mask the damping profile, for example, to avoid a region that will interact with a beam.
- **malverb** – The verbosity level that should be used by this macro, for example to assist with debugging.

matrix.mac

This macro file can be imported to an input file with `$ import matrix`.

This macro file is available to all packages.

Provide a set of routines for creating matrices. These are relatively low-level methods for constructing matrices.

Stencil Element Macro

StencilElementMacro (*name*, *value*, *minDim*, *offset*, *rowFldIndx*, *colFldIndx*)

Macro to insert a constant-value matrix StencilElement.

Parameters

- **name** – The name to give the stencil element.
- **value** – The value of the stencil element.
- **minDim** – The minimum dimensionality to apply this stencil element.
- **offset** – The offset (in index space) of the column cell from the row cell.
- **rowFldIndx** – The index in the matrix for the row. This corresponds to the writefield for multiply, to the readfield for solve.
- **colFldIndx** – The index in the matrix for the column. This corresponds to the readfield for multiply, to the writefield for solve.

STFunc Stencil Element Macro

STFuncStencilElementMacro (*name*, *value*, *minDim*, *celloffset*, *funcOffset*, *rowFldIndx*, *colFldIndx*)
A macro to insert a constant-value matrix StencilElement

Parameters

- **name** – The name to give the stencil element.
- **value** – The value of the stencil element.
- **minDim** – The minimum dimensionality to apply this stencil element.
- **celloffset** – The offset (in index space) of the column cell from the row cell.
- **funcOffset** – The offset (in real space, as a multiple of the grid spacing) of the function evaluation point from the row cell.
- **rowFldIndx** – The index in the matrix for the row. This corresponds to the writefield for multiply, to the readfield for solve.
- **colFldIndx** – The index in the matrix for the column. This corresponds to the readfield for multiply, to the writefield for solve.

Coord Prod STFunc Stencil Element Macro

CoordProdSTFuncStencilElementMacro (*name*, *value*, *minDim*, *cellOffset*, *funcOffset*, *gridDir*, *gridOffset*, *rowFldIndx*, *colFldIndx*)
A macro to insert a constant-value matrix StencilElement

Parameters

- **name** – The name to give the stencil element.
- **value** – The value of the stencil element.
- **minDim** – The minimum dimensionality to apply this stencil element.
- **celloffset** – The offset (in index space) of the column cell from the row cell.
- **funcOffset** – The offset (in real space, as a multiple of the grid spacing) of the function evaluation point from the row cell.
- **gridDir** – The direction in which differencing is being done for the stencil. Determines which grid spacing and face area to use.
- **gridOffset** – The offset (in index space) from the row cell at which the grid spacing and face area are taken.
- **rowFldIndx** – The index in the matrix for the row. This corresponds to the writefield for multiply, to the readfield for solve.
- **colFldIndx** – The index in the matrix for the column. This corresponds to the readfield for multiply, to the writefield for solve.

mirror.mac

This macro file can be imported to an input file with `$ import mirrors`.

This macro file is encrypted and available with a VSimEM or VSimMD license.

MirrorAfterFaradaySetMagFieldToZero Macro

`MirrorAfterFaradaySetMagFieldToZero`(*IXBGN*, *IYBGN*, *IZBGN*, *IXEND*, *IYEND*, *IZEND*, *BFIELD*, *DTFRACTION*)

This macro implements a mirror in the specified region for simulations without particles It would be inserted after each Faraday step.

Parameters

- **`IXBGN`** – Beginning of the mirror in the X direction.
- **`IYBGN`** – Beginning of the mirror in the Y direction.
- **`IZBGN`** – Beginning of the mirror in the Z direction.
- **`IXEND`** – End of the mirror in the X direction.
- **`IYEND`** – End of the mirror in the Y direction.
- **`IZEND`** – End of the mirror in the Z direction.
- **`BFIELD`** – Face magnetic field.
- **`DTFRACTION`** – DtFrac, typically .5 or 1.0.

MirrorAfterFaradaySetCurrentDensToZero Macro

`MirrorAfterFaradaySetCurrentDensToZero`(*IXBGN*, *IYBGN*, *IZBGN*, *IXEND*, *IYEND*, *IZEND*, *CURRENTDENS*, *BFIELD*, *DTFRACTION*)

This macro implements a mirror in the specified region for simulations with particles. It would be inserted after each faraday step after calling the MirrorAfterFaradaySetMagFieldToZero macro.

Parameters

- **`IXBGN`** – Beginning of the mirror in the X direction.
- **`IYBGN`** – Beginning of the mirror in the Y direction.
- **`IZBGN`** – Beginning of the mirror in the Z direction.
- **`IXEND`** – End of the mirror in the X direction.
- **`IYEND`** – End of the mirror in the Y direction.
- **`IZEND`** – End of the mirror in the Z direction.
- **`CURRENTDENS`** – Face current density.
- **`BFIELD`** – Face magnetic field.
- **`DTFRACTION`** – DtFrac, typically .5 or 1.0.

ncnmac.mac

This macro file can be imported to an input file with `$ import ncnmac`.

This macro file is available to all packages.

This macro file is used for using the Crank-Nicholson implicit solve for advancing the electromagnetic fields. It is an update to the cnmacro macro.

ncnMatrix Macro

ncnMatrix(*name*, *sign*)

Uses the ∇^2 operator. This has the standard problem for implict updates of hyperbolic systems. Namely, that the operator becomes singular for very implicit times and so takes long to converge

Parameters

- **name** – The name of the matrix.
- **sign** – Sign (1) for the explicit or application part, (-1) for for the implicit or solve part.

ncnEpetraUpdater Macro

ncnEpetraUpdaters()

Defines the explicit and implicit updaters for Crank-Nicholson EM. This macro takes no parameters

cnEmFieldAlgorithm Macro

cnEmFieldAlgorithm(*withNodalFields*)

This macros inserts the Crank-Nicholson update. It assumes that edgeElec and faceMag have been defined. It defines any fields needed for just this algorithm.

Parameters **withNodalFields** – Whether to create and update nodalE and nodalB. This is set to nonzero to be used with particles. With large implicitness, one may need to set maxcellxing for the particles.

originRadiatorPort.mac

This macro file can be imported to an input file with `$ import radPort`.

This macro file is available to the VSimEM and VSimMD packages.

A tuned phase-velocity port is an open or outgoing boundary condition. Waves traveling at exactly the specified phase velocity of the boundary will exit the simulation with no reflection at all. Waves traveling at other phase velocities will partially exit and partially reflect, with a power reflection oefficient of rho=(VphaseWave-VphaseBoundary)^2/(VphaseWave+VphaseBoundary)^2.

The user supplied phase-velocity is assumed to be that for normal incidence to the simualtion boundary. An additional 1/cosine(angle) factor is applied to the phase velocity, where the angle is between the boundary normal and the line from the origin to the point on the boundary.

This boundary condition is subject to slow-time instability if there is metallic structure and associated fringe or near fields too close to the boundary. For this reason, the boundary condition works best if used to bound a radiation type simulation, where the boundary is sufficiently distant from the radiation source.

When working within a macro defined multifield (i.e. using yee) The RadPort can be defined with one simple macro call, given below.

addRadPortBoundary Macro

addRadPortBoundary(*position*, *VPHASE*)

This macro is used to apply a port boundary optimized for a radiator at the origin with a macro generated multifield. It handles the before and after Ampere updaters as wells as inserting the update steps in the correct order.

Parameters

- **position** – Which face to apply the port boundary, specified by lowerX,lowerY,lowerZ,upperX,upperY or upperZ.
- **VPHASE** – Phase velocity tuning parameter, in meters/second.

If using a user defined multifield, the port must be applied once before, and once after the ampere updated. These two function calls are given below.

applyPortAtBoundary_beforeAmpere Macro

applyPortAtBoundary_beforeAmpere(*IDIR*, *LOWERUPPER*, *VPHASE*, *DGRID*, *DT*, *EFIELD*, *NX*, *NY*, *NZ*)

The boundary is presently designed for cartesian dimensions only.

This macro should be called before the Ampere update steps

Parameters

- **IDIR** – 0,1,2 for the direction, x, y, or z, of the outgoing wave.
- **LOWERUPPER** – 0,1 for the lower plane or upper plane, respectively.
- **VPHASE** – Phase velocity tuning parameter, in meters/second.
- **DGRID** – Cell size, e.g., DX, DY, or DZ, in the direction of IDIR, in front of boundary, in meters.
- **DT** – Time step, in seconds.
- **EFIELD** – Name of the Yee (edge) electric field use in the Ampere updater.
- **nx** – Number of cells in x direction.
- **ny** – Number of cells in y direction.
- **nz** – Number of cells in z direction.

Use of this boundary requires that the Ampere updater properly omit updating of the electric fields at the domain limits. This is insured when the Ampere updater looks like:

```
<FieldMultiUpdater yeeAmpere>
  kind = yeeAmpereUpdater
  lowerBounds = [0 0 0]
  upperBounds = [NX NY NZ]
  contractFromBottomInNonComponentDir = 1
  .
  .
</FieldMultiUpdater>
```

The “contractFromBottomInNonComponentDir = 1” being critical to this end.

applyPortAtBoundary_afterAmpere Macro

applyPortAtBoundary_afterAmpere(*IDIR*, *LOWERUPPER*, *VPHASE*, *DGRID*, *DT*, *EFIELD*, *NX*, *NY*, *NZ*)

This macro should be called after the Ampere update steps

Parameters

- **IDIR** – 0,1,2 for the direction, x, y, or z, of the outgoing wave.

- **LOWERUPPER** – 0,1 for the lower plane or upper plane, respectively
- **VPHASE** – Phase velocity tuning parameter, in meters/second.
- **DGRID** – Cell size, e.g., DX, DY, or DZ, in the direction of IDIR, in front of boundary, in meters.
- **DT** – Time step, in seconds.
- **EFIELD** – Name of the Yee (edge) electric field use in the Ampere updater.
- **nx** – Number of cells in x direction.
- **ny** – Number of cells in y direction.
- **nz** – Number of cells in z direction.

perfectDispersion.mac

This macro file can be imported to an input file with `$ import perfectDispersion`.

This macro file is available to the VSimPA package.

This macro file consists of the macros necessary for defining the smoothed B update for dispersionless propagation.

For further information on the method used, please see *[CBC+13]*.

perfDispCoeffs Macro

perfDispCoeffs (*DELTA*, *DXI*, *DYI*, *DZI*)

Defines the controlled dispersion smoothing stencil coefficients.

Parameters

- **DELTA** – The length of one cell.
- **DXI** – 1/DX.
- **DYI** – 1/DY.
- **DZI** – 1/DZ.

perfectDispersionFaradayUpdaters Macro

perfectDispersionFaradayUpdaters (*magField*, *elecField*, *DELTA*, *lb*, *ub*)

perfectDispersionFaradayUpdaters (*magField*, *elecField*, *DELTA*, *lb*, *ub*, *updaterName*)

A macro to define basic updaters for all 3 components. If `updaterName` is not provided there will be fieldUpdaters called BxUpdater, ByUpdater and BzUpdater, which should be used in place of the regular Yee Faraday update in the `updateSteps`.

Parameters

- **magField** – Name of the magnetic field.
- **elecField** – Name of the electric field.
- **DELTA** – The length of one cell.
- **lb** – The lower bounds of the updater (A vector i.e. [0 0 0]).
- **ub** – The upper bounds of the updater (A vector i.e. [NX NY NZ]).

perfDispPoissonSolver Macro

`perfDispPoissonSolver` (*factor*, *beamBeta*, *boundary*, *DELTA*, *DXI*, *DYI*, *DZI*)

Define an updater for a controlled dispersion Poisson solve. This is used for field initialization.

Parameters

- **`factor`** – Factor to multiply the offsets by.
- **`beamBeta`** – Normalized beam velocity.
- **`boundary`** – Name of the grid boundary.
- **`DELTA`** – The length of one cell.
- **`DXI`** – 1/DX.
- **`DYI`** – 1/DY.
- **`DZI`** – 1/DZ.

perfDispDx Macro

`perfDispDx` (*DXI*, *beamBeta*, *rowIdx*, *colIdx*)

Define stencils for (negative) controlled dispersion forward derivatives in the moving frame. For the lab frame, set beamBeta = 0. This macro is for the X-direction. It does require the variables DELTA, DYI and DZI to all be defined elsewhere in the simulation.

Parameters

- **`DXI`** – 1/DX.
- **`beamBeta`** – Normalized beam velocity.
- **`rowIdx`** – The row of the matrix to be filled.
- **`colIdx`** – The column of the matrix to be filled.

perfDispDy Macro

`perfDispDy` (*DYI*, *rowIdx*, *colIdx*)

Define stencils for (negative) controlled dispersion forward derivatives in the moving frame. This macro is for the Y-direction. It requires the variables DELTA, DXI and DZI to all be defined elsewhere in the simulation.

Parameters

- **`DYI`** – 1/DY.
- **`rowIdx`** – The row of the matrix to be filled.
- **`colIdx`** – The column of the matrix to be filled.

perfDispDz Macro

`perfDispDx` (*DZI*, *beamBeta*, *rowIdx*, *colIdx*)

Define stencils for (negative) controlled dispersion forward derivatives in the moving frame. This macro is for the Z-direction. It requires the variables DELTA, DYI and DXI to all be defined elsewhere in the simulation.

Parameters

- **DZI** – 1/DZ.
- **rowIdx** – The row of the matrix to be filled.
- **colIdx** – The column of the matrix to be filled.

perfDispPMLs Macro

perfDispPMLs(*OUTLBx*, *OUTLBy*, *OUTLBz*, *OUTUBx*, *OUTUBy*, *OUTUBz*, *INLBx*, *INLBy*, *INLBz*, *INUBx*, *INUBy*, *INUBz*, *elecfield*, *magfield*, *pmlefield*, *pmlbfield*, *lybegin*, *lyend*, *lypml*, *lzbegin*, *lzend*, *lzpml*)

This macro establishes controlled dispersion PML boundary conditions on the YZ faces of the simulation environment. Not that it does require the variables SIGMA_MAX, KAPPA_MAX, PML_EXP, DELTA to be defined elsewhere in the simulation.

Parameters

- **OUTLBx** – Lower bound of the simulation in X (grid cells).
- **OUTLBy** – Lower bound of the simulation in Y (grid cells).
- **OUTLBz** – Lower bound of the simulation in Z (grid cells).
- **OUTUBx** – Upper bound of the simulation in X (grid cells).
- **OUTUBy** – Upper bound of the simulation in Y (grid cells).
- **OUTUBz** – Upper bound of the simulation in Z (grid cells).
- **INLBx** – Lower bound in X, excluding the PML region (grid cells).
- **INLBy** – Lower bound in Y, excluding the PML region (grid cells).
- **INLBz** – Lower bound in Z, excluding the PML region (grid cells).
- **INUBx** – Upper bound in X, excluding the PML region (grid cells).
- **INUBy** – Upper bound in Y, excluding the PML region (grid cells).
- **INUBz** – Upper bound in Z, excluding the PML region (grid cells).
- **elecfield** – The name of the electric field.
- **magfield** – The name of the magnetic field.
- **pmlefield** – The name of the electric field of the pml.
- **pmlbfield** – The name of the magnetic field of the pml.
- **lybegin** – The point at which the lower PML layer ends in the y-direction (m).
- **lyend** – The point at which the upper PML layer begins in the y-direction (m).
- **lypml** – The length of a PML layer in the y-direction (m).
- **lzbegin** – The point at which the lower PML layer ends in the z-direction (m).
- **lzbegin** – The point at which the upper PML layer begins in the z-direction (m).
- **lzpml** – The length of a PML layer in the z-direction (m).

pml.mac

This macro file can be imported to an input file with

```
$ import pml
```

This macro file is available to all packages.

This macro file applies a *perfectly matched layer* (PML) in the slab specified. PML's provide boundary conditions for the Yee algorithm that allows outgoing waves to leave without reflections (ideally).

There are two versions of this macro:

- pml is used for computations on a CPU
- gpupml is used for computations on a GPU

They retain the exact same functionality, however GPU and CPU macros cannot be mixed.

Note: PMLs fail to be reflectionless for some materials, including photonic crystals. It is recommended to use the Matched Absorbing Layer (MAL) instead. PMLs may also fail when combined with other active boundary conditions, including but not limited to; ports, particles at the boundary, or structures that are not normal to the boundary. See *Perfectly Matched Layer* in VsimReference. For additional options within Text Setup, see the *PmlRegion* section of the VSim Reference Manual.

applyPML Macro

applyPML(*elecfield*, *magfield*, *nufieldname*, *nxl*, *nyl*, *nzl*, *nxu*, *nyu*, *nzu*, *pmldir*, *pmlsign*, *startval*, *delta*, *namebn*, *nameen*, *nameed*, *namebd*)

OR

applyPML(*elecfield*, *magfield*, *nufieldname*, *nxl*, *nyl*, *nzl*, *nxu*, *nyu*, *nzu*, *pmldir*, *pmlsign*, *startval*, *delta*, *namebn*, *nameen*, *nameed*, *namebd*, *dumpper*)

OR

applyPML(*elecfield*, *magfield*, *nufieldname*, *nxl*, *nyl*, *nzl*, *nxu*, *nyu*, *nzu*, *pmldir*, *pmlsign*, *startval*, *delta*, *namebn*, *nameen*, *nameed*, *namebd*, *dumpper*, *verb*)

Parameters

- **elecfield** – Name of the electric field.
- **magfield** – Name of the magnetic field.
- **nufieldname** – Name of the field to store nu profile.
- **nxl** – Lower bound in the x direction.
- **nyl** – Lower bound in the y direction.
- **nzl** – Lower bound in the z direction.
- **nxu** – Upper bound in the x direction.
- **nyu** – Upper bound in the y direction.
- **nzu** – Upper bound in the z direction.
- **pmdir** – 0, 1, or 2 for the x, y, or z direction of the PML.
- **pmlsign** – The sign of the outgoing direction.

- **startval** – Any offset of the grid start position in the direction of the PML (XSTART, YSTART, ZSTART).
- **delta** – The grid size in the PML direction.
- **namebn** – A name for the bfield numerator damping update.
- **nameen** – A name for the efield numerator damping update.
- **nameed** – A name for the efield denominator damping update.
- **namebd** – A name for the bfield denominator damping update.
- **dumpper** – The dumpPeriod, if given.
- **verb** – The verbosity, if given.

Note: One must use update step names that interleave properly with whatever Yee updater one is using. For the Yee macro in the em.mac file, the updates are:

step1.0 - the first half B update

step2.0 - the full E update

step3.0 - the second half B update

So step0.5, step1.5, step2.5, and step3.5 are good choices for the four update names. The following line must be added to the MultiField block to insure the steps occur in the correct order.

updateStepOrder = [step0.5 step1.0 step1.5 step2.0 step2.5 step3.0 step3.5]

port.mac

This macro file can be imported to an input file with `$ import port`.

This macro file is available to the VSimEM or VSimMD packages.

A tuned phase-velocity port is an open or outgoing boundary condition. Waves traveling at exactly the specified phase velocity of the boundary will exit the simulation with no reflection at all. Waves traveling at other phase velocities will partially exit and partially reflect, with a power reflection coefficient of $\rho = (v_{p,wave} - v_{p,bc})^2/(v_{p,wave} + v_{p,bc})^2$.

An incident wave can be added as an option. If the incident wave travels at exactly the specified phase velocity, the wave will have exactly the specified form. If the incident wave has a different phase velocity, then its normalization will be not be exact. The incident wave can optionally be adjusted for feedback.

This boundary condition is subject to slow-time instability if there is metallic structure and associated fringe or near fields too close to the boundary. For this reason, the boundary condition works best if used to terminate a length of smooth-walled waveguide, or to bound a radiation type simulation, where the boundary is sufficiently distant from the radiation source. The boundary is presently designed for cartesian dimensions.

Ports to be used with Macro generated multifields

addPortBoundary Macro

addPortBoundary (*position*, *VPHASE*)

This macro is used if adding a port boundary in conjunction with a macro generated multifield. It is not compatible with user generated multifields.

Parameters

- **position** – Which face to apply the port boundary, specified by lowerX,lowerY,lowerZ,upperX,upperY or upperZ.
- **vphase** – Phase velocity tuning parameter, in meters/second. For example, $v_p = \frac{c}{\sqrt{(1-(f_{co}/f)^2)}}$ in a waveguide.

addIncidentWaveAtPort Macro

addIncidentWaveAtPort (*position*, *VPHASE*, *ICMP*, *FEEDBACKOPT*, *STFUNCIN*)

This macro is designed to add a wave launcher which at a speciied phase-velocity on a port boundary. This is primarily used for driving waveguides. It is intended to be used with a macro generated multifeld.

Parameters

- **position** – Either lowerX,lowerY,lowerZ,upperX,upperY or upperZ. The wave launcher will be applied to the specified face.
- **vphase** – Phase velocity tuning parameter, in meters/second. For example, $v_p = \frac{c}{\sqrt{(1-(f_{co}/f)^2)}}$ in a waveguide.
- **icmp** – Incident wave's component of the electric field. In many cases, both transverse components of the incident wave are nonzero, in which case there will be two invocations of addIncidentWaveAtBoundaryPort, one for each component.
- **feedbackopt** – If no feedback then `feedbackopt` = 0, or the name of the **History** providing the feedback factor. See *historySTFunc*.
- **stfuncin** – The function *expression (STFunc)* providing the incident wave. This is often defined using another macro. This expression will be evaluated at negative time, so it is common to use max(t,0.0) for time, to insure a zero value at negative times.

Example

```
$ import yee
$ import port

# Add the port
addPortBoundary(lowerX,vphase(FREQ_CENTER))

# Port in the wave
addIncidentWaveAtPort(lowerX,vphase(FREQ_CENTER),1,0,CURR_
↪DENS*signalOuter(t)*waveguideEyProfile(y,z))

# Add MALs on top bottom and sides
addMALBoundary(lowerY,2*WAVELENGTH)
addMALBoundary(upperY,2*WAVELENGTH)

saveEmField()
```

Ports to be used with user defined multifields

If using a user defined multifield, call the first macro before the Ampere UpdateStep, and the second macro after the Ampere UpdateStep, as the names imply. Use of this boundary requires that the Ampere updater properly omit updating of the electric fields at the domain limits. The `contractFromBottomInNonComponentDir` = 1 being critical to this end. These macros assume the **MultiField** setup for electromagnetics is being used.

applyPortAtBoundary_beforeAmpere Macro

applyPortAtBoundary_beforeAmpere (*idir*, *lowerupper*, *vphase*, *dgrid*, *dt*, *efield*, *nx*, *ny*, *nz*)

Defines all of the elements for tuned phase-velocity port that need to be applied before the Ampere UpdateStep. The applyPortAtBoundary_beforeAmpere macro is located in the file `port.mac`.

applyPortAtBoundary_afterAmpere Macro

applyPortAtBoundary_afterAmpere (*idir*, *lowerupper*, *vphase*, *dgrid*, *dt*, *efield*, *nx*, *ny*, *nz*)

Defines all of the elements for tuned phase-velocity port that need to be applied after the Ampere UpdateStep. The applyPortAtBoundary_afterAmpere macro is located in the file `port.mac`.

addIncidentWaveAtBoundaryPort Macro

addIncidentWaveAtBoundaryPort (*idir*, *lowerupper*, *vphase*, *dgrid*, *dt*, *efield*, *nx*, *ny*, *nz*, *imcp*, *feedbackopt*, *stfuncin*)

Defines all of the elements to add an incident wave launcher at a tuned phase-velocity port boundary. The addIncidentWaveAtBoundaryPort is located in the file `port.mac`.

applyPortAtBoundary_beforeAmpere / applyPortAtBoundary_afterAmpere / addIncidentWaveAtBoundaryPort Macro Parameters

param idir 0,1,2 for the direction, x, y, or z, of the outgoing wave.

param lowerupper 0,1 for the lower plane or upper plane, respectively.

param vphase Phase velocity tuning parameter, in meters/second. For example, $v_p = \frac{c}{\sqrt{(1-(f_{co}/f)^2)}}$ in a waveguide.

param dgrid Cell size, e.g., dx, dy, or dz, in the direction of idir, in front of boundary, in meters.

param dt Time step, in seconds.

param efield Name of the Yee (edge) electric field use in the Ampere updater.

param nx Number of cells in the x direction.

param ny Number of cells in the x direction.

param nz Number of cells in the x direction.

param icmp Incident wave's component of the electric field. In many cases, both transverse components of the incident wave are nonzero, in which case there will be two invocations of addIncidentWaveAtBoundaryPort, one for each component.

param feedbackopt If no feedback then `feedbackopt` = 0, or the name of the **History** providing the feedback factor. See *historySTFunc*.

param stfuncin The function *expression (STFunc)* providing the incident wave. This is often defined using another macro. This expression will be evaluated at negative time, so it is common to use max(t,0.0) for time, to insure a zero value at negative times.

Example

```
applyPortAtBoundary_beforeAmpere(0,1,vphaseWave,DX,DT,edgeE,NX,NY,NZ)
<UpdateStep AmpereStep>
  toDtFrac = 1.0
  updaters = [yeeAmpere]
  messageFields = [edgeE]
</UpdateStep>
applyPortAtBoundary_afterAmpere(0,1,vphaseWave,DX,DT,edgeE,NX,NY,NZ)
addIncidentWaveAtBoundaryPort(0,1,vphaseWave,DX,DT,edgeE,NX,NY,NZ,1,0,inputWave(x,y,z,
↪t))
```

requiredBlocks.mac

This macro file can be imported to an input file with `$ import requiredBlocks`.

This macro file is available to all packages.

This macro file creates the required plocks of the grid and decomp blocks as well as the required global variables.

Create Required Blocks Macro

createRequiredBlocks (*dim*, *precision*, *timestep*, *numsteps*, *dumpperiod*, *nx*, *ny*, *nz*, *lx*, *ly*, *lz*, *sx*, *sy*, *sz*)

Defines the dimensionality,floattype,dt,nsteps,dumpPeriodicity variables as well as the Grid and Decomp Blocks. file `requiredBlocks.mac`.

Parameters

- **dim** – Dimensionality of the simulation domain.
- **precision** – floattype of the simulation.
- **timestep** – dt for the simulation.
- **numsteps** – Number of timesteps to run the simulation.
- **dumpperiod** – The frequency of data dumps.
- **nx** – Number of cells in the x direction.
- **ny** – Number of cells in the y direction.
- **nz** – Number of cells in the z direction.
- **lx** – Length of domain in the x direction.
- **ly** – Length of domain in the y direction.
- **lz** – Length of domain in the z direction.
- **sx** – Start of simulation in the x direction (m).
- **sy** – Start of simulation in the y direction (m).
- **sz** – Start of simulation in z direction (m).

Standard Time Step Macro

`standardTimeStep` (*dmfrac*, *timestepFactor*)

Calculates the length of a time step, according to the Courant-Levy-Friedrichs condition. Returns a value, usage ($ DT = standardTimeStep(dmfrac,timestepFactor))

Parameters

- **`dmfrac`** – DeyMittraFraction, 1. if no grid boundaries are present.
- **`timestepFactor`** – Safety factor by which to modify the timestep. Typically .9995.

scatteringBox.mac

This macro file can be imported to an input file with `$ import scatteringBox`.

This macro file is available to the VSimEM package.

This macro file can be used to define a scattering box. In a scattering box a wave can be launched from an angle, and the only waves that will leave the box are those which have been scattered off an object under test. In this macro their are two main versions. The first, listed below is meant to work with a macro generated multifield. The other is actually a set of three meant to work with a user generated multifield.

There are two versions of this macro, one, scatteringBox, is for use when doing computations on a CPU, the other, scatteringBoxGPU, is used for computations on a GPU. They retain the exact same functionality, however GPU and CPU macros cannot be mixed.

Scattering Box Macro

This particular macro comes in 3 varieties. There is scatteringBox, scatteringBox1Dielectric and scatteringBox2Dielectric. scatteringBox is meant to be used if the simulation environment is a vacuum. scatteringBox1Dielectric is to be used if there is a uniform dielectric medium in the simulation space. scatteringBox2Dielectric is to be used if there are two seperate dielectrics in the simulation environment, and it is desired for the effects on EM waves caused by both dielectrics be confined to the scattering wave box. They all use slightly different arguments, and have some constraints on how the dielectric is defined delineated below.

`scatteringBox` (*name*, *AMPLITUDE*, *FREQUENCY*, *WAVE_ANGLE_X*, *WAVE_ANGLE_Y*, *WAVE_ANGLE_Z*, *RISE_TIME*, *lx*, *ly*, *lz*, *ux*, *uy*, *uz*)

This macro will create a scattering box. The wave will be launched at a user specified angle, with a given frequency, amplitude and rise time. The size of the box is determine by the user (in grid cells). In this version of the macro it is assumed that the background permittivity is vacuum.

`scatteringBox1Dielectric` (*name*, *AMPLITUDE*, *FREQUENCY*, *WAVE_ANGLE_X*, *WAVE_ANGLE_Y*, *WAVE_ANGLE_Z*, *RISE_TIME*, *lx*, *ly*, *lz*, *ux*, *uy*, *uz*, *EPSILON_1*)

This macro will create a scattering box. The wave will be launched at a user specified angle, with a given frequency, amplitude and rise time. The size of the box is determine by the user (in grid cells). This macro also will allow for a uniform dielectric in the simulation space. This dielectric must be defined using the addDielectric macro and must be uniform across the simulation space. The electromagnetic effects of this dielectric will be confined to the scattering box. This is very useful if it is desired to observe only the waves scattered due to an object buried in dielectric. The Epsilon value of the dielectric must be defined.

`scatteringBox2Dielectric` (*name*, *AMPLITUDE*, *FREQUENCY*, *WAVE_ANGLE_X*, *WAVE_ANGLE_Y*, *WAVE_ANGLE_Z*, *RISE_TIME*, *lx*, *ly*, *lz*, *ux*, *uy*, *uz*, *EPSILON_1*, *EPSILON_2*, *DIELECTRIC_1*, *DIELECTRIC_2*, *BOUNDARY*, *EPS_INTERSECTION*)

This macro will create a scattering box. The wave will be launched at a user specified angle, with a given frequency, amplitude and rise time. The size of the box is determine by the user (in grid cells). This macro supports the ability to confine the electromagnetic effects of a wave impacting two dielectrics within the scattering wave box. For example the RCS of a ship in the ocean can be taken with the effects of waves reflecting off the ocean cancelled out. You must seperately define the dielectric, using the addDielectric macro. At the moment support is only provided for the dielectric intersection to exist on the Y-Z plane, and it must be uniform along the Y-Z plane.

Parameters

- **`name`** – Name of the scattering box.
- **`AMPLITUDE`** – Amplitude of the wave launched within the scattering box.
- **`FREQUENCY`** – Frequency of the wave launched within the scattering box.
- **`WAVE_ANGLE_X`** – Angle of the launched wave as measured off the X axis (degrees).
- **`WAVE_ANGLE_Y`** – Angle of the launched wave as measured off the Y axis (degrees).
- **`WAVE_ANGLE_Z`** – Angle of the launched wave as measured off the Z axis (degrees).
- **`RISE_TIME`** – Rise time of the launched wave.
- **`lx`** – Lower bound of the scattered wave in the X-direction (grid cells).
- **`ly`** – Lower bound of the scattered wave in the Y-direction (grid cells).
- **`lz`** – Lower bound of the scattered wave in the Z-direction (grid cells).
- **`ux`** – Upper bound of the scattered wave in the X-direction (grid cells).
- **`uy`** – Upper bound of the scattered wave in the Y-direction (grid cells).
- **`uz`** – Upper bound of the scattered wave in the Z-direction (grid cells).
- **`EPSILON_1`** – Epsilon value of the 1st dielectric. (Only necessary for scatteringBox1Dielectric and scatteringBox2Dielectric)
- **`EPSILON_2`** – Epsilon value of the 2nd dielectric. (Only necessary for scatteringBox2Dielectric)
- **`DIELECTRIC_1`** – Function defining the spatial profile of the first dielectric. (Only necessary for scatteringBox2Dielectric)
- **`DIELECTRIC_2`** – Function defining the spatial profile of the second dielectric. (Only necessary for scatteringBox2Dielectric)
- **`BOUNDARY`** – Function describing the spatial profile of the first and second dielectric. It is important that the dielectric constant at the intersection be an average of the first and second dielectrics. (Only necessary for scatteringBox2Dielectric)
- **`EPS_INTERSECTION`** – X-coordinate of the intersection of the first and second dielectrics. (Meters) (Only necessary for scatteringBox2Dielectric)

If using a user generated multifield, the following macros are to be used. These macros do not support two dielectrics within the scattering box.

The user starts by making three setup calls, in the following order.

`fullScatteringBox_Region` (*xbgn*, *ybgn*, *zbgn*, *dx*, *dy*, *dz*, *dt*, *nx*, *ny*, *nz*, *nxBgnBox*, *nyBgnBox*, *nzBgnBox*, *nxEndBox*, *nyEndBox*, *nzEndBox*, *edgeDname*, *faceBname*)

`fullScatteringBox_Wave` (*freq*, *cosx*, *cosy*, *cosz*, *ExR*, *EyR*, *EzR*, *ExI*, *EyI*, *EzI*, *epsRelBackground*)

`fullScatteringBox_Updaters` ()

The first two, Region and Wave, can be anywhere in the input file. The third, Updaters, must be inside the MultiField block.

The user must make this call just before the Ampere UpdateStep: fullScatteringBox_ApplyB().

The user must make this call just after the Ampere UpdateStep: fullScatteringBox_ApplyD().

Full Scattering Box Region Macro

`fullScatteringBox_Region` (*xbgn*, *ybgn*, *zbgn*, *dx*, *dy*, *dz*, *dt*, *nx*, *ny*, *nz*, *nxBgnBox*, *nyBgnBox*, *nzBgnBox*, *nxEndBox*, *nyEndBox*, *nzEndBox*, *edgeDname*, *faceBname*)

This macro establishes the region of the scattering box.

Parameters

- **`xbgn`** – The start point of the simulation in the X direction.
- **`ybgn`** – The start point of the simulation in the Y direction.
- **`zbgn`** – The start point of the simulation in the Z direction.
- **`dx`** – Length of one cell in the X direction.
- **`dy`** – Length of one cell in the Y direction.
- **`dz`** – Length of one cell in the Z direction.
- **`dt`** – Length of one timestep.
- **`nx`** – Number of cells in the X direction in the simulation.
- **`ny`** – Number of cells in the Y direction in the simulation.
- **`nz`** – Number of cells in the Z direction in the simulation.
- **`nxBgnBox`** – The start point of the scattering box in the X direction (grid cells).
- **`nyBgnBox`** – The start point of the scattering box in the Y direction (grid cells).
- **`nzBgnBox`** – The start point of the scattering box in the Z direction (grid cells).
- **`nxEndBox`** – The end point of the scattering box in the X direction (grid cells).
- **`nyEndBox`** – The end point of the scattering box in the Y direction (grid cells).
- **`nzEndBox`** – The end point of the scattering box in the Z direction (grid cells).
- **`edgeDname`** – Name of the electric field used in the Ampere update.
- **`faceBname`** – Name of the Magnetic field used in the Ampere update.

Full Scattering Box Wave Macro

`fullScatteringBox_Wave` (*FREQUENCY*, *WAVE_ANGLE_X*, *WAVE_ANGLE_X*, *WAVE_ANGLE_X*, *ExR*, *EyR*, *EzR*, *ExI*, *EyI*, *EzI*, *EPSILON*)

This macro sets up the wave to be launched within the scattering box.

Parameters

- **`FRQUENECY`** – The frequency of the launched wave.
- **`WAVE_ANGLE_X`** – Angle of the launched wave as measured off the X axis (degrees).
- **`WAVE_ANGLE_Y`** – Angle of the launched wave as measured off the Y axis (degrees).

- **WAVE_ANGLE_Z** – Angle of the launched wave as measured off the Z axis (degrees).
- **ExR** – The real part of the amplitude in the X direction.
- **EyR** – The real part of the amplitude in the Y direction.
- **EzR** – The real part of the amplitude in the Z direction.
- **ExI** – The imaginary part of the amplitude in the X direction. (Necessary for circular polarization)
- **EyI** – The imaginary part of the amplitude in the Y direction. (Necessary for circular polarization)
- **EzI** – The imaginary part of the amplitude in the Z direction. (Necessary for circular polarization)
- **EPSILON** – The epsilon value within the scattering box. (1 for vacuum/air)

Full Scattering Box Updaters Macro

fullScatteringBox_Updaters()

This macro sets up the updaters necessary for the scatttering box to function. It accepts no parameters but *must* be placed within the multifields block.

Full Scattering Box Apply B Macro

fullScatteringBox_ApplyB()

This macro must be called immediately before the Ampere UpdateStep.

Full Scattering Box Apply D Macro

fullScatteringBox_ApplyD()

This macro must be called immediately after the Ampere UpdateStep.

smooth.mac

This macro file can be imported to an input file with `$ import smooth`.

This macro file is available to all packages, but was created for the VSimPA examples.

This macro file provides some commands useful for smoothing fields like the current density field, which can help with mitigating numerical dispersion in the EM (Yee) field updates.

fourPassAndReloc Macro

fourPassAndReloc(*name*, *J*, *J1*, *J2*, *S*, *nxlo*, *nylo*, *nzlo*, *nxhi*, *nyhi*, *nzhi*, *dir*, *dtfrac*)

Single-pass 1-2-1 digital smoothing of all J components (3-d vector): Smooth the first aux field; Write to the second.

Parameters

- **name** – Name for the updater.
- **J** – The depField to be smoothed.

- **J1** – The first auxiliary field in which J is copied.
- **J2** – The second auxiliary field used for smoothing. J2 should be used as an input of the yeeAmpere updater.
- **S** – Stride of the stencil - usually 1.
- **nxlo** – Lower bound in X.
- **nylo** – Lower bound in Y.
- **nzlo** – Lower bound in Z.
- **nxhi** – Upper bound in X.
- **nyhi** – Upper bound in Y.
- **nzhi** – Upper bound in Z.
- **dir** – The physical direction where the smoothing operator is applied (0: smoothing in the x direction, 1:y, 2:z).
- **dtfrac** – The fraction of the time step at which the currents are updated.

fourPassAndRelocScalar Macro

fourPassAndRelocScalar(*name*, *J*, *J1*, *J2*, *S*, *nxlo*, *nylo*, *nzlo*, *nxhi*, *nyhi*, *nzhi*, *dir*, *dtfrac*)

Single-pass 1-2-1 digital smoothing for a 1d-vector: Smooth the first aux field; write to the second.

Parameters

- **name** – Name for the updater.
- **J** – The depField to be smoothed.
- **J1** – The first auxiliary field in which J is copied.
- **J2** – The second auxiliary field used for smoothing. J2 should be used as an input of the yeeAmpere updater.
- **S** – Stride of the stencil - usually 1.
- **nxlo** – Lower bound in X.
- **nylo** – Lower bound in Y.
- **nzlo** – Lower bound in Z.
- **nxhi** – Upper bound in X.
- **nyhi** – Upper bound in Y.
- **nzhi** – Upper bound in Z.
- **dir** – The physical direction where the smoothing operator is applied (0: smoothing in the x direction, 1:y, 2:z).
- **dtfrac** – The fraction of the time step at which the currents are updated.

Example usage

In order to use,

- Include the macro `$import smooth`

- Add two dummy fields of the same type as the field you want to smooth (eg `Jaux`, `Jsmooth`)
- Before the Ampere update step, apply the macro: `fourPassAndReloc(smthBeam, Jdep, Jaux, Jsmooth, 1, 0, 0, 0, NX, NY, NZ, 0, 0.5)` This assumes you have a `vectorDepositor`to record the current density in a field :samp:`Jdep`, modify to your own situation.
- If you are using updateStepOrder, the following should be manually added before the Ampere update `smthBeamcopy`, `smthBeamsmooth1`, `smthBeamsmooth2`, `smthBeamsmooth3`, `smthBeamsmooth4` and `smthBeamreloc`

smoothers.mac

This macro file can be imported to an input file with `$ import smoothers`.

This macro file is available to all packages.

This macro file provides some commands useful for smoothing fields.

copyToAuxJ Macro

`copyToAuxJ`(*name*, *restorefield*, *readfield*, *writefield*)

Copy a 4-vector field to a 3-vector field. This is necessary for smoothing currents, as one must copy to a field with ordinary messaging. It does make use of the number of cells in the X,Y and Z directions as NX,NY,NZ

Parameters

- **`name`** – Name for the updater.
- **`restorefield`** – Name of field from which to restore updater time upon restart. The smoothers should be applied to the half-way time of an update step (toDtFrac=0.5). In order for restart to work, Vorpal needs to know the time at a full step, not half. By setting the *restorefield* parameter to a field that is solved at a toDtFrac=1.0, Vorpal can properly restart. A good suggestion is to use an edgeE or faceB field as in the example shown below. If using the yee.mac macro, the fields are named *elecfield* and *magfield* explicitly.
- **`readfield`** – The field to copy from.
- **`writefield`** – The field to write to.

copyFromAuxJ Macro

`copyFromAuxJ`(*name*, *restorefield*, *readfield*, *writefield*)

Copy to a 4-vector field from a 3-vector. After smoothing a field, it may need to be copied into a 4-vector field for the standard EM updates. It makes use of the number of cells in the X,Y and Z directions as NX,NY,NZ

Parameters

- **`name`** – Name for the updater.
- **`restorefield`** – Name of field from which to restore updater time upon restart. The smoothers should be applied to the half-way time of an update step (toDtFrac=0.5). In order for restart to work, Vorpal needs to know the time at a full step, not half. By setting the *restorefield* parameter to a field that is solved at a toDtFrac=1.0, Vorpal can properly restart. A good suggestion is to use an edgeE or faceB field as in the example shown below. If using the yee.mac macro, the fields are named *elecfield* and *magfield* explicitly.
- **`readfield`** – The field to copy from.

- **`writefield`** – The field to write to.

yzsmth Macro

`yzsmth` (*name*, *restorefield*, *updatefield*, *component*, *smfrac*)

Defines a yz-smoother for getting perfect EM dispersion. The matrix is antismoother, but then a solve is done. Requires NX, NY, NZ, DX2, DYI2 DZI2

Parameters

- **`name`** – Name for the updater.
- **`restorefield`** – The field from which the updater time is taken at restore. The smoothers should be applied to the half-way time of an update step (toDtFrac=0.5). In order for restart to work, Vorpal needs to know the time at a full step, not half. By setting the *restorefield* parameter to a field that is solved at a toDtFrac=1.0, Vorpal can properly restart. A good suggestion is to use an edgeE or faceB field as in the example shown below. If using the yee.mac macro, the fields are named *elecfield* and *magfield* explicitly.
- **`updatefield`** – The field to be smoothed.
- **`component`** – The component of the field to be smoothed.
- **`smfrac`** – The amount of smoothing. The value of 1, when applied to the curl of B before adding to E gives perfect dispersion.

xsmth Macro

`xsmth` (*name*, *restorefield*, *readfield*, *writefield*, *comp*, *sfac*, *ixbeg*, *ixend*)

Defines a x-smoother for modifying the EM dispersion. Start and stop values for smoothing are defined, as one may have to do something different at the limits to preserve divergence.

Requires NX, NY, NZ

Parameters

- **`name`** – Name for the updater.
- **`restorefield`** – The field from which the updater time is taken at restore. The smoothers should be applied to the half-way time of an update step (toDtFrac=0.5). In order for restart to work, Vorpal needs to know the time at a full step, not half. By setting the *restorefield* parameter to a field that is solved at a toDtFrac=1.0, Vorpal can properly restart. A good suggestion is to use an edgeE or faceB field as in the example shown below. If using the yee.mac macro, the fields are named *elecfield* and *magfield* explicitly.
- **`readfield`** – The field to be smoothed.
- **`writefield`** – The field into which to put the smoothed result.
- **`comp`** – The component of the field to be smoothed.
- **`sfac`** – The smoothing factor, gives 1-2-1 when the value of this is one.
- **`ixbeg`** – The x cell to begin at.
- **`ixend`** – The x cell to end at.

smooth Macro

smooth (*name*, *restorefield*, *readfield*, *writefield*, *smfacx*, *smfacy*)

Defines an xy-smoother for smoothing in x and y in one fell swoop. Requires NX, NY, NZ

Parameters

- **name** – Name for the updater.
- **restorefield** – The field from which the updater time is taken at restore. The smoothers should be applied to the half-way time of an update step (toDtFrac=0.5). In order for restart to work, Vorpal needs to know the time at a full step, not half. By setting the *restorefield* parameter to a field that is solved at a toDtFrac=1.0, Vorpal can properly restart. A good suggestion is to use an edgeE or faceB field as in the example shown below. If using the yee.mac macro, the fields are named *elecfield* and *magfield* explicitly.
- **readfield** – The field to be smoothed.
- **writefield** – The field into which to put the smoothed result.
- **smfacx** – The smoothing factor for x. One for 1-2-1 smoothing.
- **smfacy** – The smoothing factor for y. One for 1-2-1 smoothing.

smoothyz Macro

smoothyz (*name*, *restorefield*, *readfield*, *writefield*, *smy*, *smz*)

Forward smoother for y and z at once.

Requires NDIM, NX, NY, NZ

Parameters

- **name** – Name for the updater.
- **restorefield** – The field from which the updater time is taken at restore. The smoothers should be applied to the half-way time of an update step (toDtFrac=0.5). In order for restart to work, Vorpal needs to know the time at a full step, not half. By setting the *restorefield* parameter to a field that is solved at a toDtFrac=1.0, Vorpal can properly restart. A good suggestion is to use an edgeE or faceB field as in the example shown below. If using the yee.mac macro, the fields are named *elecfield* and *magfield* explicitly.
- **readfield** – The field to be smoothed.
- **writefield** – The field into which to put the smoothed result.
- **smy** – The smoothing factor to use for y. Unity gives standard 1/4, 1/2, 1/4 smoothing.
- **smz** – The smoothing factor to use for z. Unity gives standard 1/4, 1/2, 1/4 smoothing.

copyxshft Macro

copyxshft (*name*, *restorefield*, *readfield*, *writefield*, *comp*, *ixbeg*, *ixbmp*)

Updater to copy a component from another cell displaced in x

Parameters

- **name** – Name for the updater.

- **restorefield** – The field from which the updater time is taken at restore. The smoothers should be applied to the half-way time of an update step (toDtFrac=0.5). In order for restart to work, Vorpal needs to know the time at a full step, not half. By setting the *restorefield* parameter to a field that is solved at a toDtFrac=1.0, Vorpal can properly restart. A good suggestion is to use an edgeE or faceB field as in the example shown below. If using the yee.mac macro, the fields are named *elecfield* and *magfield* explicitly.
- **readfield** – The field to be smoothed.
- **writefield** – The field into which to put the smoothed result.
- **ixbeg** – The cell in the x direction that is copied into.
- **ixbmp** – The x index displacement to the cell that is copied from.

Example use of the smoothers

```
Current smoothing.  Copy J, smooth, copy back.

Copy first.  Cannot smooth directly on SumRhoJ as not valid in guard cells.
copyToAuxJ(copyjbeg, faceB, SumRhoJ, auxRhoJ1)

y-z smoothing of current goes over entire region.
smoothyz(jsmoothyz, faceB, auxRhoJ1, auxRhoJ2, 1.0, 1.0)

x smoothing by component basis required for charge conservation.
y and z components copied in place in last cells prior to smoothing.
Gives 3/4, 1/4 in last cells.
copyxshft(copyjydn, faceB, auxRhoJ2, auxRhoJ2, 1, -1,  1)
copyxshft(copyjyup, faceB, auxRhoJ2, auxRhoJ2, 1, NX, -1)
copyxshft(copyjzdn, faceB, auxRhoJ2, auxRhoJ2, 2, -1,  1)
copyxshft(copyjzup, faceB, auxRhoJ2, auxRhoJ2, 2, NX, -1)

x components in first and last cell copied over, not smoothed, for charge
conservation.
copyxshft(copyjxdn, faceB, auxRhoJ2, auxRhoJ1, 0, 0,  0)
copyxshft(copyjxup, faceB, auxRhoJ2, auxRhoJ1, 0, $NX-1$, 0)

x component not smoothed in first or last cell for charge conservation.
xsmth(jxsmoothx, faceB, auxRhoJ2, auxRhoJ1, 0, 1., 1, $NX-1$)

y and z components smoothed throughout interior.
xsmth(jysmoothx, faceB, auxRhoJ2, auxRhoJ1, 1, 1., 0, NX)
xsmth(jzsmoothx, faceB, auxRhoJ2, auxRhoJ1, 2, 1., 0, NX)

Copy back. DO NOT message, as SumRhoJ messaging adds in from other ranks.
copyFromAuxJ(copyjend, faceB, auxRhoJ1, SumRhoJ)
```

solverbcs.mac

This macro file can be imported to an input file with `$ import solverbcs`.

This macro file is available to all packages.

This macro file sets up boundary conditions for electrostatic simulations.

dirichletBC Macro

dirichletBC (*name*, *md*, *lbx*, *lby*, *lbz*, *ubx*, *uby*, *ubz*, *comp*, *exp*, *scl*)

This macro establishes a Dirichlet boundary condition for electrostatic solvers.

Parameters

- **name** – Name of the boundary condition.
- **md** – The minimum dimension needed for this boundary condition.
- **lbx** – The lower-x bound over which this is applied.
- **lby** – The lower-y bound over which this is applied.
- **lbz** – The lower-z bound over which this is applied.
- **ubx** – The upper-x bound over which this is applied.
- **uby** – The upper-y bound over which this is applied.
- **ubz** – The upper-z bound over which this is applied.
- **comp** – The component (row of the matrix) being filled.
- **exp** – The expression for the function defining the value.
- **scl** – The scaling applied to all matrix and vector entries.

neumannBC Macro

neumannBC (*name*, *md*, *lbx*, *lby*, *lbz*, *ubx*, *uby*, *ubz*, *comp*, *exp*, *dir*, *pm*, *scl*)

This macro establishes a Neumann boundary condition for electrostatic solvers.

Parameters

- **name** – Name of the boundary condition.
- **md** – The minimum dimension needed for this boundary condition.
- **lbx** – The lower-x bound over which this is applied.
- **lby** – The lower-y bound over which this is applied.
- **lbz** – The lower-z bound over which this is applied.
- **ubx** – The upper-x bound over which this is applied.
- **uby** – The upper-y bound over which this is applied.
- **ubz** – The upper-z bound over which this is applied.
- **comp** – The component (row of the matrix) being filled.
- **exp** – The expression for the function defining the value.
- **dir** – 0, 1 or 2 for if the boundary condition is on the X, Y, or Z, boundary.
- **pm** – ‘+1’ for the low side, ‘-1’ for the high side.
- **scl** – The scaling applied to all matrix and vector entries.

dirichletFromField Macro

dirichletFromField(*name*, *md*, *lbx*, *lby*, *lbz*, *ubx*, *uby*, *ubz*, *comp*, *field*, *fcomp*, *scl*)

This macro establishes a Dirichlet boundary condition for electrostatic solvers from a field.

Parameters

- **name** – Name of the boundary condition.
- **md** – The minimum dimension needed for this boundary condition.
- **lbx** – The lower-x bound over which this is applied.
- **lby** – The lower-y bound over which this is applied.
- **lbz** – The lower-z bound over which this is applied.
- **ubx** – The upper-x bound over which this is applied.
- **uby** – The upper-y bound over which this is applied.
- **ubz** – The upper-z bound over which this is applied.
- **comp** – The component (row of the matrix) being filled.
- **field** – The field name.
- **fcomp** – The component of the field to be used.
- **scl** – The scaling applied to all matrix and vector entries.

specularBox.mac

This macro file can be imported to an input file with `$ import specularBox`.

This macro file is available to all packages.

reflectingBox(*LEFTPLATE*, *RIGHTPLATE*, *FRONTPLATE*, *BACKPLATE*, *BOTTOMPLATE*, *TOPPLATE*)

Sets up fully reflecting boxes in any dimension. This macro does not take as arguments, but needs NX, NY, NZ, XSTART, YSTART, ZSTART, DX, DY, DZ.

Parameters

- **LEFTPLATE** – Position of left (in x) reflecting surface.
- **RIGHTPLATE** – Position of right (in x) reflecting surface.
- **FRONTPLATE** – Position of front (in y) reflecting surface.
- **BACKPLATE** – Position of back (in y) reflecting surface.
- **BOTTOMPLATE** – Position of bottom (in z) reflecting surface.
- **TOPPLATE** – Position of top (in z) reflecting surface.

reflectingBoxCylindrical(*ZLOWER*, *RLOWER*, *ZUPPER*, *RUPPER*)

Sets up fully reflecting boxes in any dimension for simulations performed in cylindrical coordinates. Currently this reflecting box can only be set up along the edges of the simulation space. This macro does not take as arguments, but needs NZ, NR, NPHI

This macro is encrypted and available with the VSimBase license.

Parameters

- **ZLOWER** – Position of the reflecting surface on the lower Z surface (m). If left blank a reflecting surface will not be placed here.
- **RLOWER** – Position of the reflecting surface on the lower R surface (m). If left blank a reflecting surface will not be placed here.
- **ZUPPER** – Position of the reflecting surface on the upper Z surface (m). If left blank a reflecting surface will not be placed here.
- **RUPPER** – Position of the reflecting surface on the upper R surface (m). If left blank a reflecting surface will not be placed here.

statics.mac

This macro file can be imported to an input file with `$ import statics`.

This macro file is available to all packages.

This poorly-named macro file aids in the creation of Laplacians used in electro- and magneto-statics problems.

Laplacian Macro

laplacian (*name*, *md*, *lbx*, *lby*, *lbz*, *ubx*, *uby*, *ubz*, *comp*, *fld*, *fcomp*, *dcoeff*, *xcoeff*, *ycoeff*, *zcoeff*)

This macro is encrypted and available with the VSimBase License.

Parameters

- **nam** – The base name of the boundary condition.
- **m** – The minimum dimension needed for this boundary condition.
- **lb** – The lower bound over which this is applied in the X-direction.
- **lb** – The lower bound over which this is applied in the Y-direction.
- **lb** – The lower bound over which this is applied in the Z-direction.
- **ub** – The upper bound over which this is applied in the X-direction.
- **ub** – The upper bound over which this is applied in the Y-direction.
- **ub** – The upper bound over which this is applied in the Z-direction.
- **com** – The component (row of the matrix) being filled.
- **fl** – The name of the field used to fill the associated source (RHS) vector.
- **fcom** – The component of the field.
- **dcoef** – Diagonal coefficient.
- **xcoef** – Coefficient for x offset stencils.
- **ycoef** – Coefficient for y offset stencils.
- **zcoef** – Coefficient for z offset stencils.

STFunc Laplacian Macro

stFuncLaplacian (*name*, *md*, *lbx*, *lby*, *lbz*, *ubx*, *uby*, *ubz*, *comp*, *fld*, *fcomp*, *xcoeff*, *ycoeff*, *zcoeff*, *func*)

Laplacian whose matrix entries are calculated using an STFunc.

This macro is encrypted and available with the VSimBase License.

Parameters

- **nam** – The base name of the boundary condition.
- **m** – The minimum dimension needed for this boundary condition.
- **lb** – The lower bound over which this is applied in the Y-direction.
- **lb** – The lower bound over which this is applied in the Z-direction.
- **ub** – The upper bound over which this is applied in the X-direction.
- **ub** – The upper bound over which this is applied in the Y-direction.
- **ub** – The upper bound over which this is applied in the Z-direction.
- **com** – The component (row of the matrix) being filled.
- **fl** – The name of the field used to fill the associated source (RHS) vector.
- **fcom** – The component of the field.
- **dcoef** – Diagonal coefficient.
- **xcoef** – Coefficient for x offset stencils.
- **ycoef** – Coefficient for y offset stencils.
- **zcoef** – Coefficient for z offset stencils.
- **fun** – The expression for the function defining the value.

Coord Prod Laplacian Macro

coordProdLaplacian (*name*, *md*, *lbx*, *lby*, *lbz*, *ubx*, *uby*, *ubz*, *comp*, *fld*, *fcomp*, *scl*, *func*)

This macro is for a laplacian on variable (coordProd) grids.

This macro is encrypted and available with the VSimBase License.

Parameters

- **nam** – The base name of the boundary condition.
- **m** – The minimum dimension needed for this boundary condition.
- **lb** – The lower bound over which this is applied in the X-direction.
- **lb** – The lower bound over which this is applied in the Y-direction.
- **lb** – The lower bound over which this is applied in the Z-direction.
- **ub** – The upper bound over which this is applied in the X-direction.
- **ub** – The upper bound over which this is applied in the Y-direction.
- **ub** – The upper bound over which this is applied in the Z-direction.
- **com** – The component (row of the matrix) being filled.
- **fl** – The name of the field used to fill the associated source (RHS) vector.

- **fcom** – The component of the field.
- **sc** – The scaling factor by which all matrix elements will be multiplied.
- **fun** – The expression for the function defining the value.

Curl Curl Macro

CurlCurl (*name*, *md*, *lbx*, *lby*, *lbz*, *ubx*, *uby*, *ubz*, *comp*, *fld*, *fcomp*, *dir*, *pm*, *dcoeff*, *tcoeff1*, *tcoeff2*, *ncoeff1*, *ncoeff2*)

This macro is encrypted and available with the VSimBase License.

Parameters

- **nam** – The base name of the boundary condition.
- **m** – The minimum dimension needed for this boundary condition.
- **lb** – The lower bound over which this is applied in the X-direction.
- **lb** – The lower bound over which this is applied in the Y-direction.
- **lb** – The lower bound over which this is applied in the Z-direction.
- **ub** – The upper bound over which this is applied in the X-direction.
- **ub** – The upper bound over which this is applied in the Y-direction.
- **ub** – The upper bound over which this is applied in the Z-direction.
- **com** – The component (row of the matrix) being filled.
- **fl** – The name of the field used to fill the associated source (RHS) vector.
- **fcom** – The component of the field.
- **di** – The scaling factor by which all matrix elements will be multiplied.
- **p** – The expression for the function defining the value.
- **dcoef** – The expression for the function defining the value.
- **tcoeff** – The expression for the function defining the value.
- **tcoeff** – The expression for the function defining the value.
- **ncoeff** – The expression for the function defining the value.
- **ncoeff2** – The expression for the function defining the value.

yee.mac

This macro file can be imported to an input file with.

:: $ import yee

This macro file is available to all packages.

This macro file contains several powerful functions that allow for an easily reconfigurable multifield to handle simulations with no particles. The macros contained here are incompatible with a user generated multifield.

There is a second version of this macro, yeeGPU. It retains most of the same functionality with some exceptions. The macros available under yeeGPU can be found in yeeGPU-macro

saveEmField Macro

saveEmField()

This macro saves the EmField as well as all current sources and boundary conditions defined using the other function calls. The default value for the name of the field is 'emField'.

saveEmField(*gridBoundary*)

The name of a grid boundary must be specified if it is used in the simulation. The default value for the name of the field is *emField*.

saveEmField(*gridBoundary*, *name*)

In the event two seperate multifields are used in one file, they must be named to be differentiated from each other.

Parameters

- **gridboundary** (*string*) – Name of the grid boundary.
- **name** (*string*) – Name of the multifield.

addParticleFields Macro

addParticleFields()

This macro will add two fields, nodalE and nodalB to the simulation. These fields can then be used in particle species blocks. The macro does not add a J field, depositors or any references to the depositors in the species blocks, so these must be added manually.

addCurrentSource Macro

addCurrentSource(*name*, *polarization*, *profile*, *lx*, *ly*, *lz*, *ux*, *uy*, *uz*)

This macro adds a current source (writing to the J field) with a user defined profile in the user defined update region.

Parameters

- **name** (*string*) – Name of the current source.
- **polarization** (*integer*) – Direction in which the wave is polarized. This is a coordinate system dependent quantity. In Cartesian coordinates 0=x, 1=y, 2=z.
- **profile** – The function that defines the current source's amplitude, frequency and phase shift.
- **lx** (*float*) – Lower bound in the x direction (meters).
- **ly** (*float*) – Lower bound in the y direction (meters).
- **lz** (*float*) – Lower bound in the z direction (meters).
- **ux** (*float*) – Upper bound in the x direction (meters).
- **uy** (*float*) – Upper bound in the y direction (meters).
- **uz** (*float*) – Upper bound in the z direction (meters).

addToCurrentSource Macro

addToCurrentSource (*name*, *polarization*, *profile*, *lx*, *ly*, *lz*, *ux*, *uy*, *uz*)

This macro adds a current source (additive to the J field) with a user defined profile in the user defined update region. The profile expression will be reevaluated and added each time step.

Parameters

- **name** (*string*) – Name of the current source.
- **polarization** (*integer*) – Direction in which the wave is polarized. This is a coordinate system dependent quantity. In Cartesian coordinates 0=x, 1=y, 2=z.
- **profile** (*string*) – An inline expression to be evaluated. This function typically defines the current source's amplitude, frequency and phase shift.
- **lx** (*float*) – Lower bound in the x direction (meters).
- **ly** (*float*) – Lower bound in the y direction (meters).
- **lz** (*float*) – Lower bound in the z direction (meters).
- **ux** (*float*) – Upper bound in the x direction (meters).
- **uy** (*float*) – Upper bound in the y direction (meters).
- **uz** (*float*) – Upper bound in the z direction (meters).

addWaveLauncher Macro

addWaveLauncher (*name*, *polarization*, *profile*, *position*)

addWaveLauncher (*name*, *polarization*, *profile*, *lx*, *ly*, *lz*, *ux*, *uy*, *uz*)

Launches a wave directly into the simulation domain by imposing an E-field on a specified boundary face.

If the wave launcher is to be placed anywhere other than on a domain boundary face, upper and lower bounds (in meters) may be specified instead.

Parameters

- **name** (*string*) – Name of the wave launcher.
- **polarization** (*integer*) – Direction in which the wave is polarized. This is a coordinate system dependent quantity. In Cartesian coordinates 0=x, 1=y, 2=z.
- **profile** (*string*) – An inline expression to be evaluated. This function typically defines the current source's amplitude, frequency and phase shift.
- **position** (*string*) – Face on which the wave launcher will be applied (lowerX, lowerY, lowerZ, upperX, upperY, upperZ).
- **lx** (*float*) – Lower bound in the x direction (meters).
- **ly** (*float*) – Lower bound in the y direction (meters).
- **lz** (*float*) – Lower bound in the z direction (meters).
- **ux** (*float*) – Upper bound in the x direction (meters).
- **uy** (*float*) – Upper bound in the y direction (meters).
- **uz** (*float*) – Upper bound in the z direction (meters).

addToWaveLauncher Macro

addToWaveLauncher (*name*, *polarization*, *func*, *position*)

addToWaveLauncher (*name*, *polarization*, *func*, *lx*, *ly*, *lz*, *ux*, *uy*, *uz*)

A type of wave launcher that adds a wave function to the existing E-field on a specified boundary face. The function will be re-evaluated and added each timestep.

If the additive wave launcher is to be used on top of a regular setter wave launcher (i.e. one of type addWave-Launcher) then make sure this is called second so that it will not be overwritten.

If the wave launcher is to be placed anywhere other than on a domain boundary face, upper and lower bounds (in cell indices) may be specified instead:

Parameters

- **name** – Name of the wave launcher.
- **polariziation** (*integer*) – Direction in which the wave is polarized. This is a coordinate system dependent quantity. In Cartesian coordinates, 0 = x, 1 = y, 2 = z.
- **func** – A function describing the form of the added wave.
- **position** – Face on which to apply the wave launcher (lowerX, lowerY, lowerZ, upperX, upperY, upperZ).
- **lx** – Lower bound in the x direction (meters).
- **ly** – Lower bound in the y direction (meters).
- **lz** – Lower bound in the z direction (meters).
- **ux** – Upper bound in the x direction (meters).
- **uy** – Upper bound in the y direction (meters).
- **uz** – Upper bound in the z direction (meters).

addAngledWaveLauncher Macro

addAngledWaveLauncher (*name*, *FREQUENCY*, *WAVE_ANGLE_X*, *WAVE_ANGLE_Y*, *WAVE_ANGLE_Z*, *TRISE*, *AMPLITUDE*, *lx*, *ly*, *lz*, *ux*, *uy*, *uz*)

This macro will create a port from which a wave can be launched at a user specified angle in the X, Y and Z directions.

Parameters

- **name** – Name of the wave launcher.
- **FREQUENCY** – Frequency of the launched wave.
- **WAVE_ANGLE_X** – Angle as measured off the X-axis of the launched wave.
- **WAVE_ANGLE_Y** – Angle as measured off the Y-axis of the launched wave.
- **WAVE_ANGLE_Z** – Angle as measured off the Z-axis of the launched wave.
- **TRISE** – Rise time of the launched wave to full amplitude.
- **AMPLITUDE** – Maximum amplitude of the launched wave.
- **lx** – Lower bound in the x direction (meters).
- **ly** – Lower bound in the y direction (meters).

- **`lz`** – Lower bound in the z direction (meters).
- **`ux`** – Upper bound in the x direction (meters).
- **`uy`** – Upper bound in the y direction (meters).
- **`uz`** – Upper bound in the z direction (meters).

addCircularWaveLauncher Macro

`addCircularWaveLauncher` (*name*, *FREQUENCY*, *TRISE*, *AMPLITUDE*, *lx*, *ly*, *lz*, *ux*, *uy*, *uz*)

This macro will create a port from which a circularly polarized wave is launched.

Parameters

- **`name`** – Name of the wave launcher.
- **`FREQUENCY`** – Frequency of the launched wave.
- **`AMPLITUDE`** – Maximum amplitude of the launched wave.
- **`lx`** – Lower bound in the x direction (meters).
- **`ly`** – Lower bound in the y direction (meters).
- **`lz`** – Lower bound in the z direction (meters).
- **`ux`** – Upper bound in the x direction (meters).
- **`uy`** – Upper bound in the y direction (meters).
- **`uz`** – Upper bound in the z direction (meters).

addPort Macro

`addPort` (*position*, *vphase*, *polarization*, *waveProfile*)

This macro adds a port boundary condition to your simulation. A port combines an open boundary condition with a wave launcher. It is most commonly used for microwave device simulations. The boundary is open to waves with phase velocity sharply peaked around "vphase". It rapidly becomes infectual as the incident phase velocity deviates from *vphase*. If a multi-component incoming wave is desired, use the addIncidentWaveAtPort macro in port.mac instead: *port.mac*.

Parameters

- **`position`** – The face at which to apply the port boundary. You can specify lowerX, lowerY, lowerZ, upperX, upperY, or upperZ.
- **`vphase`** (*`float`*) – Phase velocity of the outgoing wave expected at the boundary. Velocity should be the magnitude of the component that lies along the outward facing normal at the boundary face. The boundary will be open to this phase velocity and a narrow band around it. The sign of the phase velocity should be a positive number; its sign will be taken care of automatically.
- **`polarization`** (*`integer`*) – Direction in which the wave is polarized. This is a coordinate system dependent quantity. In Cartesian coordinates 0=x, 1=y, 2=z. If no incoming wave is desired, you must still specify a number here.
- **`waveProfile`** (*`string`*) – Mathematical expression of the wave function in terms of x, y, z, t. If no incoming wave is desired, type *None*.

addConductingBoundary Macro

addConductingBoundary (*position*)

This macro creates an electrically conducting boundary condition on the face specified by the position parameter.

Parameters **position** – Face the boundary condition will be applied to (lowerX, lowerY, lowerZ, upperX, upperY or upperZ).

addMagneticBoundary Macro

addMagneticBoundary (*position*)

This macro creates a magnetically conducting boundary condition on the face specified by the position parameter.

Parameters **position** – Face the boundary condition will be applied to (lowerX,lowerY,lowerZ,upperX,upperY or upperZ).

addMALBoundary Macro

addMALBoundary (*position*, *thickness*)

This macro creates a MAL (Matched Absorbing Layer) boundary condition on the face specified in position and thickness of the difference between its upper and lower bounds.

Parameters

- **position** – Face the boundary condition will be applied to (lowerX,lowerY,lowerZ,upperX,upperY or upperZ).
- **thickness** (*float*) – The thickness of the MAL boundary (m).

addOpenBoundary Macro

addOpenBoundary (*position*)

This macro creates an open boundary condition on the face specified by the position parameter.

Parameters **position** – Face the boundary condition will be applied to (lowerX,lowerY,lowerZ,upperX,upperY or upperZ).

addPMLBoundary Macro

addPMLBoundary (*position0*, *position1*, *position2*, *position3*, *position4*, *position5*, *energyWritePeriod*, *THICKNESS*, *SIGMA_MAX*, *PML_EXP*)

This macro creates a PML region on each of the faces specified, of the thickness specified. In addition the PML is given a user defined conductivity.

Parameters

- **position0** – If lowerX is used as an argument for this parameter the PML layer will be applied to the low end of the Y-Z plane.
- **position1** – If lowerY is used as an argument for this parameter the PML layer will be applied to the low end of the X-Z plane.
- **position2** – If lowerZ is used as an argument for this parameter the PML layer will be applied to the low end of the X-Y plane.

- **position3** – If upperX is used as an argument for this parameter the PML layer will be applied to the high end of the Y-Z plane.
- **position4** – If upperY is used as an argument for this parameter the PML layer will be applied to the high end of the X-Z plane.
- **position5** – If upperZ is used as an argument for this parameter the PML layer will be applied to the high end of the X-Y plane.
- **energyWritePeriod** – Number of time steps between having the energy dissipated by the PML recorded.
- **THICKNESS** – Thickness of the PML layer (in meters).
- **SIGMA_MAX** – A constant divided by epsilon_0 or mu_0 for electric or magnetic conductivity (i.e. 3*LIGHTSPEED/DX).
- **PML_EXP** – Power-law exponent of sigma functional form (most commonly 4).

turnOn Macro

turnOn (*t*, *RISE_TIME*)

Macro to slowly turn on an excitation source. It follows a sine squared growth profile.

Parameters

- **t** – Time variable.
- **RISE_TIME** – Amount of time to turn on the excitation (s).

launchWaveFromPort Macro

launchWaveFromPort (*position*, *polarization*, *freq*, *cutoff_freq*, *amp*, *grid_loc*)

This macro specifies a port (with amp, grid_loc and position) and then launches a wave from it.

If not applied to an entire face, upper and lower bounds (in grid cells) can be defined using the arguments below

launchWaveFromPort (*posname*, *polarization*, *freq*, *amp*, *direction*, *lx*, *ly*, *lz*, *ux*, *uy*, *uz*)

Parameters

- **position** – Face the port will be applied to (lowerX,lowerY,lowerZ,upperX,upperY or upperZ).
- **posname** – Name of user defined launch position.
- **polarization** – Direction in which the wave is polarized. This is a coordinate system dependent quantity. In Cartesian coordinates 0=x, 1=y, 2=z.
- **freq** – Frequency of the launched wave.
- **cutoff_freq** – Cutoff frqenecy of the launched wave.
- **amp** – Amplitude of the launched wave.
- **grid_loc** – Location of the port on the face determined by position.
- **lx** – Lower bound in the x direction (meters).
- **ly** – Lower bound in the y direction (meters).
- **lz** – Lower bound in the z direction (meters).

- **ux** – Upper bound in the x direction (meters).
- **uy** – Upper bound in the y direction (meters).
- **uz** – Upper bound in the z direction (meters).

calculateSAR Macro

calculateSAR(*gridBoundary*, *Epsilon*, *Sigma*)
This macro will calculate the specific absorption rate of a given geometry with a user defined permittivity and conductivity.

Parameters

- **gridBoundary** – The geometry describing the object to calcualte the specific absortion rate in.
- **Epsilon** – The permittivity of the grid boundary.
- **Sigma** – The conductivity of the grid boundary.

addThinWireModel Macro

addThinWireModel(*name*, *wireAxis*, *lx*, *ly*, *lz*, *ux*, *uy*, *uz*, *sc0*, *sc1*)
This macro will create a thin wire in the volume specified. To use this macro a volume and the longitudinal axis of the wire must be specified. The wire is then scaled to a size of the volume by the scaling factors specified, allowing for the creation of an infintesimally thin wire.

Parameters

- **name** – Name of the wire.
- **wireAxis** – Longitudinal axis of the wire.
- **lx** – Lower bound in the x direction (grid cells).
- **ly** – Lower bound in the y direction (grid cells).
- **lz** – Lower bound in the z direction (grid cells).
- **ux** – Upper bound in the x direction (grid cells).
- **uy** – Upper bound in the y direction (grid cells).
- **uz** – Upper bound in the z direction (grid cells).
- **sc0** – Scaling factor applied to the size of a grid cell. If wireAxis = 0 or 2, sc0 is applied on the y-axis, if wireAxis = 1, sc0 is applied on the x-axis
- **sc1** – Upper bound in the z direction (grid cells). If wireAxis = 0 or 1 sc1 is applied on the z-axis, if wireAxis = 2, sc1 is applied on the x-axis.

3.20 Postprocessing Tools

3.20.1 FDM

FDM class: Python class, available in the file share/FDM.py file, that provides an interface to the filter diagonalization method *[WC08]* for extracting eigenmodes and their frequencies from time-domain data. In *[ACWB09]*, we validate the method and demonstrate its accuracy.

FDM methods

__init__(*self*, *prefix*, *fieldName*, *beginDump*, *endDump*)
: Initialize the FDM object. Here `prefix` is the path to the data files, up to and excluding the underscore preceding the dataset name. `fieldName` is the name of the field to analyze to obtain frequencies. `beginDump` and `endDump` are the first and one greater than the last dump numbers to use.

setPointsRandom(*self*, *npts*)
: Select `npts` random field points to use in the analysis. Components will also be chosen randomly.

setPointsUniform(*self*, *npts*, *component = None*)
: Here, `npts` is a tuple giving the number of points to use in each spatial direction; points will be placed on a uniform grid. If `component` (an integer) is given, only that component will be used. If `component` is omitted, the component will be chosen from among all field components in a round-robin fashion.

readAndCompute(*self*)
: This reads the data specified and performs the bulk of the FDM analysis. However, **computeModes** must be called to compute a requested number of modes.

computeModes(*self*, *nmodes*)
: This computes the (complex) frequencies of `nmodes` modes.

constructModes(*self*, *fieldName*, *compl = real*)
: Construct the field `fieldName` for the previously computed modes. `compl` can be either `real` or `imag`. Eigenmode fields will be written to files named `Eigenmode_*fieldName*[I]_modenum.h5` in the same directory as the data. The letter `I` is appended to the field name for imaginary fields, and `modenum` is the mode index, starting with 0, ordered from low to high frequency.

displayModes(*self*)
: Display the computed mode information in a convenient format. This gives the number of the mode, the frequency, the inverse Q and the SVD value which allows the user to understand where the reliable mode calculations stop.

Example use of FDM

```
import FDM

extr = FDM.FDM("modeExtract2s", "elecField", 2, 14)
extr.setPointsUniform((10, 10), None)
extr.readAndCompute()
extr.computeModes(2)
extr.constructModes("elecField")
extr.displayModes()
```

3.20.2 Particle Postprocessing

Particle postprocessing utility: A separate executable for performing basic reduction of Vorpal particle data in parallel.

The VSim distribution includes separate executables, **ptclPostproc** for parallel execution and **ptclPostprocser** for serial execution, that can perform basic reductions of particle data. These executables process input files consisting of attributes and attribute blocks, similar to Vorpal input files. The **Species** block specifies a data file to reduce. Possible reductions include downselection (specified in a **ParticleSelector** block) and histogramming (with a **ParticleHistogram** block).

Particle postprocessing input parameters

`runName` (*string*)
Specifies the name of the run to postprocess. This is the beginning of the data file names, up to, and excluding, the underscore before the data set name.

The input file can also contain the following blocks:

- Species
- ParticleSelector
- ParticleHistogram

When the postprocessing utility is run, the executable will read each dump of each species that appears in at least one postprocessing (`ParticleSelector` or `ParticleHistogram`) block. It will perform the requested post-processing for each of those blocks. In parallel, the particle data will be divided evenly across the processes, and the results of the processing combined once all processes have completed their analysis. The executable will write a dump file for each postprocessing block for each dump of the associated species.

Species block

The `Species` block sets parameters for a particle species to postprocess. The name of the block must be set to the name of the species data set.

`weightIndex` (*integer*)
For a weighted species, the index in the data set of the particle weight (with indices starting at 0). This can be omitted for an unweighted species.

ParticleSelector block

The ParticleSelector selects particles from the data set for which a specified variable is in a given interval, and writes the selected particles to the postprocessed output file.

`species` (*string*)
The name of the species to postprocess.

`element` (*integer*)
The data element (with indices starting at 0) on which to base the selection.

`interval` (*float vector*)
A 2-element vector giving the minimum and maximum of the interval in which to select particles.

ParticleHistogram block

The ParticleHistogram bins the specified variables of the particle data into a one- or two-dimensional histogram with the given number of bins. This procedure only works with weighted species.

`species` (*string*)
The name of the species to postprocess.

`axes` (*integer vector*)
The data element(s) (with indices starting at 0) of the particle species to bin. The dimension of the histogram will be equal to the number of elements specified in the vector.

`numBins` (*integer vector*)
The number of bins in each dimension of the histogram. The length of this vector parameter must be the same as that of the `axes` vector.

Example postprocessing input file

```
# The name of the original run.  In this example, it is a 3D run.
runName = lpa

# The number of dimensions in lpa is 2, so we set a variable for the number of␣
↪dimensions accordingly.
$NDIM = 2

# The species to read from, which is tagged and weighted
<Species plasmaElectrons>
  weightIndex = NDIM + 4
</Species>

# A selector for particles with relativistic longitudinal velocity
# above 3e7 m/s
<ParticleSelector uxSelector>
  species = plasmaElectrons
  element = NDIM
  interval = [3.0e7  1.0e20]
</ParticleSelector>

# A 1D histogram of relativistic longitudinal velocity
<ParticleHistogram momentum>
  species = plasmaElectrons
  axes = [NDIM]
  numBins = [100]
</ParticleHistogram>

# A 2D histogram of longitudinal phase space
<ParticleHistogram phaseSpace>
  species = plasmaElectrons
  axes = [0 NDIM]
  numBins = [300 300]
</ParticleHistogram>
```

Command-line options

-i
: The name of the preprocessed input file.

-d
: The location of the original data to reduce.

CHAPTER

FOUR

ENGINE (VORPAL) EXECUTION

4.1 Vorpal Command Line Options

When running Vorpal from the command line, the input filename and runtime options are specified as command line options. To use multiple options, the command line syntax is:

```
vopalser|vorpal -i filename [-o prefix_name] [-dim num] [-dt fnum] [-rnum] [-sd] [-
↪nd] [-nc] [-oc] [other options]
```

in which `vorpalser` is used to run a serial computation or **`vorpal`** for parallel. See user-guide-running-vorpal-from-the-command-line-serial-computation for details about serial computation. See the User Guide: Running in Parallel for details of command line invocation with parallel computation.

Commonly used options that you can specify on the command line include:

`-i filename`

Read input from file named *filename*.

For example:

```
vorpalser -i esPtclInCell.pre
```

`-o prefix_name`

Base names of output files on the text string *prefix_name*.

For example, if you want output files named `newesPtclInCell` rather than `esPtclInCell`, use:

```
vorpalser -i esPtclInCell.pre -o newesPtclInCell
```

`-dim num`

Run simulation using *num* spatial dimensions. This option overrides the `dimension` parameter specified in the input file.

For example, if you want to run `esPtclInCell` in 2D rather than 3D, use:

```
vorpalser -i esPtclInCell.pre -dim 2
```

`-dt fnum`

Use time step of size *fnum.* This option overrides the `dt` parameter defined in the .pre file.

For example, if you want to run `esPtclInCell` with the timestep duration 9.9e-12 seconds, use:

```
vorpalser -i esPtclInCell.pre -dt 9.9e-12
```

-n num
Run the simulation for *num* time steps. This option overrides the `nsteps` parameter.

For example, if you want to run `esPtclInCell` with 50 time steps rather than 10, use:

```
vorpalser -i esPtclInCell.pre -dt -n 50
```

-d num
Dump data every *num* time steps. This option overrides the `dumpperiodicity` parameter.

For example, to run `esPtclInCell` and dump output after every 5 time steps, use:

```
vorpalser -i esPtclInCell.pre -d 5
```

-r num`
Restart Vorpal from dump *num.*

For example, if you want to restart esPtclInCellSteps using the output dumped at time step 50, use:

```
vorpalser -i esPtclInCellSteps.pre -r 50
```

-sd
Dump data at start of simulation. This option is useful for debugging purposes. It lets you see whether Vorpal used the data you wanted it to use at the start of the simulation.

-nd
Disable data dumping.

-nc
Redirect all of the `*_comms_*.txt` output to `/dev/null`.

-oc [rank]
Suppress the creation of all comms text files except for that file from the specified [rank] process. This flag is overriden by the `-nc` flag, above.

-b
Provide a barrier at each timestep. This means that, when running in parallel at the end of a time step, the simulation will halt all processes that are more rapid than others so that the slower processes can catch up before the next timestep initiates.

-id
Forces each processor to dump into its own separate file, these files can then be concatenated into a single file for analysis.

-ns
Disable particle sorting. Sorting affects the way particles in cells and memory are handled and can lead to some inefficiencies both with sorting on or off. It is up to the user to discover whether or not to turn the sorting off. However, there are some algorithms, especially in the plasma discharge realm, in which sorting is needed.

-gpudevnum
Select the GPU to use to run a serial run.

-ra
Read all particles from all domains on a restart. Particularly useful if you try to restart with a different number of cores or configuration of domains to the dataset from which the earlier data is taken.

-v, --version
Show Vorpal version. This provides information relevant to developers if you run into issues.

-validate
Validate input file semantics.

-svn, --svn
Show svn revision number of this build.

--license
Show license info.

-iargs
Pass options (including substitutions) to internal txpp. For example:

```
vorpalser -i esPtclInCell.pre -iargs NDIM=2, DX=0.1
```

4.2 Customizing Environment Variables

Commonly used environment variables that you can specify in the shell include:

SIM_DATA_PATH
Define a list of directories for Vorpal to search data files from, in order to hold a central repository for EEDL, LXCAT and other files. When a data file is needed in a simulation, Vorpal will search it in order of current directory and all directories listed in SIM_DATA_PATH. One can set up this environment variable before running VSimComposer. The default SIM_DATA_PATH in the distribution contains potentially useful data files.

CHAPTER

FIVE

ANALYZERS

5.1 Guide to Analyzers

5.1.1 Introduction to Analyzers

Analyzers are executables provided with VSim for post processing simulation-produced data. They can produce one or a few numbers, such as mode frequencies, or they can produce large data files, like a density field from a particle file. In the latter case, the data is written into VizSchema compliant HDF5 files, and they can then be visualized in the Visualization pane.

5.1.2 Using an Analyzer

The analyzer executables are located in the same directory as the Vorpal executable. They may be used either within VSimComposer's analysis tab Python environment or invoked on the command line. For command-line usage, one must have correctly set the environment as described in command-line-execution-environment of the User Guide.

5.2 Available Analyzers

5.2.1 addPtclComponentKEeV.py

This analyzer creates a column which provides the particle energies (based on the magnitude of particle velocities) in eV. This column allows energetic particles to be differentiated within the simulation. One may also bin and average these quantities using the binning view in the visualize pane. This provides access to the plasma electron temperature, which in turn is important for assessing the mesh size against the Debye length.

If `outputSpeciesName` is not given, then the column of data is appended to the named species data set file with the names _KEeV, _KEeVx, _KEeVy, or _KEeVz depending on which component(s) are specified. If columns of these names already exist, then the script will print a message and do nothing.

Ultrarelativistic calculations of Particle Kinetic Energy in eV

If one of the *KEeVx*, *KEeVy*, or *KEeVz* component options is invoked, the analyzer performs a simplified calculation of the kinetic energy, in eV, assuming ultrarelativistic motion in the corresponding direction. Because of the assumptions available, these may be faster than the default `KEeV` option.

`-s` `<simname>`, **`--simulationName`**`=<simname>`, **`(string, required)`**
<simname> is the name of the simulation to be analyzed. The file extension should NOT be included in this text field.

-S <spname>, **--speciesName**=<spname>, **(string, required)**
<spname> is the name of the species to be subselected, to append the longitudinal or component energy in eV.

-A <add/remove>, **--addRemoveFlag**=<add/remove>, **(string, required)**
Set to either `add` or `remove`.

- If `add`, the component specified in the `component` argument will be appended to the species dataset. If the specified component already exists in an H5 file, the `add` operation will be ignored (on that file only, every file is checked individually).
- If `remove`, the last component of the species dataset will be removed, provided the last component is one of `KEeV`, `KEeVx`, `KEeVy`, or `KEeVz`. If no eV component exists in an H5 file, the `remove` operation will be ignored (on that file only, every file is checked individually).

-m, --magnitude, (bool)
Whether to output KEeV: 1 for yes, 0 for no.

-x, --outputXcomponent, (bool)
Whether to output KEeVx: 1 for yes, 0 for no.

-y, --outputYcomponent, (bool)
Whether to output KEeVy: 1 for yes, 0 for no.

-z, --outputZcomponent, (bool)
Whether to output KEeVz: 1 for yes, 0 for no.

-o <outspname>, **--outputSpeciesName**=<outspname>, **(string)**
<outspname> is the name of the species to write. Will construct filename from this species name, otherwise will use same file.

-w, --overwrite, (flag)
Whether a dataset or group should be overwritten if it already exists.

Output

This analysis script, by default, overwrites your existing species data with a file containing the additional column of energy in eV.

Alternately, if the addRemoveFlag is set to `remove`, it will remove the last component of the dataset if that component is one of `KEeV`, `KEeVx`, `KEeVy`, or `KEeVz`.

5.2.2 annotateFieldOnLine.py

This analyzer enables the raw h5 data recorded in a fieldonLine history to be available as a 2D field containing spatial cell index on the ordinate and time on the abcissa. For fieldonLine histories containing multiple components, there is an option to choose the index to test.

-s <simname>, **--simulationName**=<simname>, **(string, required)**
<simname> is the name of the simulation to be analyzed. The file extension should NOT be included in this text field.

-H <histname>, **--historyName**=<histname>, **(string, required)**
<histname> is the name of the species to be analyzed.

-C <comp>, **--component**=<comp>, **(integer, optional)**
<comp> is the component to analyze. If set, this chooses a specific index of the history.

-x, --positionMultiplier, (float, optional)
If set, multiplies the cell index for the fieldOnLine history eg by DX to determine the position.

-a, --positionOffset, (float, optional)
If set, offsets the position by this value. Adjusts the minimum value to use for cell index when plotting, eg XSTART. If used without `positionMultiplier` then this could be used for the minimum cell index so that the plot shows actual cell index, rather than containing cells numbered from zero.

-w, --overwrite, (flag)
Whether a dataset or group should be overwritten if it already exists.

Output

This analysis script outputs a VizSchema file containing 2D data on a uniform Cartesian mesh, spatial cell index (along the line specified in the history) on the ordinate, time on the abscissa.

5.2.3 annotateSpeciesAbsPtclData2.py

This analysis script enables the raw h5 data recorded in a speciesAbsPtclData2 history to be available as a field showing the number density of the macroparticles deposited in each cell, and any of the other parameters you have written out. It requires that the first components measured in the history are the particles' spatial coordinates, like `ptclAttributes = [xPosition yPosition kineticEnergy current ]` for a 2D simulation, whether a ZR or xy coordinate system, and is specific to particles collected with `collectMethod = recordForEachPtcl`.

-s `<simname>`**, --simulationName**`=<simname>`**, (string, required)**
<simname> is the name of the simulation to be analyzed. The file extension should NOT be included in this text field.

-H `<histname>`**, --historyName**`=<histname>`**, (string, required)**
<histname> is the name of the species to be analyzed.

-T, --threshold, (int, optional)
Particle densities below this will be ignored.

-w, --overwrite, (flag)
Whether a dataset or group should be overwritten if it already exists.

Output

This analysis script outputs a VizSchema field file evaluating the sum of particles deposited in each cell in that timestep, as well as a cumulative sum total.

5.2.4 computeBeam2ModeCoupling.py

This analyzer convolves a beam current with the dominant mode frequency so you can tell how much your beam is coupling to a high-power microwave device. Inputs are the name of the current density field representing the particle beam, the directional component to couple, and the frequency of the mode that is going to be coupled to. The start time is also specified in order to compute the coupling after any transient time.

-s `<simname>`**, --simulationName**`=<simname>`**, (string, required)**
<simname> is the name of the simulation to be analyzed. The file extension should NOT be included in this text field.

-j `<curdenfield>`**, --jName**`=<curdenfield>`**, (string, required)**
<curdenfield> is the name of the current density field to analyze.

-c <comp>, **--component**=<comp>, **(int, required)**
<comp> is the component 0,1,or 2 of the current density field to analyze.

-f, --frequency, (float, required)
The frequency to be analyzed.

-t, --startTime, (float, required)
The start time of the analysis.

-w, --overwrite, (flag)
Whether a dataset or group should be overwritten if it already exists.

Output

This analyzer prints the coupling coefficients to the screen and writes out a text file namd *I1.txt* with the coupling as well. If the file *I1.txt* exists, then it will only be overwritten if the -w, –overwrite flag is specified as true.

5.2.5 computeCavityG.py

It is important to measure the efficiency with which a resonating cavity stores energy. A very useful cavity figure of merit for this efficiency is the Geometry Factor, G, which depends only on the geometry of the cavity.

This analyzer computes this factor G, along with the mode frequency, and the squared magnetic field strength integrated over both the domain and the cavity surface.

-s <simname>, **--simulationName**=<simname>, **(string, required)**
<simname> is the name of the simulation to be analyzed. The file extension should NOT be included in this text field.

-g <geometry>, **--geometryName**=<geometry>, **(string)**
<geometry> is the name of the surface geometry object on which to calculate the G factor.

-b <bd>, **--beginDump**=<bd>, **(int, optional)**
<bd> is the first dump number to process.

-e <ed>, **--endDump**=<ed>, **(int, optional)**
Last dump number to process. Must be larger than `beginDump` and not greater than the total number of available dumps.

Output

A VizSchema compliant output named `Bsurf` containing information about the cavity, including the G merit factor.

5.2.6 computeDebyeLength.py

This analyzer script generates particle number density, temperature (K) and Debye length, and creates a dataset for visualization containing the fraction of Debye length occupied by each mesh cell, based on the data in particle files. Only implemented for 2D-Cylindrical coordinate systems, e.g. ZR. It is important to calculate the kinetic energy of the particles before running this analyzer. You do this by executing *addPtclComponentKEeV.py*.

-s <simname>, **--simulationName**=<simname>, **(string, required)**
<simname> is the name of the simulation to be analyzed. The file extension should NOT be included in this text field.

-S <spname>**, --speciesName**=<spname>**, (string, required)**
: <spname> is the name of the electron species for which particle number density to be calculated. If there is more than one electron species, the calculation may not be valid.

-w, --overwrite, (flag)
: Whether a dataset or group should be overwritten if it already exists.

Output

This analyzer script outputs particle number density ($/m^3$), temperature ($T = \frac{3}{2} \cdot k_B$), and the ratio of debye length measured in the units of the maximum grid dimension at that point ($D = \lambda_D/\Delta x$). All data is output as VizSchema compliant fields visualizable in Composer. A high value, greater than 10, is an indication that the Debye length is well resolved. A value less than 1 means that the Debye length is not resolved, and is an indication that particles in the simulation are likely to artificially heat. Some mitigation is possible with higher order particles but normally the best solution is to increase the grid resolution. These density fields can help to analyze the particle distribution results in the simulation domain.

5.2.7 computeEmittanceFromDump.py

This analysis script computes the RMS emittance of a beam in Cartesian coordinates and provides various other beam phase space quantities as a snapshot at a specific time.

It contains options for analyzing only a section of the longitudinal space, in case the same species has been used for two bunch operation. One should take care with 2D simulations as a greater overall charge is modeled to obtain an equivalent wake to a 3D simulation with a Gaussian distribution in z. For the 2D simulation, there is no Gaussian in z, so charge is uniform and normalized to 1m.

The analyzer will report whether it was able to find a weight label, and therefore use the relative number of particles per macroparticle to determine the emittance.

The emittance is calculated as the square root of the determinant of the covariance matrix, whose elements are the products of the averaged position and angle calculated from the transverse momentum and average velocity.

-s <simname>**, --simulationName**=<simname>**, (string, required)**
: <simname> is the name of the simulation to be analyzed. The file extension should NOT be included in this text field.

-S <spname>**, --speciesName**=<spname>**, (string, required)**
: <spname> is the name of the species to be analyzed.

-m, --movingWindow, (int, required)
: Whether to adapt the domain set by x1plane and x2plane to move with time, or the entire domain if x1plane and/or x2plane are unset.

-v, --windowSpeed, (float, optional)
: If the moving window is selected (above), the speed with which to move it. Defaults to the speed of light.

-x, --x1plane, (float, optional)
: If set, this filters the data such that particles with x less than (x1plane-ct) are excluded from the calculation.

-X, --x2plane, (float, optional)
: If set, this filters the data such that particles with x greater than (x2plane-ct) are excluded from the calculation.

-n, --numSlices, (int, optional)
: This feature allows one to choose the number of slices (between x1plane and x2plane) for the purposes of computing sliced emittance.

-w, --overwrite, (flag)
Whether a dataset or group should be overwritten if it already exists.

Output

This analysis script outputs additional histories in a vizschema (VSim) compatible output file so that variables like bunch RMS size, longitudinal and transverse emittance, and other phase space quantities can be plotted as a function of time. Additionally the analyzer produces a csv file containing the RMS values of the same various phase space measurements as a function of time.

Usage and testing

Start with the **electronBeamDrivenPlasmaT.pre** example file, and click on **View Input File**. Underneath

```
$ BEAM_GAMMA_INV_2 = BEAM_GAMMA_INV * BEAM_GAMMA_INV
```

(shortly after 200 lines into the input file) insert the text

```
$ SIGMA_Y_PRIME = 1.e-3
$ SIGMA_Y = BEAM_UX*SIGMA_Y_PRIME
```

Then in the definition of the `velocityGenerator` in the species `ElectronBeam` around line 620 change the expression for `component1`.

```
<STFunc component1>
  kind = expression
  expression = gauss(SIGMA_Y)
</STFunc>
```

You then have set both σ_y and $\sigma_{y'}$, so it is possible to simply calculate the geometric and normalized emittance.

Run this file for 5 steps, dumping every step, then click on the **Analyze** tab.

Select the **computeEmittance.py** analyzer.

Set up your options as shown using the base name of your input file (without extension), something like `ElectronBeamDrivenPlasma1` and the name of the species for analysis `ElectronBeam`.

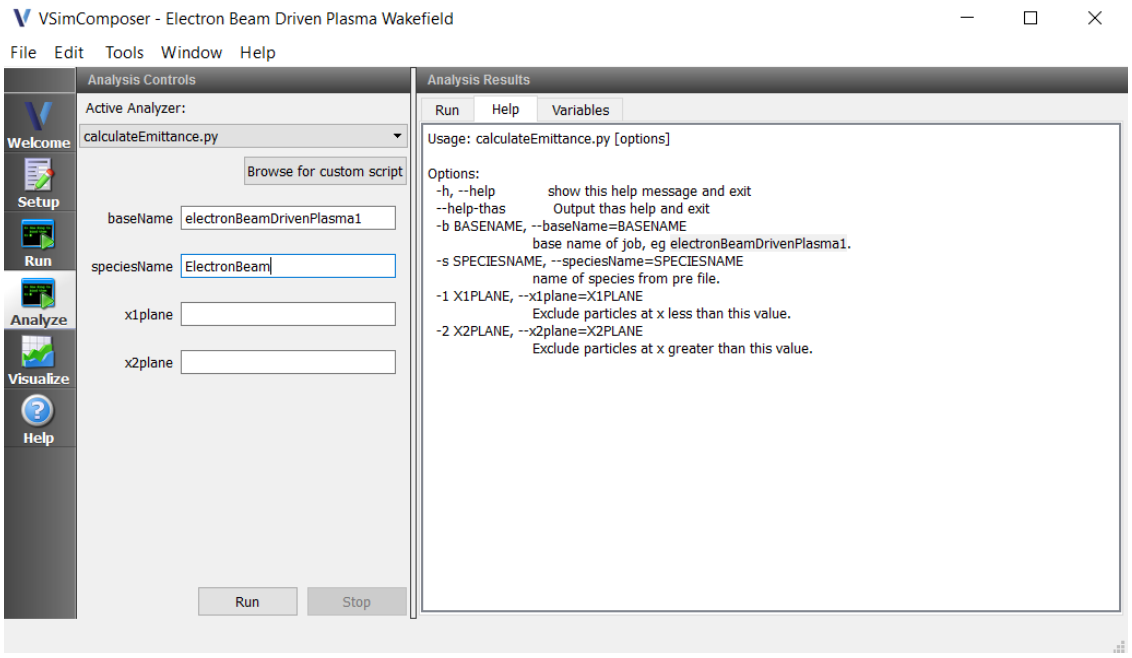
Click on **Run**. The output should look something similar to that shown below.

We can see that our `SIGMA_R` setting further up the file has translated into $\sigma_y = 2 \cdot 10^{-5}$, which is shown, $\sigma_{y'} = 1 \cdot 10^{-3}$, so geometric emittance for this uncorrelated case is $\epsilon_y = 2 \cdot 10^{-8}$. As $\gamma = 490.2$, the normalized emittance is around under $\epsilon_{yn} = 1 \cdot 10^{-5}$.

Proceed to the **Visualize** tab and select the **1-D Fields**. You may want to reload data before looking at the results. This is demonstrated in the figure. You can see the emittance in y is constant or decreases by a negligible fraction, transferred to the x component in the self-magnetic field, but due to the beam being completely monoenergetic in x to start, the x emittance grows rapidly. It is straightforward to add a gaussian function for the `BEAM_UX` to watch what happens there too.

5.2.8 computeEmittanceOnPlane.py

This analysis script, computeEmittanceOnPlane.py, computes the RMS emittance of a beam in Cartesian coordinates as it crosses a plane, and provides various other beam phase space quantities using a speciesDataOnPlane history. As with calculating the emittance from species dump files, some care should be taken with 2D simulations of novel acceleration scenarios as a greater overall charge is often modeled to obtain a field strength to an equivalent 3D



Active Analyzer: calculateEmittance.py

baseName: electronBeamDrivenPlasma1

speciesName: ElectronBeam

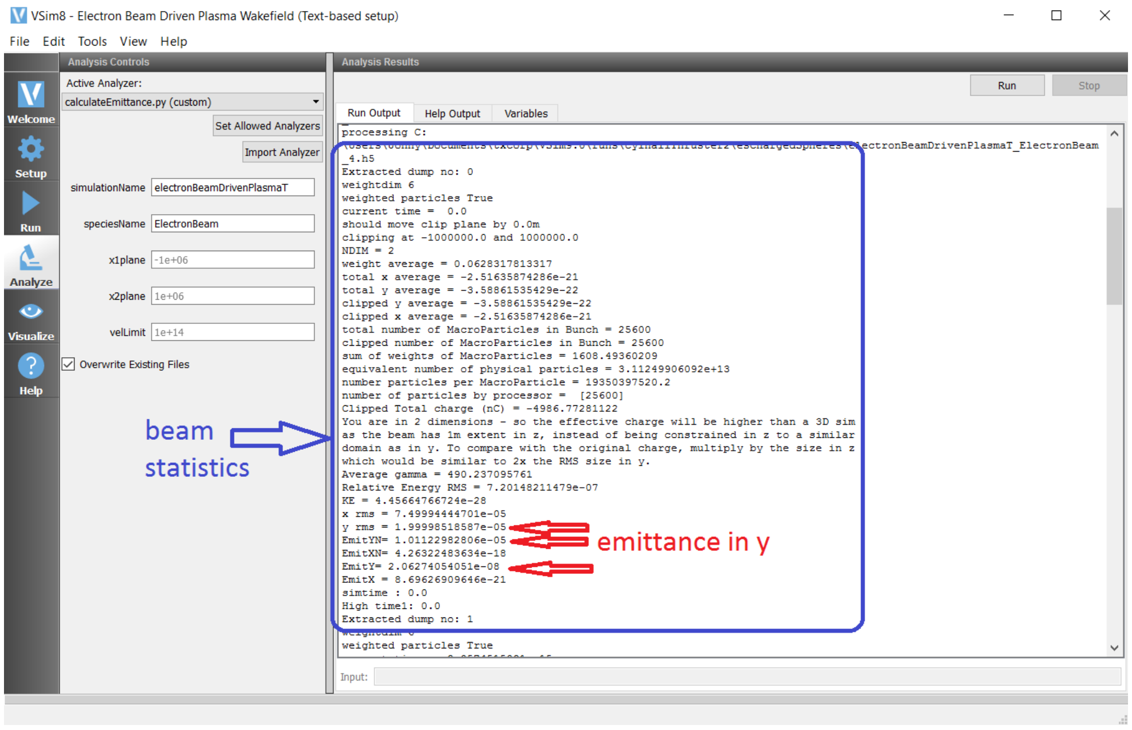
x1plane:

x2plane:

```
Usage: calculateEmittance.py [options]

Options:
  -h, --help            show this help message and exit
  --help-thas           Output thas help and exit
  -b BASENAME, --baseName=BASENAME
                        base name of job, eg electronBeamDrivenPlasma1.
  -s SPECIESNAME, --speciesName=SPECIESNAME
                        name of species from pre file.
  -1 X1PLANE, --x1plane=X1PLANE
                        Exclude particles at x less than this value.
  -2 X2PLANE, --x2plane=X2PLANE
                        Exclude particles at x greater than this value.
```

Fig. 5.1: Setting up the analyzer



Active Analyzer: calculateEmittance.py (custom)

simulationName: electronBeamDrivenPlasmaT

speciesName: ElectronBeam

x1plane: -1e+06

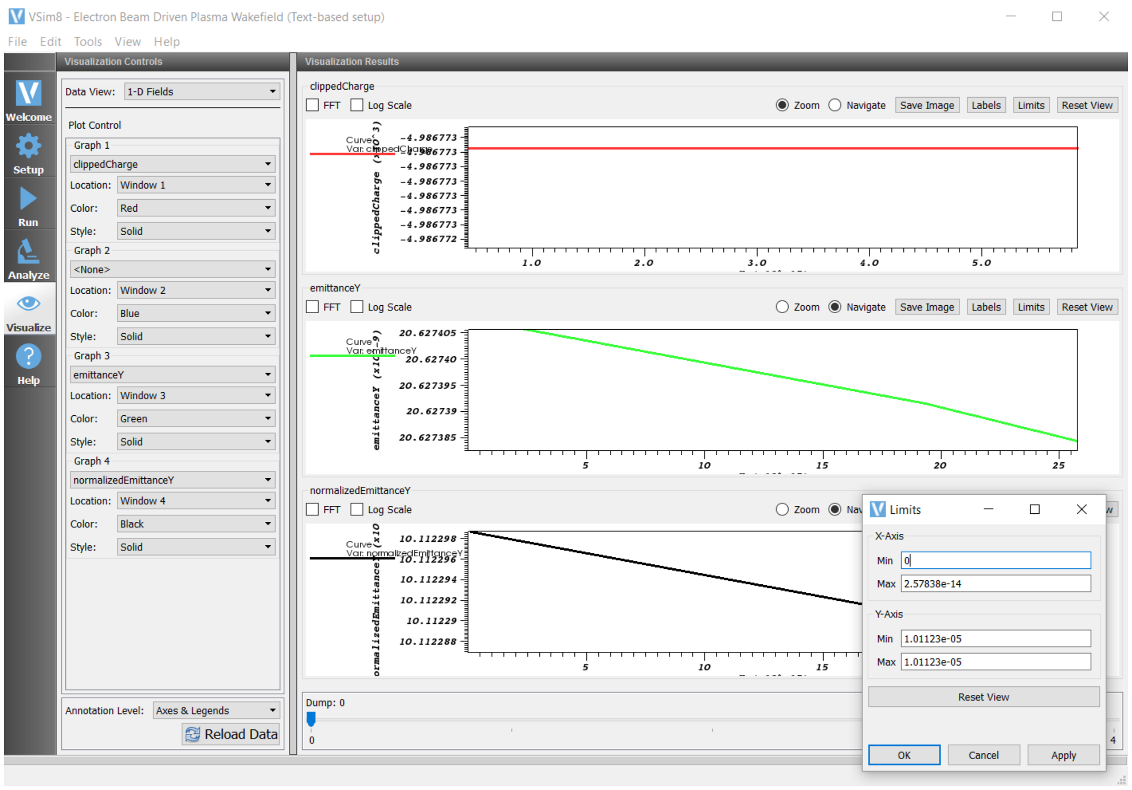
x2plane: 1e+06

velLimit: 1e+14

[✓] Overwrite Existing Files

```
processing C:
[illegible]ectronBeamDrivenPlasmaT_ElectronBeam
_4.h5
Extracted dump no: 0
weightdim 6
weighted particles True
current time =  0.0
should move clip plane by 0.0m
clipping at -1000000.0 and 1000000.0
NDIM = 2
weight average = 0.0628317813317
total x average = -2.51635874286e-21
total y average = -3.58861535429e-22
clipped y average = -3.58861535429e-22
clipped x average = -2.51635874286e-21
total number of MacroParticles in Bunch = 25600
clipped number of MacroParticles in Bunch = 25600
sum of weights of MacroParticles = 1608.49360209
equivalent number of physical particles = 3.11249906092e+13
number particles per MacroParticle = 19350397520.2
number of particles by processor =  [25600]
Clipped Total charge (nC) = -4986.77281122
You are in 2 dimensions - so the effective charge will be higher than a 3D sim
as the beam has 1m extent in z, instead of being constrained in z to a similar
domain as in y. To compare with the original charge, multiply by the size in z
which would be similar to 2x the RMS size in y.
Average gamma = 490.237095761
Relative Energy RMS = 7.20148211479e-07
KE = 4.45664766724e-28
x rms = 7.49994444701e-05
y rms = 1.99998518587e-05
EmitYN= 1.01122982806e-05
EmitXN= 4.26322483634e-18
EmitY= 2.06274054051e-08
EmitX = 8.69626909646e-21
simtime : 0.0
High time1: 0.0
Extracted dump no: 1
weightdim 6
weighted particles True
```

Fig. 5.2: Analysis Output



Data View: 1-D Fields

Graph 1: clippedCharge; Location: Window 1; Color: Red; Style: Solid

Graph 2: <None>; Location: Window 2; Color: Blue; Style: Solid

Graph 3: emittanceY; Location: Window 3; Color: Green; Style: Solid

Graph 4: normalizedEmittanceY; Location: Window 4; Color: Black; Style: Solid

Annotation Level: Axes & Legends

Dump: 0

X-Axis: Min 0, Max 2.57838e-14; Y-Axis: Min 1.01123e-05, Max 1.01123e-05

Fig. 5.3: 1-D Fields showing analysis output over time

simulation with a Gaussian distribution in z. For the 2D simulation, there is no Gaussian in z, so charge is uniform and normalized to 1m.

The speciesDataOnPlane history records metadata which is checked to see if the species is weighted. The analyzer will report if the weights were found in this metadata.

The emittances are calculated as the square root of the determinant of the covariance matrix whose elements are the products of the averaged position and primed coordinates.

Input Parameters

-s <simname>, **--simulationName**=<simname>, **(string, required)**
 <simname> is the name of the simulation to be analyzed.

-S <spname>, **--speciesName**=<spname>, **(string, required)**
 <spname> is the name of the species to be analyzed.

-H <histname>, **--historyName**=<histname>, **(string, required)**
 <histname> is the name of the speciesDataOnPlane history to be analyzed.

-w, --overwrite, (flag)
 Whether a dataset or group should be overwritten if it already exists.

Output

This analysis script outputs additional histories in a vizschema (VSim) compatible output file. These contain phase space measurements like bunch RMS size, longitudinal and transverse emittance, etc. The time slices in which they are measured correspond to the times the species data was dumped. Both time-slice measurements and cumulative measurement of reduce phase space quantities up to that time are recorded. Additionally the analyzer produces a csv file containing the RMS values of the same various phase space measurements.

5.2.9 computeFarFieldFromKirchhoffBox.py

This analyzer script analyzes a user-specified history. It will perform a near to far field transformation on the history it is given. This allows for examination of the far field parameters, or radar cross section, of a particular object. The size of the sphere used as the near field is specified in the .pre file, typically with the variable name RS.

The Kirchhoff Box should surround all structures but be a few cells inside any boundary cells, including the cells associated with any MALs or PMLs.

The start time should be set to a time that allows for the desired signal to have reached all points on the Kirchhoff Box. This is typically rise time of the pulse plus the time to cross the far-field box diagonal.

The end time of the box is typically the start time plus another box crossing time or two, plus the desired length of time one wishes to have in the far field, e.g. one or two cycles of an oscillating signal. The simulation can end after the end time of the box.

This script calculates the far fields at the specified points on the far sphere, as designated by the sphere radius and a number of points in theta and phi spherical angles. It then puts that data along with the field magnitude on a spherical mesh that can be visualized with VSimComposer in a separate file for each analysis time. It does this for multiple far field times, as controlled by the timeStepStride parameter.

Input Parameters

-s `<simname>`**, --simulationName**=`<simname>`**, (string, required)**
<simname> is the name of the simulation to be analyzed. The file extension should NOT be included in this text field.

-l `<fldlabel>`**, --fieldLabel**=`<fldlabel>`**, (string, optional)**
<fldlabel> is the name of the field for which the far field is desired. This defaults to `E` which should work.

-r, --farFieldRadius, (float)
The radius at which the far field will be calculated. This is typically set to a number that is large (10x is reasonable) compared with the size of the Kirchoff Box. However, the algorithm remains accurate for any radius outside the Kirchoff Box, even if that is technically in the radiation near field.

-n, --timeStepStride, (int)
From the first calculation of the far field, how often should it calculate the far field, as a multiplier of the simulation time step.

-g, --getFourierComponent, (int)
Whether to time integrate assuming a single Fourier frequency component.

-f, --frequency, (float)
Fourier component frequency, ignored if getFourierComponent = 0.

-t, --numTheta, (int)
The number of theta points to be calculated.

-p, --numPhi, (int)
The number of phi points to be calculated.

-z, --zeroThetaDirection, (string)
A vector pointing to where the polar angle is zero, typically the positive z direction.

-x, --zeroPhiDirection, (string)
A vector pointing to where the azimuthal angle is zero, typically the positive x direction.

-v, --varyingRadiusMesh, (int)
Whether to vary the mesh by the magnitude of the field.

-i, --simpsonIntegration, (int)
Use higher order Simpson integration.

-w, --overwrite, (flag)
Whether a dataset or group should be overwritten if it already exists.

Output

This analysis script outputs a series of HDF5 files named *<simulationName>_FarField<fieldName>_NN.vsh5*, where <simulationName> is the name of the simulation, and <fieldName> (e.g, E) is the field for which the far field is being calculated. This data can be seen in the **Visualize** tab under **Data Overview Scalar Data** as `farE` (if E is the fieldName). If the **Visualize** tab has already been opened the **Reload Data** button must be clicked in order to load the new data.

The dumps are not synchronized with the regular Vorpal data dumps, because the far field times may be strongly time delayed due to the large far field radius. Instead these dumps start at number 0, with the first far field analysis time, which is computed internally to be consistent with the required data being available in the Kirchhoff Box data stream. E.g., this time is delayed from the first dump time by half, or more, of a box diagonal transit time, depending on the centering of the Kirchhoff Box. Far field dumps are then output every `timeStepStride` as long as the necessary data for the computation is available in the Kirchhoff Box data stream.

Further analysis of these dumps is possible to get a single frequency amplitude, and is often done with a user-generated custom script. Such scripts can also be provided upon request.

5.2.10 computeFarFieldRadiation.py

This analyzer script analyzes a user-specified history. It will perform a near to far field transformation on the history it is given. This allows for examination of the far field parameters, or radar cross section of a particular object. The size of the sphere used as the near field is specified in the .pre file, typically with the variable name RS.

-s <simname>, **--simulationName**=<simname>, **(string, required)**
<simname> is the name of the simulation to be analyzed. The file extension should NOT be included in this text field.

-H <histname>, **--historyName**=<histname>, **(string, required)**
<histname> is the name of the history dataset; typically farField or RCS. The file extension should NOT be included in this text field.

-p, --numPhi, (int, required)
The number of phi points in the history. Specified in the input file.

-e, --numTheta, (int, required)
The number of theta points in the history. Specified in the input file.

-t, --numTime, (int, optional)
The number of time points spanning the far field time interval. May have been specified in the input file as NTFAR, otherwise it is 16.

-n, --SLL, (float, true)
The lowest side lobe level (dB). If set to -40 side lobes, down to -40 dB will be displayed.

-g, --normalize, (int, optional)
If set to 1, the dB gain values will be scaled so that the highest is 1. If set to 0, there is no change.

-w, --overwrite, (flag)
Whether a dataset or group should be overwritten if it already exists.

Output

This analyzer script outputs a history dataset into the HDF5 file named `simulationName_historyName.vsh5`, where historyName is the name of the farfield history dataset. In the **Visualize** tab of VSimComposer it is visualizable in **Scalar Data**. Note that if the **Visualize** tab has already been opened the **Reload Data** button must be clicked in order to load the new file.

5.2.11 computeInverseQ.py

This analyzer script computes the inverse Q from a time series. For an oscillating `history` with amplitude damping rate $\omega/2Q$, calculates the quantity $1/Q$, where $\omega = 2\pi\times$ `frequency`.

-s `<simname>`, **--simulationName**=`<simname>`, **(string, required)**
<simname> is the name of the simulation to be analyzed. The file extension should NOT be included in this text field.

-f, --frequency, (float)
Frequency of the low-pass filter.

-H `<histname>`, **--historyName**=`<histname>`, **(string)**
<histname> is the history to analyze.

-O `<outname>`, **--outputFileName**=`<outname>`, **(string, optional)**
<outname> is the name of the file into which the inverse Q will be written.

-c `<comp>`, **--component**=`<comp>`, **(int)**
<comp> is the component to select within a multi-component dataset Default = 0

-w, --overwrite, (flag)
Whether a dataset or group should be overwritten if it already exists.

Output

This analyzer script outputs two columns of data with the titles Time (s) and Inverse Q. The script also creates a VizSchema-compliant HDF5 file that contains a new dataset that can be visualized in Composer. The name of the new HDF5 file is *SIMULATIONNAME_inverseQ.vsh5* that contains the dataset inverseQ_FREQ, where FREQ is the filter frequency with periods replaced by underscores.

5.2.12 computePtclLimits.py

This analysis script helps locate particles which have breached the computational domain. This helps determine where they may have come from, to identify potential issues with sources and sinks. Additionally the analyzer writes a 'history' type particle dataset with a time mesh given by dumps which can be plotted on one axis against the extents of the species coordinates as they vary through the simulation. As such it can also find the fastest particles in a simulation, which is useful for checking that timesteps and mesh size are okay.

-s `<simname>`, **--simulationName**=`<simname>`, **(string, required)**
<simname> is the name of the simulation to be analyzed. The file extension should NOT be included in this text field.

-S `<spname>`**, --speciesName**=`<spname>`**, (string, required)**
: <spname> is the name of the species to be analyzed.

-O `<outname>`**, --outputFileName**=`<outname>`**, (string, optional)**
: <outname> is the filename to write the history output containing maximum and minimum particle coordinates for each timestep. The default is simulationname_MaxMinData_*.vsh5. If a filename is given, the user should include the simulationName and the extension .vsh5.

-w, --overwrite, (flag)
: Whether a dataset or group should be overwritten if it already exists.

Output

The output is a VizSchema compatible dataset containing a set of 'history' or 1D vs time data. The contents will be xmax, ymax, vxmax, vymin, etc. and should be equally suited to cylindrical and Cartesian datasets.

5.2.13 computePtclNumDensity.py

This analyzer script generates particle number density fields based on the particles data files. It works in 1D, 2D, and 3D Cartesian, as well as 2D RZ cylindrical.

The script can optionally perform a spatial average on the resulting data set. The spatial average is weighted and computed over an NxN square kernel. The size of the kernel and number of iterations are specified in the analyzer input. For example, if we set avgNxN to 5, the kernel would be weighted like this:

1	2	4	2	1
2	4	8	4	2
4	8	16	8	4
2	4	8	4	2
1	2	4	2	1

Setting iterateAvg to 2 will perform the averaging twice, for a smoother result.

Spatial averaging can become very slow for kernels much larger than 3, especially if the number of iterations is 2 or more, or if the dimensionality is 3.

-s `<simname>`**, --simulationName**=`<simname>`**, (string, required)**
: <simname> is the name of the simulation to be analyzed. The file extension should NOT be included in this text field.

-S `<spname>`**, --speciesName**=`<spname>`**, (string, required)**
: <spname> is the name of the species for which particle number density is to be calculated.

-N, --avgNxN, (int)
: Determines the size of the averaging kernel. Type an odd integer for N, and kernel will be N long in 1D, NxN in 2D, or NxNxN in 3D. Set to 1 if no averaging is desired.

-i, --iterateAvg, (int)
: If avgNxN > 1, this is the number of iterations of the spatial averaging function. Ignored if avgNxN = 1.

-w, --overwrite, (flag)
: Whether a dataset or group should be overwritten if it already exists.

Output

This analyzer script outputs particle number density data as fields. These density fields can help to analyze the particle distribution results in the simulation domain. The files written will have the .vsh5 file extension and will show up under **Scalar Data** on the **Visualization** pane as **speciesNameDensity**.

5.2.14 computeS11Parameters.py

This analysis script calculates the S11Parameter at one point between two sets of electromagnetic field data. For the history to work correctly, the simulation must have at least two separate sets of electromagnetic fields.

`-s` `<simname>`, **`--simulationName`**`=<simname>`, **`(string, required)`**
: <simname> is the name of the simulation to be analyzed. The file extension should NOT be included in this text field.

`-t` `<histcal>`, **`--historyCAL`**`=<histcal>`, **`(string, required)`**
: <histcal> is the name of the history used in the calibration run.

`-p` `<histpat>`, **`--historyPATCH`**`=<histpat>`, **`(string, required)`**
: <histpat> is the name of the patch history used in the test run.

`-E, --nElements, (int, optional)`
: The number of time steps in the simulation run. Defaults to 5000.

`-e, --nPoints, (int, optional)`
: The number of points to analyze within the history. Defauts to 500.

`-c, --dt, (float, required)`
: The length of one time step. Can be found in the **Run** tab under **Runtime Options**.

`-n, --nZeros, (int, optional)`
: The total length of the FFT. Can be used for zero padding. Defaults to 50000.

`-w, --overwrite, (flag)`
: Whether a dataset or group should be overwritten if it already exists.

Output

This analyzer script outputs a VizSchema file of the title *dataset_SParameters.h5*. It can be visualized as a 1D-Field. If the **Visualize** tab has already been opened the **Reload Data** button must be clicked in order to load the new file.

5.2.15 computeSParamsFromHists.py

This script fourier transforms (in time) the FieldSlabData histories, and from that computes the frequency-domain Poynting fluxes. Finally, by simple division, it provides the S parameter, the transmission coefficient.

`-s` `<simname>`, **`--simulationName`**`=<simname>`, **`(string, required)`**
: <simname> is the name of the simulation to be analyzed. The file extension should NOT be included in this text field.

`-f, --firstStep, (int, optional)`
: The number of the first time step.

`-l, --lastStep, (int, optional)`
: The number of the last time step.

-O, --stepOffset, (int, optional)
Offset between out and in time segments.

-L, --maxWavelength, (float, required)
Largest wavelength (micron).

-S, --minWavelength, (float, required)
Smallest wavelength (micron).

-d, --inDirection, (int, required)
Normal of the input plane: 0, 1, or 2.

-e <inelec>, **--inSlabE**=<inelec>, **(string, required)**
<inelec> is the input electric field array history.

-b <inmag>, **--inSlabB**=<inmag>, **(string, required)**
<inmag> is the input magnetic field array history.

-1, --inSign, (int, optional)
Sign of the input Poynting vector.

-D, --outDirection, (int, required)
Normal of the output plane: 0, 1, or 2.

-E <outelec>, **--outSlabE**=<outelec>, **(string, required)**
<outelec> is the output electric field array history.

-B <outmag>, **--outSlabB**=<outmag>, **(string, required)**
<outmag> is the output magnetic field array history.

-2, --outSign, (int, optional)
Sign of the Poynting vector to write.

-n <outname>, **--outputFileName**=<outname>, **(string, optional)**
<outname> is the name of the file into which to write the output.

-x <outsufx>, **--outputSuffix**=<outsufx>, **(string, optional)**
<outsufx> is a comma-delimited array of suffixes for the analyzer histories.

-w, --overwrite, (flag)
Whether a dataset or group should be overwritten if it already exists.

Output

A set of VsHdf5 compatible files with the frequency-domain Poynting fluxes and the S parameter.

5.2.16 computeSParamsViaOverlapIntegral.py

A python module for calculating the overlap integral for two FieldSlabData histories. The script Fourier transforms (in time) the FieldSlabData histories, and from that computes overlap integral for each frequency bin.

-s <simname>, **--simulationName**=<simname>, **(string, required)**
<simname> is the name of the simulation to be analyzed. The file extension should NOT be included in this text field.

-f, --firstStep, (int, optional)
The number of the first time step.

-l, --lastStep, (int, optional)
The name of the last time step.

-O, --stepOffset, (int, optional)
: Offset between out and in time segments.

-L, --maxWavelength, (float, required)
: Largest wavelength (micron)

-S, --minWavelength, (float, required)
: Smallest wavelength (micron)

-e <inelec>, **--inSlabE**=<inelec>, **(string, required)**
: <inelec> is the input electric field array history.

-b <inmag>, **--inSlabB**=<inmag>, **(string, required)**
: <inmag> is the input magnetic field array history.

-1, --inSign, (int, required)
: The sign of the Unit Normal used in the flux integral of the inSlab (1, or -1).

-E <outelec>, **--outSlabE**=<outelec>, **(string, required)**
: <outelec> is the output electric field array history.

-B <outmag>, **--outSlabB**=<outmag>, **(string, required)**
: <outmag> is the output magnetic field array history.

-2, --outSign, (int, required)
: The sign of the Unit Normal used in the flux integral of the outSlab (1, or -1).

-D, --direction, (int, required)
: Direction of unit normal used in flux integrals of field array histories (0, 1, or 2).

-n <outname>, **--outputFileName**=<outname>, **(string, optional)**
: <outname> is the name of the output VizSchema file to write.

-x <outsufx>, **--outputSuffix**=<outsufx>, **(string, optional)**
: <outsufx> is a comma-delimited array of suffixes for the analyzer histories.

-w, --overwrite, (flag)
: Whether a dataset or group should be overwritten if it already exists.

Output

A set of VizSchema (and so VSim Composer) compatible files with the frequency-domain Poynting fluxes and the S parameters.

5.2.17 computeSpectrogram.py

This analysis script, computeSpectrogram.py, takes a history of field at a coordinate or cell index, and works out the spectrogram, writing it into a 2D dataset containing frequency vs time which may be viewed in VSimComposer.

-s <simname>, **--simulationName**=<simname>, **(string, required)**
: <simname> is the name of the simulation to be analyzed. The file extension should NOT be included in this text field.

-H <histname>, **--historyName**=<histname>, **(string, required)**
: <histname> is the name of the history dataset to analyze, for example, E1.

-f, --fourierTransSize, (integer, required)
: Number of timesteps (or history points) to use for each Fourier transform. Determines lowest measurable frequency in the transform.

-n, --noOverlap, (integer, required)
Number of timesteps that will overlap between successive measurements of frequency. Set this to between zero and the size of the Fourier transform.

-F, --maxFreq, (float, required)
When preparing the data, discard frequencies above this maximum frequency. FFTs show a maximum frequency corresponding to 2*dt. This is higher than any frequency you are reliably going to be able to calculate.

-W, --windowType, (integer, required)
Specify a windowing function for the spectrogram Default is 1 - Hanning.

1. Hanning
2. Hamming
3. Blackmann
4. Bartlett

If you are unfamiliar with these functions, please consult numpy documentation.

-c `<comp>`, **--component**=`<comp>`, **(integer, optional)**
<comp> is the component to select if history contains multicomponent data. Default = 0

-w, --overwrite, (flag)
Whether a dataset or group should be overwritten if it already exists.

Output

This analysis script outputs a VizSchema compatible HDF5 file containing a 2D field readable in VSimComposer, time on the ordinate, frequency on the abcissa.

5.2.18 computeTimeSeriesAmplitude.py

This analyzer script computes the amplitude of a time series or more specifically calculates the instantaneous amplitude of `history` at the specified `frequency`.

-s `<simname>`, **--simulationName**=`<simname>`, **(string, required)**
<simname> is the name of the simulation to be analyzed. The file extension should NOT be included in this text field.

-f, --frequency, (float, required)
The frequency at which `history` will be analyzed.

-H `<histname>`, **--historyName**=`<histname>`, **(string, required)**
<histname> is the name of the history dataset to analyze.

-n `<outname>`, **--outputFileName**=`<outname>`, **(string, optional)**
<outname> is the name of the file into which the amplitude will be written.

-c `<comp>`, **--component**=`<comp>`, **(int, optional)**
<comp> is the component to select within a multi-component dataset Default = 0

-w, --overwrite, (flag)
Whether a dataset or group should be overwritten if it already exists.

Output

This script prints out formatted text in two columns. The first column is the time axis in seconds, and the second column is the amplitude of the time series. The scripts also creates a VizSchema-compliant Hdf5 file that contains a new dataset that can be visualized in Composer. The name of the new HDF5 file is *SIMULATIONNAME_timeSeriesAmplitude.vsh5* that contains the dataset *Amplitude_FREQ*, where *FREQ* is the filter frequency with periods replaced by underscores.

5.2.19 computeTimeSeriesFrequency.py

This analyzer script computes the frequency vs time of a time series. The script calculates the instantaneous frequency of `history` at the specified `frequency`.

-s `<simname>`, **--simulationName**=`<simname>`, **(string, required)**
    <simname> is the name of the simulation to be analyzed. The file extension should NOT be included in this text field.

-f, --frequency, (float, required)
    The frequency at which `history` will be analyzed.

-H `<histname>`, **--historyName**=`<histname>`, **(string, required)**
    <histname> is the history dataset to analyze.

-n `<outname>`, **--outputFileName**=`<outname>`, **(string, optional)**
    <outname> is the name of the file into which the amplitude will be written.

-c `<comp>`, **--component**=`<comp>`, **(int, optional)**
    <comp> is the component to select within a multi-component dataset. Default = 0.

-w, --overwrite, (flag)
    Whether a dataset or group should be overwritten if it already exists.

Output

This script prints out formatted text in two columns. The first column is the time axis in seconds, and the second column is the frequency of the time series. The script also creates a VizSchema compliant HDF5 file that contains a new dataset that can be visualized in Composer. The name of the new HDF5 file is *SIMULATIONNAME_timeSeriesFrequency.vsh5* that contains the dataset *Frequency_FREQ*, where *FREQ* is the filter frequency with periods replaced by underscores.

5.2.20 computeTransitTimeFactor.py

This analyzer computes the accelerating voltage and transit time of a cavity mode. Typically one will first run the extractModes.py analyzer to compute the the cavity modes, then run computeTransitTimeFactor.py with the modes as input.

-s `<simname>`, **--simulationName**=`<simname>`, **(string, required)**
    <simname> is the name of the simulation to be analyzed. The file extension should NOT be included in this text field.

-b `<bd>`, **--beginDump**=`<bd>`, **(int, optional)**
    <bd> is the initial dump number.

-e `<ed>`, **--endDump**=`<ed>`, **(int, optional)**
    <ed> is the dnding dump number.

-B, --beta, (float, required)
    Fraction of the speed of light for the particle bunch for which we are calculating transit time factor.

-a, --axis, (int, true)
Axis along which to compute time transit factor.

-0, --offsetx0, (float, optional)
Distance from center to offset axis along the x0 direction.

-1, --axis, (float, optional)
Distance from center to offset axis along the x1 direction.

-w, --overwrite, (flag)
Whether a dataset or group should be overwritten if it already exists.

Output

Accelerating voltage, time-independent accelerating voltage and transit time factor, computed along the specified axis in the center of the domain, are printed to the screen.

5.2.21 convertFieldComponentCartToCylX.py

This analyzer converts 3D Field data from Cartesian to cylindrical coordinates. The script analyzes 3D vector field data specified on a 3D Cartesian mesh $F_x(x, y, z), F_y(x, y, z), F_z(x, y, z)$ and provides a copy of the field's components from a Cartesian system to a cylindrical polar system around the x-axis of the Cartesian mesh $F_{z'}(x, y, z), F_{r'}(x, y, z), F_{\phi'}(x, y, z)$. We treat the system as right handed, so the angle ϕ' increases from the (Cartesian) y-axis towards the z-axis, and $z' = x$ points perpendicular from the plane swept out by ϕ'.

-s <simname>, **--simulationName**=<simname>, **(string, required)**
<simname> is the name of the simulation to be analyzed. The file extension should NOT be included in this text field.

-f, --fieldName, (string)
Name of the field to rewrite, e.g. edgeE.

-w, --overwrite, (flag)
Whether a dataset or group should be overwritten if it already exists.

Output

The output is a field, whose name is "<fieldName>xRPhi" (where the option *--fieldName* is used). The components z', r', ϕ' (where $x- > z$, $\sqrt{y^2 + z^2}- > r$ and ϕ) are calculated from the arctangent of z and y.

This analyzer script outputs an HDF5 file, titled baseName_fieldNamexRPhi.h5, i.e. magnetron2D_edgeExRPhi. The fields will be available in the **Visualize** tab of VSimComposer under **Scalar Data**, where there will be a list of three components, 0, 1, and 2 which map to z', r' and ϕ' respectively.

If the **Visualize** tab has already been opened, the **Reload Data** button must be clicked in order to load the new file.

5.2.22 convertFieldComponentCartToCylZ.py

This analyzer converts 3D Field data from Cartesian to cylindrical coordinates. The script analyzes 3D vector field data specified on a 2D or 3D Cartesian mesh $F_x(x, y, z), F_y(x, y, z), F_z(x, y, z)$, and provides a copy of the field's components from a Cartesian system to a cylindrical system around the z-axis of the Cartesian mesh $F_{z'}(x, y, z), F_{r'}(x, y, z), F_{\phi'}(x, y, z)$. We treat the system as right handed, so the angle ϕ' increases from the (Cartesian) x-axis towards the y-axis, and $z' = z$ points perpendicular from the plane swept out by ϕ'.

Additionally, this script also adds a derived 2D vector to the **input** HDF5 file that contains the X and Y components of the field, which enables quiver plots for the field data for simulations on 2D meshes. Typically this might be used for plotting magnetic field data in cartesian plots.

`-s` `<simname>`, **`--simulationName`**`=<simname>`, **`(string, required)`**
: <simname> is the name of the simulation to be analyzed. The file extension should NOT be included in this text field.

`-f, --FieldName, (string)`
: The name of the field to be analyzed with a Cartesian coordinate system. i.e. edgeE or faceB.

`-w, --overwrite, (flag)`
: Whether a dataset or group should be overwritten if it already exists.

Output

The output is a field, whose name is `<fieldName>zRPhi` (where the `FieldName` is used). The components z', r', ϕ' (where $z- > z'$, $\sqrt{y^2+y^2}- > r$ and ϕ) are calculated from the arctangent of z and y.

This analyzer script outputs an HDF5 file named *baseName_fieldNamezRPhi.h5*, i.e. *magnetron2D_edgeEzRPhi.h5*. The fields will be available in the **Visualize** tab of VSimComposer under **Scalar Data**, where there will be a list of three components, 0, 1, and 2, which map to z', r' and ϕ' respectively.

If the **Visualize** tab has already been opened, the **Reload Data** button must be clicked in order to load the new file.

5.2.23 convertPtclComponentsCartToCylX.py

Sometimes we perform simulations with natural cylindrical symmetry in Cartesian coordinates to avoid issues arising from the small timestep caused by small cells near the axis. It is helpful to interpret the results in cylindrical coordinates, rather than the Cartesian coordinates of the simulation mesh. This analyzer performs the conversion of the particle coordinates and the particle velocity. New _r, _phi, _vr, _vphi columns are added to the particle file's coordinates. If one is wishing to restart, these columns will need removing before proceeding. This assumes the x-axis in the Cartesian simulation would correspond to z', the coordinate along r = 0 in the cylindrical coordinate system. z and vz in the transformed frame is not written, as this is simply the same as x and vx in the original system, which are already present in the dataset.

`-s` `<simname>`, **`--simulationName`**`=<simname>`, **`(string, required)`**
: <simname> is the name of the simulation to be analyzed. The file extension should NOT be included in this text field.

`-S` `<spname>`, **`--speciesName`**`=<spname>`, **`(string, required)`**
: <spname> is the name of the species from which you want to read Cartesian coordinates.

`-o` `<outspname>`, **`--outputSpeciesName`**`=<outspname>`, **`(string)`**
: <outspname> is the name of the species into which you want to write cylindrical coordinates. If unset, will add these to the species from which it read the Cartesian output. This will limit your ability to restart the simulation unless you manually remove these columns from the dataset before restarting.

`-w, --overwrite, (flag)`
: Whether a dataset or group should be overwritten if it already exists.

Output

This analysis script updates the species datafile, adding _r and _phi components for both position and velocity. As $x \to z'$, there is no new component required for z.

Example usage

This analysis script may be useful for interpreting the klystron and helix TWT examples in VSim for Microwave Devices, which naturally have particles traveling down a cylindrical beampipe oriented along x.

5.2.24 convertPtclComponentsCartToCylZ.py

convertPtclComponentsCartToCylZ.py differs from convertPtclComponentsCartToCylX.py as it assumes the z axis of the original Cartesian simulation and of the axis of the cylindrical coordinate system is shared.

Sometimes we perform simulations with natural cylindrical symmetry in Cartesian coordinates to avoid issues arising from the small timestep caused by small cells near the axis. It is helpful to interpret the results in cylindrical coordinates, rather than the Cartesian coordinates of the simulation mesh. This analyzer performs the conversion of the particle coordinates and the particle velocity. New _r, _phi, _vr, _vphi columns are added to the particle file's coordinates. If one is wishing to restart, these columns will need removing before proceeding. This assumes the z-axis in the Cartesian simulation would correspond to z', the coordinate along r = 0 in the cylindrical coordinate system. z and vz in the transformed frame is not written, as this is simply the same as z and vz in the original system, which are already present in the dataset.

-s `<simname>`, **--simulationName**=`<simname>`, **(string, required)**
<simname> is the name of the simulation to be analyzed. The file extension should NOT be included in this text field.

-S `<spname>`, **--speciesName**=`<spname>`, **(string, required)**
<spname> is the name of the species from which you want to read the particle dataset in Cartesian coordinates.

-o `<outspname>`, **--outputSpeciesName**=`<outspname>`, **(string)**
<outspname> is the name of the species into which you want to write cylindrical coordinates. If unset, will add these to the species from which it read the Cartesian output. This will limit your ability to restart the simulation unless you manually remove these columns from the dataset before restarting.

-w, --overwrite, (flag)
Whether a dataset or group should be overwritten if it already exists.

Output

This analysis script updates the species datafile, adding _r and _phi components for both position and velocity. As $z \to z'$, there is no new component required for z.

Example usage

This analysis script may be useful for interpreting the 2D magnetron example in VSim for Microwave Devices, which has particles traveling in x,y space with a cylindrical central cathode oriented along z.

5.2.25 createMissingPtclsDumps.py

This analyzer writes empty particle dump files for any missing particle dumps. This is a workaround for the fact that VisIt does not know how to handle missing dumps and instead continues to display the previous particle dump of the simulation. It automatically determines all species in the .pre file and checks for field dumps without a corresponding particle dataset for each species.

-s `<simname>`, **--simulationName**=`<simname>`, **(string, required)**
<simname> is the name of the simulation to be analyzed. The file extension should NOT be included in this text field.

-S <spname>, **--speciesName**=<spname>, **(string, required)**
<spname> is the name of the species to analyze.

-w, --overwrite, (flag)
Whether a dataset or group should be overwritten if it already exists.

Output

This analyzer writes *simulationName_speciesName_NN.h5*, for each value of NN not having a particle dump but having field dumps.

5.2.26 createParticleTracks.py

This analysis script will write a new file for each particle that you wish to track over time. This requires the use of tagged particle species with a unique tag identifier for each particle.

The new .h5 file will contain the particle positions and velocities for each time dump of your simulation. Since the analyzer reads in the data from existing species dump files (e.g. *simulation_electrons_**.h5), it is beneficial to set your simulation to dump frequently.

Note: This analyzer requires tagged particle species, as the order of particles may be affected by particle creation, removal and sorting operations.

-s <simname>, **--simulationName**=<simname>, **(string, required)**
<simname> is the name of the simulation to be analyzed. The file extension should NOT be included in this text field.

-S <spname>, **--speciesName**=<spname>, **(string, required)**
<spname> is the name of the tagged species that contains the particle to be tracked.

-p, --ptclIndex, (int, required)
Tag ID of the particle to be tracked.

-w, --overwrite, (flag)
Whether a dataset or group should be overwritten if it already exists.

Output

createParticletclsTracks.py writes *simulationname_speciesnameTrackNN_0.h5*, for each value of NN where NN=ptclIndex is the tag number in the particle file to use.

5.2.27 exportSpecies.py

This analysis script interrogates particles data files, and writes them in a text file format suitable for use in the fileDensSrc kind of species source, or for those who want to have ascii format data to read into another code.

-s <simname>, **--simulationName**=<simname>, **(string, required)**
<simname> is the name of the simulation to be analyzed. The file extension should NOT be included in this text field.

-S <spname>, **--speciesName**=<spname>, **(string, required)**
<spname> is the name of the species which is the source of the export operation.

-d, --dumpNum, (int)
Which dump number to export.

-O <outname>, **--outputFileName**=<outname>, **(string, optional)**
<outname> is the name of the output file, including extension. Not used if numOutFiles > 1.

-o <outbasename>, **--outputBaseName**=<outbasename>, **(string)**
Name of the output file, no extension, compatible with the fileDensSrc applyPeriod parameter. Only used if numOutFiles > 1.

-N, --numOutFiles, (int)
Will convert the H5 file to this many .dat files, each containing totalNumPtcls/numOutFiles. Useful for loading the electrons over a period of time.

-P, --applyPeriod, (int)
If using the fileDensSrc applyPeriod parameter, set this equal to your applyPeriod.

-R, --randomizePtcls, (int)
0 or 1. If 1, will copy particles in random order.

-w, --overwrite, (flag)
Whether a dataset or group should be overwritten if it already exists.

Output

This analysis script outputs one or more ascii files readable in any test editor, potentially with a filtered set of particles ready for re-import.

5.2.28 extractModes.py

This analyzer extracts the frequencies and modes of a field from time-series data using the Filter Diagonalization Method (FDM) *[WC08]*. It is useful for simulations in which one has modes of well defined frequencies, and where only a modest number (<~20) of modes are excited. An example simulation would be that of electromagnetic oscillations rung up by a narrow frequency band of excitation current.

-s <simname>, **--simulationName**=<simname>, **(string, required)**
<simname> is the name of the simulation to be analyzed. The file extension should NOT be included in this text field.

-f <fld>, **--field**=<fld>, **(string, required)**
<fld> is the name of the field (E or B) to which we apply the FDM.

-b <bd>, **--beginDump**=<bd>, **(int, optional)**
Initial dump number (defaults to 0) used by the FDM method. This must be after the source excitation has been turned off.

-e <ed>, **--endDump**=<ed>, **(int, optional)**
Ending dump number (defaults to last dump) used by the FDM method. This is exclusive, so if `beginDump` = 10 and `endDump` = 15, FDM will use dumps 10, 11, 12, 13, and 14.

-m <nummodes>, **--nModes**=<nummodes>, **(int, required)**
Number of potential modes in the frequency range.

-t <samptype>, **--sampleType**=<samptype>, **(int, optional)**
Sampling type (0, the default, for uniform, 1 for random), which determines how the modes are sampled on the grid.

-u <unipts>, **--numberUniformPts**=<unipts>, **(int)**
Number of points to sample uniformly in each direction (required if <samptype> = 0, otherwise ignored).

-r <ranpts>, **--numberRandomPts**=<ranpts>, **(int)**
: Number of points to sample randomly (required if <samptype> = 1, otherwise ignored.)

-c <con>, **--construct**=<con>, **(int)**
: Whether to construct modes. Default is to construct.

-w, --overwrite, (flag)
: Whether a dataset or group should be overwritten if it already exists.

Output

This analyzer outputs the data in columns. The frequency is in the column labeled **freq [Hz]** (eigenmode frequency), the inverse quality factor is in the column labeled **invQ**, and the singular value decomposition index is in the column labeled **SVD**.

This analyzer also creates new eigenmode fields that may be viewed in the VSimComposer **Visualize** tab. For example, a simulation with an electric field `E` would have the scalar fields E_x/y/z (eigenE) in addition to the electric fields E_x/y/z (E).

Usage Recommendations

Current Source

It is recommended that the current source used to excite a cavity take the following form:

$$I(t) = I_0 \left\{ \frac{\sin\left[2\pi f_{hi}(t-\tau/2)\right] - \sin\left[2\pi f_{lo}(t-\tau/2)\right]}{2\pi\left(f_{hi}-f_{lo}\right)(t-\tau/2)} \right\} \exp\left[-(\delta f)^2 (t-\tau/2)^2\right] H(\tau - t)$$

as depicted in Fig. 5.4

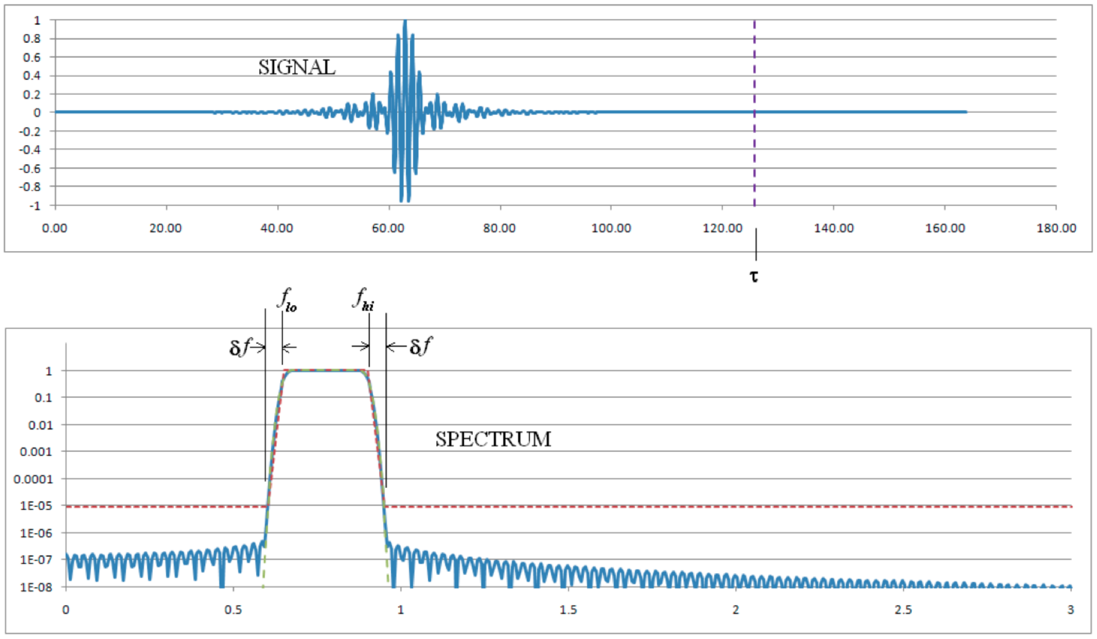


Fig. 5.4: Time and frequency space plots of current source

$I_0 = I(\tau/2)$ is the maximum current, $\tau = 2u^2/\pi\delta f$ is the duration of the excitation, f_{lo} and f_{hi} are respectively the lower and upper frequency limits of the excitation, H is the Heaviside function, and δf and u together control the steepness of the frequency spectrum near f_{lo} and f_{hi}. The ratio of the frequency space current source amplitude for frequencies outside the interval $[f_{lo} - \delta f, f_{hi} + \delta f]$ to those inside the interval can be specified by the parameter u with the following table:

Ratio	u
10^{-4}	2.76
10^{-5}	3.13
10^{-6}	3.46
10^{-7}	3.77
10^{-8}	4.06

The input option *--beginDump* should be set to a dump for which $t > \tau$.

Frequency Option

There are some rough guidelines for choosing a suitable value of the parameter δf:

- If examining the low end of a spectrum, where the mode density is not too high, $\delta f \approx f_{lo}/4$ is reasonable.
- If examining a narrow range of frequencies at the high end of a spectrum, $\delta f \approx (f_{hi} - f_{lo})/4$ is reasonable.
- If intentionally trying to exclude modes outside of, but close to, the nominal spectrum from f_{lo} to f_{hi}, a smaller value of δf may be necessary. If δf is too small, however, heavily damped modes may disappear entirely.
- If anticipating modes with loss parameter Q_m near frequency f_m, δf should satisfy $\delta f \geq 2f_m/Q_m$ to ensure the modes are present after the excitation turns off.
- For loss-less systems where $Q \to \infty$, δf may be as small as desired.
- Missing modes may occur if the duration of the simulation is too large, causing modes with finite decay to disappear (in which case, δf should be increased).

Dump Interval and Number of Dumps

The following are some guidelines for choosing the dump interval and number of dumps, *--endDump* - *--beginDump*.

- The dump interval should be smaller than half a period of the highest frequency present.
- The number of dumps should cover at least one cycle of the smallest frequency present.
- There should be at least three times as many dumps as the number of modes found.

5.2.29 extractModesViaOperator.py

This analyzer extracts the frequencies and modes of a field from time-series data using the Filter Diagonalization Method (FDM) from *[WBC13]*. This script is useful for simulations in which one has modes of well defined frequencies, and where only a modest number (<~20) of modes are excited. An example simulation would be that of electromagnetic oscillations rung up by a narrow frequency band of excitation current. The simulation must have been run with uniformly-spaced dumps or with dumps in groups of three for calculation of d / dt or d^2 / dt^2. For example, dumpSteps = [1000 1001 1002 1500 1501 1502 2000 2001 2002 …] Mode extraction appears to work best when dump groups are spaced at nearly the oscillation period of one of the modes being sought.

-s <simname>, **--simulationName**=<simname>, **(string, required)**
<simname> is the name of the simulation for which modes will be extracted. The file extension should NOT be included in this text field.

-E <E>, **--electricField**=<E>, **(string, required)**
Name of the electric field (if complex, comma-separate the parts with real part first; e.g., *E,EI*)

-B <B>, **--magneticField**=<B>, **(string, required)**
Name of the magnetic field (if complex, comma-separate the parts with real part first; e.g., *B,BI*)

-o <op>, **--operator**=<op>, **(string, required)**
One of [“d2dt2”, “ddt”]

-d <dr>, **--dumpRange**=<dr>, **(string, required)**
Dump range (in python slice format) over which to perform mode extraction. For example, **1:20**. If format is e.g. **1:**, final dump will be detected.

-S <cs>, **--cellSamples**=<cs>, **(string, required)**
Numpy slices for structured field sample selection; e.g. “0,:,:”

-c <cut>, **--cutoff**=<cut>, **(float, required)**
Smallest singular value (relative to largest singular value) to keep

-C <con>, **--construct**=<con>, **(int, optional)**
Whether to construct modes.

-w, --overwrite, (flag)
Whether a dataset or group should be overwritten if it already exists.

Output

First, the script prints the singular values of the basis formed by the midpoint dumps (e.g. dumps 1001, 1501, 2001, . . . from the example above). If the singular values of the dump basis fall to machine precision, mode extraction can be expected to work; otherwise, more dump groups may be required. Second, the script prints eigenmode frequencies and wavelengths (c divided by frequencies), along with extraction errors for each mode found. Extraction errors indicate how well extracted eigenmodes approximate a true eigenmode of the chosen operator. Frequencies are printed to this precision for convenience. Third, if the construct parameter was set to 1, the script writes eigenmode fields to vsh5 files that may be viewed in the VSimComposer **Visualize** tab. Look for scalar and vector data with names like `E` (EigenmodeReal).

Usage Recommendations

Current Source

It is recommended that the current source used to excite a cavity take the following form

$$I(t) = I_0 \left\{ \frac{\sin\left[2\pi f_{hi}(t-\tau/2)\right] - \sin\left[2\pi f_{lo}(t-\tau/2)\right]}{2\pi\left(f_{hi}-f_{lo}\right)(t-\tau/2)} \right\} \exp\left[-(\delta f)^2 (t-\tau/2)^2\right] H(\tau - t)$$

as depicted in Fig. 5.5

$I_0 = I(\tau/2)$ is the maximum current, $\tau = 2u^2/\pi\delta f$ is the duration of the excitation, f_{lo} and f_{hi} are respectively the lower and upper frequency limits of the excitation, H is the Heaviside function, and δf and u together control the steepness of the frequency spectrum near f_{lo} and f_{hi}. The ratio of the frequency space current source amplitude for frequencies outside the interval $[f_{lo} - \delta f, f_{hi} + \delta f]$ to those inside the interval can be specified by the parameter u with the following table:

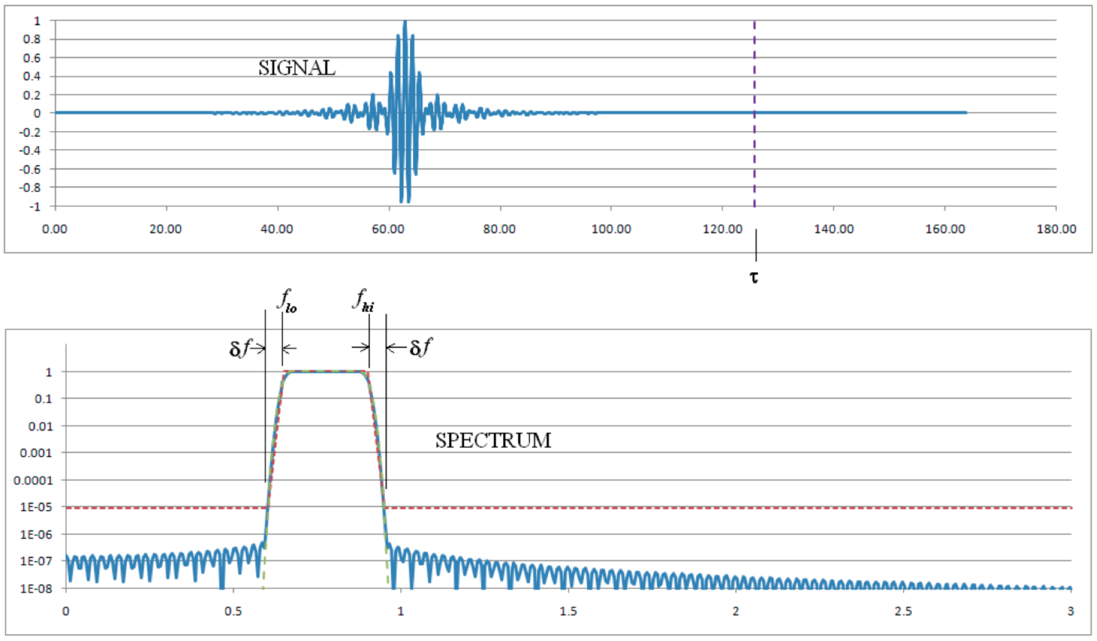


Fig. 5.5: Time and frequency space plots of current source

Ratio	u
10^{-4}	2.76
10^{-5}	3.13
10^{-6}	3.46
10^{-7}	3.77
10^{-8}	4.06

The input parameter `initialDump` should be set to a dump for which $t > \tau$.

Frequency Parameter

There are some rough guidelines for choosing a suitable value of the parameter δf:

- If examining the low end of a spectrum, where the mode density is not too high, $\delta f \approx f_{lo}/4$ is reasonable.
- If examining a narrow range of frequencies at the high end of a spectrum, $\delta f \approx (f_{hi} - f_{lo})/4$ is reasonable.
- If intentionally trying to exclude modes outside of, but close to, the nominal spectrum from f_{lo} to f_{hi}, a smaller value of δf may be necessary. If δf is too small, however, heavily damped modes may disappear entirely.
- If anticipating modes with loss parameter Q_m near frequency f_m, δf should satisfy $\delta f \geq 2f_m/Q_m$ to ensure the modes are present after the excitation turns off.
- For loss-less systems where $Q \to \infty$, δf may be as small as desired.
- Missing modes may occur if the duration of the simulation is too large, causing modes with finite decay to disappear (in which case, δf should be increased).

Dump Interval and Number

The following are some guidelines for choosing the dump interval and number of dumps, `endingDump - initialDump`.

- The dump interval should be smaller than half a period of the highest frequency present.
- The number of dumps should cover at least one cycle of the smallest frequency present.
- There should be at least three times as many dumps as the number of modes found.

5.2.30 performLowPassFilter.py

This script performs a low-pass filter on a time series that is stored in a Vorpal simulation history file. This script attenuates the amplitude of signals at a frequency higher than the specified frequency, and outputs a new time series as amplitude vs. time for the filtered signal.

`-s` `<simname>`, **`--simulationName`**`=<simname>`, **`(string, required)`**
: <simname> is the name of the simulation to be analyzed. The file extension should NOT be included in this text field.

`-f, --frequency, (float, required)`
: The frequency at which `history` will be analyzed.

`-H, --historyName, (string, required)`
: The history to analyze.

`-O` `<outname>`, **`--outputFileName`**`=<outname>`, **`(string, optional)`**
: <outname> is the name of the file into which the transformed time series will be written. Default is `SIMULATIONNAME_lowPassFilter_.vsh5`.

`-c` `<comp>`, **`--component`**`=<comp>`, **`(int, optional)`**
: <comp> is the component to select within a multi-component dataset. Default = 0.

`-w, --overwrite, (flag)`
: Whether a dataset or group should be overwritten if it already exists

Output

This script prints out formatted text in two columns representing the transformed time series after the low-pass filter has been applied. The first column is the time axis in seconds, and the second column is the amplitude of the time series. n It also creates a VizSchema compliant HDF5 file that contains the transformed time series that can be visualized in Composer. The name of the new dataset is *simulationName_lowPassFilterFREQ.vsh5*, where *FREQ* is the low-pass filter frequency with periods replaced by underscores.

5.2.31 performTwoHistoryArithmetic.py

This analyzer allows mathematical operations between two histories from the same simulation and writes the results (addition, multiplication, division, subtraction) to a third history. First we will look for histories in individual vsh5 files named baseName_history.vsh5 and if we do not find them we'll check the regular history file. This behavior reduces problems on restart.

`-s` `<simname>`, **`--simulationName`**`=<simname>`, **`(string, required)`**
: <simname> is the name of the simulation to be analyzed. The file extension should NOT be included in this text field.

-H <histname>, **--history1Name**=<histname>, **(string, required)**
<histname> is the first history to use in the mathematical operation, H1.

-I <histname>, **--history2Name<histname>, (string, required)**
<histname> is the second history to use in the mathematical operation, H2.

-O <histname>, **--newhistoryName**=<histname>, **(string, required)**
<histname> is the history name in which to record the result, H3.

-x <operation>, **--operation**=<operation>, **(string, required)**
<operation> is the mathematical operation to be applied:

a: H3=H1+H2

m: H3=H1*H2

d: H3=H1/H2

s: H3=H1-H2

-w, --overwrite, (flag)
Whether a dataset or group should be overwritten if it already exists.

Output

The output is a VizSchema compatible history file called *simulationName_newHistoryName.vsh5* containing the result. This result should be available in VSimComposer.

Example

Consider the case where you have two different particle species incident on a wall, and separate absorbers for both. You may include histories for just the individual species within the simulation input, and combine them using this analyzer to determine the net current afterwards.

5.2.32 putFieldOnSurfaceMesh.py

This analyzer reads a range of VSim output field dump files and an associated geometry file, and creates field data on an unstructured mesh representing the values of the field components on the surface of the geometry.

Appropriate boundary conditions are inferred depending on the type of field (electric or magnetic). Only those elements of the field projected on the conducting surface will be displayed. A common use of this tool is to view the surface currents on an object.

-s <simname>, **--simulationName**=<simname>, **(string, required)**
<simname> is the name of the simulation to be analyzed. The file extension should NOT be included in this text field.

-g <geometry>, **--geometryName**=<geometry>, **(string, required)**
<geometry> is the name of the conducting geometry.

-f <fldname>, **--fieldName**=<fldname>, **(string, required)**
<fldname> is the name of the field to be painted on the grid boundary.

-b <bd>, **--beginDump**=<bd>, **(int, optional)**
<bd> is the first memory dump to process. Defaults to 0.

-e <ed>, **--endDump**=<ed>, **(int, optional)**
<ed> is the last memory dump to process. Defaults to 0.

-O <outname>, **--outputFieldName**=<outname>, **(string, optional)**
<outname> is the file name of the files into which the surface field data should be written. If specified, then output files will have the form *simulationName_outputFieldName_dumpNumber.vsh5*. If not specified, then files are created as *simulationName_geometryNamefieldName_dumpNumber.vsh5*, where *geometryNamefieldName* is the concatenation of the string values passed for the **-g** and **-f** command line options.

Output

VizSchema compatible output dump files are created that contain (possibly multi-dimensional) fields and the surface mesh on which they are defined. Naming conventions are such that the new output files are named *simulationName_geometryNamefieldName_dumpNumber.vsh5*. Optionally, users can specify the output file names independently if so desired.

The fields can be found in the **Visualize** tab of VSimComposer and can be visualized in **Scalar Data**. If the **Visualize** tab has already been opened the **Reload Data** button must be clicked in order to load the new files. If the prefile generates a magnitude field, this will be available for the “painted” field.

5.2.33 subselectParticles.py

This analysis script, selects particles within a range.

-s <simname>, **--simulationName**=<simname>, **(string, required)**
<simname> is the name of the simulation with the particle files. The file extension should NOT be included in this text field.

-S <spname>, **--speciesName**=<spname>, **(string, required)**
<spname> is the name of the species to be subselected.

-o <outspname>, **--outputSpeciesName**=<outspname>, **(string, optional)**
<outspname> is the name of the species that will be the result of the subselection.

-c <comp>, **--component**=<comp>, **(int, optional)**
<comp> is the component index integer to filter. Depends on species kind and dimensionality.

-l <lb>, **--lowerBounds**=<lb>, **(float, optional)**
Minimum value of the component for keeping a particle.

-u <ub>, **--upperBounds**=<ub>, **(float, optional)**
Maximum value of the component for keeping a particle.

-f <fl>, **--flipLimits**=<fl>, **(int, required)**
Where upper and lower bounds are both provided, this allows subselecting particles outside the range, rather than inside.

-w, --overwrite, (flag)
Whether a dataset or group should be overwritten if it already exists.

Output

This analysis script outputs a set of VizSchema compatible HDF5 files readable in VSimComposer, VisIt and other tools containing a filtered set of particles.

CHAPTER SIX

TRADEMARKS AND LICENSING

INDEX

Symbols

-0, –offsetx0, (float, optional)
 computeTimeSeriesFrequency.py command line option, 616
-1, –axis, (float, optional)
 computeTimeSeriesFrequency.py command line option, 616
-1, –inSign, (int, required)
 computeSParamsViaOverlapIntegral.py command line option, 613
-2, –outSign, (int, optional)
 computeSParamsFromHists.py command line option, 612
-2, –outSign, (int, required)
 computeSParamsViaOverlapIntegral.py command line option, 613
-A <add/remove>, –addRemoveFlag=<add/remove>, (string, required)
 addPtclComponentKEeV.py command line option, 600
-B <B>, –magneticField=<B>, (string, required)
 extractModesViaOperator.py command line option, 623
-B <outmag>, –outSlabB=<outmag>, (string, required)
 computeSParamsFromHists.py command line option, 612
 computeSParamsViaOverlapIntegral.py command line option, 613
-B, –beta, (float, required)
 computeTimeSeriesFrequency.py command line option, 615
-C <comp>, –component=<comp>, (integer, optional)
 annotateFieldOnLine.py command line option, 600
-C <con>, –construct=<con>, (int, optional)
 extractModesViaOperator.py command line option, 623
-D, –direction, (int, required)
 computeSParamsViaOverlapIntegral.py command line option, 613
-D, –outDirection, (int, required)
 computeSParamsFromHists.py command line option, 612
-E <E>, –electricField=<E>, (string, required)
 extractModesViaOperator.py command line option, 623
-E <outelec>, –outSlabE=<outelec>, (string, required)
 computeSParamsFromHists.py command line option, 612
 computeSParamsViaOverlapIntegral.py command line option, 613
-E, –nElements, (int, optional)
 computeS11Parameters.py command line option, 611
-F, –maxFreq, (float, required)
 computeSpectrogram.py command line option, 614
-H <histname>, –history1Name=<histname>, (string, required)
 command line option, 625
-H <histname>, –historyName=<histname>, (string)
 computeInverseQ.py command line option, 609
-H <histname>, –historyName=<histname>, (string, required)
 annotateFieldOnLine.py command line option, 600
 annotateSpeciesAbsPtclData2.py command line option, 601
 computeEmittanceOnPlane.py command line option, 606
 computeFarFieldRadiation.py command line option, 608
 computeSpectrogram.py command line option, 613
 computeTimeSeriesAmplitude.py command line option, 614
 computeTimeSeriesFrequency.py command line option, 615
-H, –historyName, (string, required)
 performLowPassFilter.py command line option, 625
-I <histname>, –history2Name<histname>, (string, required)
 command line option, 626
-L, –maxWavelength, (float, required)
 computeSParamsFromHists.py command line option, 612
 computeSParamsViaOverlapIntegral.py command line option, 613

-N, –avgNxN, (int)
computePtclNumDensity.py command line option, 610
-N, –numOutFiles, (int)
exportSpecies.py command line option, 620
-O <histname>, –newhistoryName=<histname>, (string, required)
command line option, 626
-O <outname>, –outputFieldName=<outname>, (string, optional)
putFieldOnSurfaceMesh.py command line option, 626
-O <outname>, –outputFileName=<outname>, (string, optional)
computeInverseQ.py command line option, 609
computePtclLimits.py command line option, 610
exportSpecies.py command line option, 620
performLowPassFilter.py command line option, 625
-O, –stepOffset, (int, optional)
computeSParamsFromHists.py command line option, 611
computeSParamsViaOverlapIntegral.py command line option, 612
-P, –applyPeriod, (int)
exportSpecies.py command line option, 620
-R, –randomizePtcls, (int)
exportSpecies.py command line option, 620
-S <cs>, –cellSamples=<cs>, (string, required)
extractModesViaOperator.py command line option, 623
-S <spname>, –speciesName=<spname>, (string, required)
addPtclComponentKEeV.py command line option, 599
computeDebyeLength.py command line option, 602
computeEmittanceFromDump.py command line option, 603
computeEmittanceOnPlane.py command line option, 606
computePtclLimits.py command line option, 609
computePtclNumDensity.py command line option, 610
convertPtclComponentsCartToCylX.py command line option, 617
convertPtclComponentsCartToCylZ.py command line option, 618
createMissingPtclsDumps.py command line option, 618
createParticleTracks.py command line option, 619
exportSpecies.py command line option, 619
subselectparticles.py command line option, 627
-S, –minWavelength, (float, required)
computeSParamsFromHists.py command line option, 612
computeSParamsViaOverlapIntegral.py command line option, 613
-T, –threshold, (int, optional)
annotateSpeciesAbsPtclData2.py command line option, 601
-W, –windowType, (integer, required)
computeSpectrogram.py command line option, 614
-X, –x2plane, (float, optional)
computeEmittanceFromDump.py command line option, 603
-a, –axis, (int, true)
computeTimeSeriesFrequency.py command line option, 615
-a, –positionOffset, (float, optional)
annotateFieldOnLine.py command line option, 600
-b <bd>, –beginDump=<bd>, (int, optional)
computeCavityG.py command line option, 602
computeTimeSeriesFrequency.py command line option, 615
extractModes.py command line option, 620
putFieldOnSurfaceMesh.py command line option, 626
-b <inmag>, –inSlabB=<inmag>, (string, required)
computeSParamsFromHists.py command line option, 612
computeSParamsViaOverlapIntegral.py command line option, 613
-c <comp>, –component=<comp>, (int)
computeInverseQ.py command line option, 609
-c <comp>, –component=<comp>, (int, optional)
computeTimeSeriesAmplitude.py command line option, 614
computeTimeSeriesFrequency.py command line option, 615
performLowPassFilter.py command line option, 625
subselectparticles.py command line option, 627
-c <comp>, –component=<comp>, (int, required)
computeBeam2ModeCoupling.py command line option, 601
-c <comp>, –component=<comp>, (integer, optional)
computeSpectrogram.py command line option, 614
-c <con>, –construct=<con>, (int)
extractModes.py command line option, 621
-c <cut>, –cutoff=<cut>, (float, required)
extractModesViaOperator.py command line option, 623
-c, –dt, (float, required)
computeS11Parameters.py command line option, 611
-d <dr>, –dumpRange=<dr>, (string, required)
extractModesViaOperator.py command line option, 623
-d, –dumpNum, (int)
exportSpecies.py command line option, 619

-d, –inDirection, (int, required)
computeSParamsFromHists.py command line option, 612
-e <ed>, –endDump=<ed>, (int, optional)
computeCavityG.py command line option, 602
computeTimeSeriesFrequency.py command line option, 615
extractModes.py command line option, 620
putFieldOnSurfaceMesh.py command line option, 626
-e <inelec>, –inSlabE=<inelec>, (string, required)
computeSParamsFromHists.py command line option, 612
computeSParamsViaOverlapIntegral.py command line option, 613
-e, –nPoints, (int, optional)
computeS11Parameters.py command line option, 611
-e, –numTheta, (int, required)
computeFarFieldRadiation.py command line option, 608
-f <fl>, –flipLimits=<fl>, (int, required)
subselectparticles.py command line option, 627
-f <fld>, –field=<fld>, (string, required)
extractModes.py command line option, 620
-f <fldname>, –fieldName=<fldname>, (string, required)
putFieldOnSurfaceMesh.py command line option, 626
-f, –FieldName, (string)
convertFieldComponentCartToCylZ.py command line option, 617
-f, –fieldName, (string)
convertFieldComponentCartToCylX.py command line option, 616
-f, –firstStep, (int, optional)
computeSParamsFromHists.py command line option, 611
computeSParamsViaOverlapIntegral.py command line option, 612
-f, –fourierTransSize, (integer, required)
computeSpectrogram.py command line option, 613
-f, –frequency, (float)
computeFarFieldFromKirchhoffBox.py command line option, 607
computeInverseQ.py command line option, 609
-f, –frequency, (float, required)
computeBeam2ModeCoupling.py command line option, 602
computeTimeSeriesAmplitude.py command line option, 614
computeTimeSeriesFrequency.py command line option, 615
performLowPassFilter.py command line option, 625
-g <geometry>, –geometryName=<geometry>, (string)
computeCavityG.py command line option, 602
-g <geometry>, –geometryName=<geometry>, (string, required)
putFieldOnSurfaceMesh.py command line option, 626
-g, –getFourierComponent, (int)
computeFarFieldFromKirchhoffBox.py command line option, 607
-g, –normalize, (int, optional)
computeFarFieldRadiation.py command line option, 608
-i, –iterateAvg, (int)
computePtclNumDensity.py command line option, 610
-i, –simpsonIntegration, (int)
computeFarFieldFromKirchhoffBox.py command line option, 608
-j <curdenfield>, –jName=<curdenfield>, (string, required)
computeBeam2ModeCoupling.py command line option, 601
-l <fldlabel>, –fieldLabel=<fldlabel>, (string, optional)
computeFarFieldFromKirchhoffBox.py command line option, 607
-l <lb>, –lowerBounds=<lb>, (float, optional)
subselectparticles.py command line option, 627
-l, –inSign, (int, optional)
computeSParamsFromHists.py command line option, 612
-l, –lastStep, (int, optional)
computeSParamsFromHists.py command line option, 611
computeSParamsViaOverlapIntegral.py command line option, 612
-m <nummodes>, –nModes=<nummodes>, (int, required)
extractModes.py command line option, 620
-m, –magnitude, (bool)
addPtclComponentKEeV.py command line option, 600
-m, –movingWindow, (int, required)
computeEmittanceFromDump.py command line option, 603
-n <outname>, –outputFileName=<outname>, (string, optional)
computeSParamsFromHists.py command line option, 612
computeSParamsViaOverlapIntegral.py command line option, 613
computeTimeSeriesAmplitude.py command line option, 614
computeTimeSeriesFrequency.py command line option, 615
-n, –SLL, (float, true)

computeFarFieldRadiation.py command line option, 608
-n, –nZeros, (int, optional)
computeS11Parameters.py command line option, 611
-n, –noOverlap, (integer, required)
computeSpectrogram.py command line option, 613
-n, –numSlices, (int, optional)
computeEmittanceFromDump.py command line option, 603
-n, –timeStepStride, (int)
computeFarFieldFromKirchhoffBox.py command line option, 607
-o <op>, –operator=<op>, (string, required)
extractModesViaOperator.py command line option, 623
-o <outbasename>, –outputBaseName=<outbasename>, (string)
exportSpecies.py command line option, 620
-o <outspname>, –outputSpeciesName=<outspname>, (string)
addPtclComponentKEeV.py command line option, 600
convertPtclComponentsCartToCylX.py command line option, 617
convertPtclComponentsCartToCylZ.py command line option, 618
-o <outspname>, –outputSpeciesName=<outspname>, (string, optional)
subselectparticles.py command line option, 627
-p <histpat>, –historyPATCH=<histpat>, (string, required)
computeS11Parameters.py command line option, 611
-p, –numPhi, (int)
computeFarFieldFromKirchhoffBox.py command line option, 607
-p, –numPhi, (int, required)
computeFarFieldRadiation.py command line option, 608
-p, –ptclIndex, (int, required)
createParticleTracks.py command line option, 619
-r <ranpts>, –numberRandomPts=<ranpts>, (int)
extractModes.py command line option, 620
-r, –farFieldRadius, (float)
computeFarFieldFromKirchhoffBox.py command line option, 607
-s <simname>, –simulationName=<simname>, (string, required)
addPtclComponentKEeV.py command line option, 599
annotateFieldOnLine.py command line option, 600
annotateSpeciesAbsPtclData2.py command line option, 601
command line option, 625
computeBeam2ModeCoupling.py command line option, 601
computeCavityG.py command line option, 602
computeDebyeLength.py command line option, 602
computeEmittanceFromDump.py command line option, 603
computeEmittanceOnPlane.py command line option, 606
computeFarFieldFromKirchhoffBox.py command line option, 607
computeFarFieldRadiation.py command line option, 608
computeInverseQ.py command line option, 609
computePtclLimits.py command line option, 609
computePtclNumDensity.py command line option, 610
computeS11Parameters.py command line option, 611
computeSParamsFromHists.py command line option, 611
computeSParamsViaOverlapIntegral.py command line option, 612
computeSpectrogram.py command line option, 613
computeTimeSeriesAmplitude.py command line option, 614
computeTimeSeriesFrequency.py command line option, 615
convertFieldComponentCartToCylX.py command line option, 616
convertFieldComponentCartToCylZ.py command line option, 617
convertPtclComponentsCartToCylX.py command line option, 617
convertPtclComponentsCartToCylZ.py command line option, 618
createMissingPtclsDumps.py command line option, 618
createParticleTracks.py command line option, 619
exportSpecies.py command line option, 619
extractModes.py command line option, 620
extractModesViaOperator.py command line option, 622
performLowPassFilter.py command line option, 625
putFieldOnSurfaceMesh.py command line option, 626
subselectparticles.py command line option, 627
-t <histcal>, –historyCAL=<histcal>, (string, required)
computeS11Parameters.py command line option, 611
-t <samptype>, –sampleType=<samptype>, (int, optional)
extractModes.py command line option, 620
-t, –numTheta, (int)

computeFarFieldFromKirchhoffBox.py command line option, 607
-t, –numTime, (int, optional)
computeFarFieldRadiation.py command line option, 608
-t, –startTime, (float, required)
computeBeam2ModeCoupling.py command line option, 602
-u <ub>, –upperBounds=<ub>, (float, optional)
subselectparticles.py command line option, 627
-u <unipts>, –numberUniformPts=<unipts>, (int)
extractModes.py command line option, 620
-v, –varyingRadiusMesh, (int)
computeFarFieldFromKirchhoffBox.py command line option, 608
-v, –windowSpeed, (float, optional)
computeEmittanceFromDump.py command line option, 603
-w, –overwrite, (flag)
addPtclComponentKEeV.py command line option, 600
annotateFieldOnLine.py command line option, 601
annotateSpeciesAbsPtclData2.py command line option, 601
command line option, 626
computeBeam2ModeCoupling.py command line option, 602
computeDebyeLength.py command line option, 603
computeEmittanceFromDump.py command line option, 603
computeEmittanceOnPlane.py command line option, 606
computeFarFieldFromKirchhoffBox.py command line option, 608
computeFarFieldRadiation.py command line option, 609
computeInverseQ.py command line option, 609
computePtclLimits.py command line option, 610
computePtclNumDensity.py command line option, 610
computeS11Parameters.py command line option, 611
computeSParamsFromHists.py command line option, 612
computeSParamsViaOverlapIntegral.py command line option, 613
computeSpectrogram.py command line option, 614
computeTimeSeriesAmplitude.py command line option, 614
computeTimeSeriesFrequency.py command line option, 615, 616
convertFieldComponentCartToCylX.py command line option, 616
convertFieldComponentCartToCylZ.py command line option, 617
convertPtclComponentsCartToCylX.py command line option, 617
convertPtclComponentsCartToCylZ.py command line option, 618
createMissingPtclsDumps.py command line option, 619
createParticleTracks.py command line option, 619
exportSpecies.py command line option, 620
extractModes.py command line option, 621
extractModesViaOperator.py command line option, 623
performLowPassFilter.py command line option, 625
subselectparticles.py command line option, 627
-x <operation> , –operation=<operation>, (string, required)
command line option, 626
-x <outsufx>, –outputSuffix=<outsufx>, (string, optional)
computeSParamsFromHists.py command line option, 612
computeSParamsViaOverlapIntegral.py command line option, 613
-x, –outputXcomponent, (bool)
addPtclComponentKEeV.py command line option, 600
-x, –positionMultiplier, (float, optional)
annotateFieldOnLine.py command line option, 600
-x, –x1plane, (float, optional)
computeEmittanceFromDump.py command line option, 603
-x, –zeroPhiDirection, (string)
computeFarFieldFromKirchhoffBox.py command line option, 607
-y, –outputYcomponent, (bool)
addPtclComponentKEeV.py command line option, 600
-z, –outputZcomponent, (bool)
addPtclComponentKEeV.py command line option, 600
-z, –zeroThetaDirection, (string)
computeFarFieldFromKirchhoffBox.py command line option, 607
__init__, 591

A

A (in module RxnRate-expPolyCrossSection), 278
A_FN (in module fieldEmitterVelGen), 263
absAndEmitBox() (built-in function), 498
absAndSav (module), 207
absCutCell (module), 207
absorberIsInGridBoundary (in module absSavTriCutCell), 210
absorbingBox() (built-in function), 497

absorbingBoxCylAxis() (built-in function), 498
absorbLocation (in module absSavTriCutCell), 210
absSavBoxCylAxis() (built-in function), 498
absSavCutCell (module), 208
absSaveDump (module), 210
absSavStairStep (module), 212
absSavTriCutCell (module), 209
absStairStep (module), 211
absTransparentPtclFlag (in module transparentBndry), 215
acc (in module diffuseBndry), 213
accelerate (in module Fluid), 190
acceleratingVoltage (module), 360
aCoeff (in module fieldBinOpUpdater), 119
aCoeff (in module scalarBinOpUpdater), 173
aCoeff (in module scalarFieldBinOpUpdater), 175
Acol (in module binaryIonization), 315
add2FieldSqrVolIntegralHist() (built-in function), 538
addAngledWaveLauncher() (built-in function), 586
addBaseSolver() (built-in function), 487
addBeamEmitter() (built-in function), 463, 464
addBinaryCombinationHistory() (built-in function), 476
addBoundaryLauncher() (built-in function), 469
addChildLangmuirEmitter() (built-in function), 483
addChirpWavePulseSTFuncBlock() (built-in function), 488
addCircularWaveLauncher() (built-in function), 587
addComplexPoyntingSurfaceIntegralHist() (built-in function), 551
addConductingBoundary() (built-in function), 588
addConductor() (built-in function), 487
addConstantProbabilitySecondaryElectronEmitter() (built-in function), 485
addCosineFlattopSTFuncBlock() (built-in function), 488
addCosineRampSTFuncBlock() (built-in function), 488
addCurrentDistribution() (built-in function), 490
addCurrentSource() (built-in function), 584
addDielectric() (built-in function), 487, 508
addDipoleCurrentDistribution() (built-in function), 490
addElectricBoundary() (built-in function), 469
addElectricFieldEnergyHistory() (built-in function), 475
addElectronNeutralFluidCollisions() (built-in function), 465
addElectronNeutralFluidElasticCollision() (built-in function), 466
addElectronNeutralFluidExcitationCollision() (built-in function), 466
addElectronNeutralFluidIonizationCollision() (built-in function), 467
addEMFieldEnergyHistory() (built-in function), 474, 475
addEnergyInBHist() (built-in function), 547
addEnergyInBHistComplex() (built-in function), 547
addEnergyInDielectricFromEHist() (built-in function), 549
addEnergyInEHist() (built-in function), 548
addEnergyInEHistComplex() (built-in function), 548
addEsDirichletBC() (built-in function), 470, 471
addEsNeumannBC() (built-in function), 471
addFarFieldBoxHistory() (built-in function), 476
addFarFieldHistory() (built-in function), 476, 501
addFarFieldHistoryPowerUser() (built-in function), 502
addFarFields() (built-in function), 516
addFeedbackSTFuncBlock() (built-in function), 488
addFieldAtCoordHist() (built-in function), 534
addFieldEnergyAtPositionHistory() (built-in function), 475
addFieldInitialCondition() (built-in function), 479
addFieldSqrSurfIntegralHist() (built-in function), 541
addFieldSqrSurfMaxHist() (built-in function), 541
addFieldSqrSurfMinHist() (built-in function), 543
addFieldSqrSurfOpHist() (built-in function), 539
addFieldSqrVolIntegralHist() (built-in function), 536
addFieldSqrVolMaxHist() (built-in function), 537
addFieldSqrVolOpHist() (built-in function), 535
addFieldSurfComplex3DCrossIntegralHist() (built-in function), 544
addFowlerNordheimEmitter() (built-in function), 483
addGaussianPulseSTFuncBlock() (built-in function), 488
addGaussianSTFuncBlock() (built-in function), 488
addHalfSinePulseSTFuncBlock() (built-in function), 488
addIncidentWaveAtBoundaryPort() (built-in function), 568
addIncidentWaveAtPort() (built-in function), 567
addInitialCondition() (built-in function), 479
addInstantWallDissipationHist() (built-in function), 550
addInteriorSecondaryElectronEmitter() (built-in function), 484
addIonNeutralFluidChargeExchangeCollision() (built-in function), 467, 468
addIonNeutralFluidCollisions() (built-in function), 465
addIonNeutralFluidMomentumExchangeCollision() (built-in function), 468
addKirchhoffSurfaceIntegral() (built-in function), 477
addLeakyChannelSTFuncBlock() (built-in function), 488
addLorentzByCarrier_DDLD() (built-in function), 495
addLorentzByConductanceRate_DDLD() (built-in function), 496
addLossyDielectric() (built-in function), 508
addMacroParticleCountHistory() (built-in function), 474
addMagneticBoundary() (built-in function), 469, 588
addMagneticFieldEnergyHistory() (built-in function), 475
addMALBoundary() (built-in function), 588
addMalBoundary() (built-in function), 469
addNeutralGas() (built-in function), 472
addOpenBoundary() (built-in function), 469, 588
addParticleAbsorberCurrentHistory() (built-in function), 474

addParticleAbsorberEnergyHistory() (built-in function), 474
addParticleAbsorberLog() (built-in function), 473
addParticleEmitterCurrentHistory() (built-in function), 474
addParticleEnergyHistory() (built-in function), 474
addParticleFields() (built-in function), 584
addParticleMomentumHistory() (built-in function), 473
addParticleSpecies() (built-in function), 480
addParticleSpeciesLoader() (built-in function), 480, 481
addParticleSpeciesSink() (built-in function), 481
addPhysicalParticleCountHistory() (built-in function), 474
addPhysicsSetFluxEmitterSource() (built-in function), 482
addPlaneWavePulseSTFuncBlock() (built-in function), 488
addPMLBoundary() (built-in function), 588
addPort() (built-in function), 587
addPortBoundary() (built-in function), 566
addPoyntingHist() (built-in function), 534
addPoyntingHistory() (built-in function), 476
addProductFunctionSTFuncBlock() (built-in function), 489
addPseudoPotentialHistory() (built-in function), 476
addPtclComponentKEeV.py command line option
  -A <add/remove>, –addRemoveFlag=<add/remove>, (string, required), 600
  -S <spname>, –speciesName=<spname>, (string, required), 599
  -m, –magnitude, (bool), 600
  -o <outspname>, –outputSpeciesName=<outspname>, (string), 600
  -s <simname>, –simulationName=<simname>, (string, required), 599
  -w, –overwrite, (flag), 600
  -x, –outputXcomponent, (bool), 600
  -y, –outputYcomponent, (bool), 600
  -z, –outputZcomponent, (bool), 600
addRadialCosineChannelSTFuncBlock() (built-in function), 488
addRadPortBoundary() (built-in function), 560
addRichardsonDushmanEmitter() (built-in function), 483
addSecondaryElectronEmitter() (built-in function), 483
addShape() (built-in function), 486
addSigmaForm (in module PmlRegion), 182
addSimpleSecondaryElectronEmitter() (built-in function), 484
addSinePlaneWaveSTFuncBlock() (built-in function), 489
addSputterEmitter() (built-in function), 485, 486
addSumFunctionSTFuncBlock() (built-in function), 489
addThinWireModel() (built-in function), 590
addTimeDependence() (built-in function), 491
addToCurrentSource() (built-in function), 585
addToMeasuredValue (in module feedbackMeasured), 386
addToWaveLauncher() (built-in function), 586
addTwoFieldSqrSurfMaxHist() (built-in function), 542
addTwoFieldSqrSurfOpHist() (built-in function), 540
addTwoFieldSqrVolMaxHist() (built-in function), 537
addTwoFieldSqrVolOpHist() (built-in function), 535
addVolInnerProd3Hist() (built-in function), 539
addVoltageGainInOscElecFieldHist() (built-in function), 545, 546
addVoltageGainInOscElecFieldHistBase() (built-in function), 544
addWaveLauncher() (built-in function), 585
Adi (in module binaryRecombination), 313
adimacros() (built-in function), 499
adiUpdaters() (built-in function), 499
advectMomentum (in module Fluid), 190
aFac (in module smooth1D), 143
affectedBySpeciesList (in module yeeStaticEmField), 88
algorithm_kind (in module nullSelfCombination), 339
algorithmType (in module nullSelfSplit), 340
allowDatasetReshapeInSerial (in module importFromFileUpdater), 124
alpha, 385
alpha (in module fieldEmitterVelGen), 262
alpha (in module integratedSurfaceFieldSquared), 369
alpha (in module peakSurfaceFieldMagnitude), 372
alpha (in module threeBodyRecombination), 325
alsoAfterRestore, 179
amperePmlUpdater (module), 155
ampereUpdaterName (in module PmlRegion), 180
amplitude, 404
amplitude (in module chirpWavePulse), 414
amplitude (in module gaussian), 418
amplitude (in module gaussianPulse), 419
amplitude (in module halfSinePulse), 420
amplitude (in module planeWavePulse), 424
amplitude (in module sinePlaneWave), 426
amplitudes (in module InitialCondition), 99
anisotropy (in module binaryElastic), 282
anisotropy (in module binaryExcitation), 284
anisotropy (in module binaryReaction), 285
anisotropy (in module chargeExchange), 285
anisotropy (in module dissociativeRecombination), 293
anisotropy (in module impactIonization), 288
anisotropy (in module negativeIonDetachment), 287
annotateFieldOnLine.py command line option
  -C <comp>, –component=<comp>, (integer, optional), 600
  -H <histname>, –historyName=<histname>, (string, required), 600

-a, –positionOffset, (float, optional), 600
-s <simname>, –simulationName=<simname>, (string, required), 600
-w, –overwrite, (flag), 601
-x, –positionMultiplier, (float, optional), 600
annotateSpeciesAbsPtclData2.py command line option
-H <histname>, –historyName=<histname>, (string, required), 601
-T, –threshold, (int, optional), 601
-s <simname>, –simulationName=<simname>, (string, required), 601
-w, –overwrite, (flag), 601
appendAveSteps() (built-in function), 504
applyNumericalCerenkovFilter() (built-in function), 470
applyPeriod, 227
applyPeriod (in module fluxPtclEmitter), 232
applyPeriod (in module gaussDensSrc), 234
applyPeriod (in module gridDenSrcVW), 225
applyPeriod (in module manualSrc), 228
applyPeriod (in module planarPtclEmitter), 235
applyPeriod (in module randdenssrc), 220
applyPML() (built-in function), 565
applyPortAtBoundary_afterAmpere() (built-in function), 561, 568
applyPortAtBoundary_beforeAmpere() (built-in function), 561, 568
applyTimes, 254
applyTimes (in module boostLoader), 229
applyTimes (in module fluxPtclEmitter), 232
applyTimes (in module gaussDensSrc), 234
applyTimes (in module gridDenSrcVW), 225
applyTimes (in module gridLoader), 230
applyTimes (in module manualSrc), 228
applyTimes (in module planarPtclEmitter), 235
applyTimes (in module randdenssrc), 221
Arad (in module binaryRecombination), 313
arbitraryVector (in module Input), 448
atomicName (in module fieldIonization), 324
atomicRatio (in module nullBgAbsorber), 342
autoGenerateTag (in module secondaryEmitter), 245
averageField() (built-in function), 503
averageFieldWindow() (built-in function), 503
axes, 592
axis (in module speciesRmsDistToAxis), 379

B

B (in module RxnRate-expPolyCrossSection), 278
B_FN (in module fieldEmitterVelGen), 264
backgroundPermittivity, 134
backupHistCalc (in module functionOfTime), 388
backupWarningLevel, 390
backupWarningLevel (in module functionOfTime), 388
BaseSolver (in module iterativeSolver), 169
basicEM() (built-in function), 505
bAuxFieldName (in module PmlRegion), 181
bCoeff (in module fieldBinOpUpdater), 119
bCoeff (in module scalarBinOpUpdater), 174
bCoeff (in module scalarFieldBinOpUpdater), 175
Bdi (in module binaryRecombination), 313
beamVelocityGen (module), 260
beta, 385
beta_FN (in module fieldEmitterVelGen), 263, 264
bFac (in module smooth1D), 143
bFieldName (in module PmlRegion), 181
bgElecField (in module Species), 192
bgEmField (in module Species), 192
bgMagField (in module Species), 192
binaryDissociation (module), 319
binaryElastic (module), 281, 317
binaryExcitation (module), 283, 316
binaryIonization (module), 314
binaryOperation (module), 382
binaryReaction (module), 285
binaryRecombination (module), 292, 312
binComponentDescriptions (in module speciesBinning), 393
binComponents (in module speciesBinning), 393
binDimDescriptions (in module speciesBinning), 393
binDims (in module speciesBinning), 393
binOp (in module fieldBinOpUpdater), 118
binOp (in module scalarBinOpUpdater), 173
binOp (in module scalarFieldBinOpUpdater), 174
bitRevDensSrc (module), 222
bitRevDensSrcVW (module), 223
bitRevSlabPosGen (module), 255
bLoop (module), 360
boostLoader (module), 229
boundary, 186
boundaryAtBottomInComponentDir (in module epetraUpdater), 116
boundaryAtBottomInNonComponentDir (in module epetraUpdater), 116
boundaryAtTopInComponentDir (in module epetraUpdater), 116
boundaryAtTopInNonComponentDir (in module epetraUpdater), 117
BoundaryCondition (in module Field), 98
boundaryCondition (in module fieldFunc), 441
boundaryPrecision (in module GridBoundary), 82
bumpReadIter (in module unaryFieldOpUpdater), 145

C

C (in module RxnRate-expPolyCrossSection), 278
C_v_FN (in module fieldEmitterVelGen), 264
C_y_FN (in module fieldEmitterVelGen), 264
cacheFunc (module), 413
calculateSAR() (built-in function), 590
calculateVolume (in module GridBoundary), 81

cellCenterInteriorness (in module gridBoundaryFunc), 443
cellCenterPos (in module gridFunc), 442
cellInsideCentroid (in module gridBoundaryFunc), 443
cellInteriorness (in module gridBoundaryFunc), 443
cellOffset (in module CoordProdSTFuncStencilElement), 163
cellOffset (in module StencilElement), 105
cellOffset (in module STFuncStencilElement), 163
cellOutsideCentroid (in module gridBoundaryFunc), 443
cellSpecies (module), 203
cellsToUpdateAboveDomain (in module FieldUpdater), 104
cellsToUpdateBelowDomain (in module FieldUpdater), 104
cellVolFrac (in module gridBoundaryFunc), 443
cellVolume (in module gridFunc), 442
center (in module fieldOnSemiCircle), 366
centerAmplitude (in module leakychannel), 423
centerSphere (in module kirchhoffSurfaceIntegral), 370
channelPosition (in module leakychannel), 423
channelPosition (in module radialCosChannel), 425
charge (in module fieldIonization), 324
charge (in module fieldIonizationAverageADK), 281
charge (in module fieldIonizationDCADK), 280
charge (in module Fluid), 190
charge (in module Species), 192
chargeDeps (in module Species), 192
chargeExchange (module), 284, 321
chargeNumbers (in module linPlasDielcUpdater), 132
childLangmuirVelGen (module), 261
chirp (in module chirpWavePulse), 414
chirpWavePulse (module), 413
cnEmFieldAlgorithm() (built-in function), 507, 560
cnEpetraUpdaters() (built-in function), 506
cnMatrix() (built-in function), 506
coeff (in module unaryOperation), 383
coefficient (in module integratedSurfaceFieldSquared), 369
coefficient (in module peakSurfaceFieldMagnitude), 372
coefficient (in module powerLawCrossSection), 277
coeffs (in module binaryOperation), 382
collectMethod (in module speciesAbsPtclData2), 391
columnFieldIndex (in module CoordProdSTFuncStencilElement), 163
columnFieldIndex (in module StencilElement), 105
columnFieldIndex (in module STFuncStencilElement), 163
columnInteriorosity (in module nodeStencilFiller), 161
ComboEmField (module), 93
command line option
    -H <histname>, –history1Name=<histname>, (string, required), 625
    -I <histname>, –history2Name<histname>, (string, required), 626
    -O <histname>, –newhistoryName=<histname>, (string, required), 626
    -s <simname>, –simulationName=<simname>, (string, required), 625
    -w, –overwrite, (flag), 626
    -x <operation> , –operation=<operation>, (string, required), 626
Comments (in module LXcatFile), 350
component (in module coordProdSTFuncStencilFiller), 162
component (in module curlUpdater), 106
component (in module curlUpdaterCoordProd), 108
component (in module fieldFunc), 441
component (in module fieldVectorReader), 164
component (in module fieldVectorWriter), 166
component (in module importFromFileUpdater), 123
component (in module nodeFieldVectorReader), 165
component (in module nodeFieldVectorWriter), 167
component (in module nodeStencilFiller), 161
component (in module open), 136
component (in module speciesAbsPtclData), 374
component (in module stencilFiller), 161
component (in module stFuncNodeVectorWriter), 168
component (in module stFuncStencilFiller), 162
component (in module STFuncUpdater), 144
component (in module stFuncVectorWriter), 168
component (in module unaryFieldOpUpdater), 145
component (in module yeeAmpereUpdater), 151
component (in module yeeFaradayUpdater), 153
componentBounds (in module coordProdSTFuncStencilFiller), 162
componentBounds (in module fieldVectorReader), 165
componentBounds (in module fieldVectorWriter), 166
componentBounds (in module nodeFieldVectorReader), 165
componentBounds (in module nodeFieldVectorWriter), 167
componentBounds (in module nodeStencilFiller), 161
componentBounds (in module stencilFiller), 161
componentBounds (in module stFuncNodeVectorWriter), 168
componentBounds (in module stFuncStencilFiller), 162
componentBounds (in module stFuncVectorWriter), 168
components (in module fieldArray), 387
components (in module fieldAtCoords), 362
components (in module fieldAtIndices), 365
components (in module fieldAverage), 363
components (in module FieldMultiUpdater), 155
components (in module InitialCondition), 99
components (in module speciesRmsMomen), 380
components (in module speciesRmsPosition), 380
computeBeam2ModeCoupling.py command line option

-c <comp>, –component=<comp>, (int, required), 601
-f, –frequency, (float, required), 602
-j <curdenfield>, –jName=<curdenfield>, (string, required), 601
-s <simname>, –simulationName=<simname>, (string, required), 601
-t, –startTime, (float, required), 602
-w, –overwrite, (flag), 602
computeCavityG.py command line option
-b <bd>, –beginDump=<bd>, (int, optional), 602
-e <ed>, –endDump=<ed>, (int, optional), 602
-g <geometry>, –geometryName=<geometry>, (string), 602
-s <simname>, –simulationName=<simname>, (string, required), 602
computeCurlDt() (built-in function), 509
computeDebyeLength.py command line option
-S <spname>, –speciesName=<spname>, (string, required), 602
-s <simname>, –simulationName=<simname>, (string, required), 602
-w, –overwrite, (flag), 603
computeEmittanceFromDump.py command line option
-S <spname>, –speciesName=<spname>, (string, required), 603
-X, –x2plane, (float, optional), 603
-m, –movingWindow, (int, required), 603
-n, –numSlices, (int, optional), 603
-s <simname>, –simulationName=<simname>, (string, required), 603
-v, –windowSpeed, (float, optional), 603
-w, –overwrite, (flag), 603
-x, –x1plane, (float, optional), 603
computeEmittanceOnPlane.py command line option
-H <histname>, –historyName=<histname>, (string, required), 606
-S <spname>, –speciesName=<spname>, (string, required), 606
-s <simname>, –simulationName=<simname>, (string, required), 606
-w, –overwrite, (flag), 606
computeFarFieldFromKirchhoffBox.py command line option
-f, –frequency, (float), 607
-g, –getFourierComponent, (int), 607
-i, –simpsonIntegration, (int), 608
-l <fldlabel>, –fieldLabel=<fldlabel>, (string, optional), 607
-n, –timeStepStride, (int), 607
-p, –numPhi, (int), 607
-r, –farFieldRadius, (float), 607
-s <simname>, –simulationName=<simname>, (string, required), 607
-t, –numTheta, (int), 607
-v, –varyingRadiusMesh, (int), 608
-w, –overwrite, (flag), 608
-x, –zeroPhiDirection, (string), 607
-z, –zeroThetaDirection, (string), 607
computeFarFieldRadiation.py command line option
-H <histname>, –historyName=<histname>, (string, required), 608
-e, –numTheta, (int, required), 608
-g, –normalize, (int, optional), 608
-n, –SLL, (float, true), 608
-p, –numPhi, (int, required), 608
-s <simname>, –simulationName=<simname>, (string, required), 608
-t, –numTime, (int, optional), 608
-w, –overwrite, (flag), 609
computeInverseQ.py command line option
-H <histname>, –historyName=<histname>, (string), 609
-O <outname>, –outputFileName=<outname>, (string, optional), 609
-c <comp>, –component=<comp>, (int), 609
-f, –frequency, (float), 609
-s <simname>, –simulationName=<simname>, (string, required), 609
-w, –overwrite, (flag), 609
computeModes, 591
computePtclLimits.py command line option
-O <outname>, –outputFileName=<outname>, (string, optional), 610
-S <spname>, –speciesName=<spname>, (string, required), 609
-s <simname>, –simulationName=<simname>, (string, required), 609
-w, –overwrite, (flag), 610
computePtclNumDensity.py command line option
-N, –avgNxN, (int), 610
-S <spname>, –speciesName=<spname>, (string, required), 610
-i, –iterateAvg, (int), 610
-s <simname>, –simulationName=<simname>, (string, required), 610
-w, –overwrite, (flag), 610
computeS11Parameters.py command line option
-E, –nElements, (int, optional), 611
-c, –dt, (float, required), 611
-e, –nPoints, (int, optional), 611
-n, –nZeros, (int, optional), 611
-p <histpat>, –historyPATCH=<histpat>, (string, required), 611
-s <simname>, –simulationName=<simname>, (string, required), 611
-t <histcal>, –historyCAL=<histcal>, (string, required), 611

-w, –overwrite, (flag), 611
computeSParamsFromHists.py command line option
-2, –outSign, (int, optional), 612
-B <outmag>, –outSlabB=<outmag>, (string, required), 612
-D, –outDirection, (int, required), 612
-E <outelec>, –outSlabE=<outelec>, (string, required), 612
-L, –maxWavelength, (float, required), 612
-O, –stepOffset, (int, optional), 611
-S, –minWavelength, (float, required), 612
-b <inmag>, –inSlabB=<inmag>, (string, required), 612
-d, –inDirection, (int, required), 612
-e <inelec>, –inSlabE=<inelec>, (string, required), 612
-f, –firstStep, (int, optional), 611
-l, –inSign, (int, optional), 612
-l, –lastStep, (int, optional), 611
-n <outname>, –outputFileName=<outname>, (string, optional), 612
-s <simname>, –simulationName=<simname>, (string, required), 611
-w, –overwrite, (flag), 612
-x <outsufx>, –outputSuffix=<outsufx>, (string, optional), 612
computeSParamsViaOverlapIntegral.py command line option
-1, –inSign, (int, required), 613
-2, –outSign, (int, required), 613
-B <outmag>, –outSlabB=<outmag>, (string, required), 613
-D, –direction, (int, required), 613
-E <outelec>, –outSlabE=<outelec>, (string, required), 613
-L, –maxWavelength, (float, required), 613
-O, –stepOffset, (int, optional), 612
-S, –minWavelength, (float, required), 613
-b <inmag>, –inSlabB=<inmag>, (string, required), 613
-e <inelec>, –inSlabE=<inelec>, (string, required), 613
-f, –firstStep, (int, optional), 612
-l, –lastStep, (int, optional), 612
-n <outname>, –outputFileName=<outname>, (string, optional), 613
-s <simname>, –simulationName=<simname>, (string, required), 612
-w, –overwrite, (flag), 613
-x <outsufx>, –outputSuffix=<outsufx>, (string, optional), 613
computeSpectrogram.py command line option
-F, –maxFreq, (float, required), 614
-H <histname>, –historyName=<histname>, (string, required), 613
-W, –windowType, (integer, required), 614
-c <comp>, –component=<comp>, (integer, optional), 614
-f, –fourierTransSize, (integer, required), 613
-n, –noOverlap, (integer, required), 613
-s <simname>, –simulationName=<simname>, (string, required), 613
-w, –overwrite, (flag), 614
computeTimeSeriesAmplitude.py command line option
-H <histname>, –historyName=<histname>, (string, required), 614
-c <comp>, –component=<comp>, (int, optional), 614
-f, –frequency, (float, required), 614
-n <outname>, –outputFileName=<outname>, (string, optional), 614
-s <simname>, –simulationName=<simname>, (string, required), 614
-w, –overwrite, (flag), 614
computeTimeSeriesFrequency.py command line option
-0, –offsetx0, (float, optional), 616
-1, –axis, (float, optional), 616
-B, –beta, (float, required), 615
-H <histname>, –historyName=<histname>, (string, required), 615
-a, –axis, (int, true), 615
-b <bd>, –beginDump=<bd>, (int, optional), 615
-c <comp>, –component=<comp>, (int, optional), 615
-e <ed>, –endDump=<ed>, (int, optional), 615
-f, –frequency, (float, required), 615
-n <outname>, –outputFileName=<outname>, (string, optional), 615
-s <simname>, –simulationName=<simname>, (string, required), 615
-w, –overwrite, (flag), 615, 616
consolidatePtclGrps (in module Species), 193
constant (module), 434
constantCrossSection (module), 276
constantDensity (in module Fluid), 191
constCurlDt() (built-in function), 510
constructModes, 591
contractFromBottomInComponentDir (in module FieldUpdater), 103
contractFromBottomInNonComponentDir (in module FieldUpdater), 103
convectDensity (in module Fluid), 190
convergenceMetric (in module iterativeSolver), 170
convertFieldComponentCartToCylX.py command line option
-f, –fieldName, (string), 616

-s <simname>, –simulationName=<simname>, (string, required), 616
-w, –overwrite, (flag), 616
convertFieldComponentCartToCylZ.py command line option
-f, –FieldName, (string), 617
-s <simname>, –simulationName=<simname>, (string, required), 617
-w, –overwrite, (flag), 617
convertPtclComponentsCartToCylX.py command line option
-S <spname>, –speciesName=<spname>, (string, required), 617
-o <outspname>, –outputSpeciesName=<outspname>, (string), 617
-s <simname>, –simulationName=<simname>, (string, required), 617
-w, –overwrite, (flag), 617
convertPtclComponentsCartToCylZ.py command line option
-S <spname>, –speciesName=<spname>, (string, required), 618
-o <outspname>, –outputSpeciesName=<outspname>, (string), 618
-s <simname>, –simulationName=<simname>, (string, required), 618
-w, –overwrite, (flag), 618
coordProdLaplacian() (built-in function), 582
coordProdNodeLaplacian() (built-in function), 511
CoordProdSTFuncStencilElement (in module coordProdSTFuncStencilFiller), 162
CoordProdSTFuncStencilElement (module), 163
CoordProdSTFuncStencilElementMacro() (built-in function), 478, 558
coordProdSTFuncStencilFiller (module), 162
copyFromAuxJ() (built-in function), 575
copyHistoryAtEachDump (in module globalvariables), 73
copyToAuxJ() (built-in function), 575
copyUniformInDir (in module importFromFileUpdater), 123
copyxshft() (built-in function), 577
correctDensity (in module Fluid), 190
cosineFlattop (module), 415
cosineRamp (module), 416
createMissingPtclsDumps.py command line option
-S <spname>, –speciesName=<spname>, (string, required), 618
-s <simname>, –simulationName=<simname>, (string, required), 618
-w, –overwrite, (flag), 619
createParticleTracks.py command line option
-S <spname>, –speciesName=<spname>, (string, required), 619
-p, –ptclIndex, (int, required), 619
-s <simname>, –simulationName=<simname>, (string, required), 619
-w, –overwrite, (flag), 619
createRegion_DDLD() (built-in function), 493
createRequiredBlocks() (built-in function), 569
crossSecScaleFactor (in module negativeIonDetachment), 337
crossSection, 311
crossSection (in module binaryDissociation), 320
crossSection (in module binaryElastic), 318
crossSection (in module binaryExcitation), 316
crossSection (in module binaryIonization), 315
crossSection (in module binaryRecombination), 313
crossSection (in module chargeExchange), 321
crossSection (in module constantCrossSection), 276
crossSection (in module electronAttachment), 338
crossSection (in module impactElastic), 327
crossSection (in module impactExcitation), 329
crossSection (in module impactIonCollisions), 331
crossSection (in module impactIonization), 334
crossSection (in module negativeIonDetachment), 336
crossSection (in module threeBodyRecombination), 326
crossSectionDataFile (in module binaryExcitation), 317
crossSectionDataFile (in module impactElastic), 328
crossSectionDataFile (in module impactExcitation), 330
crossSectionDataFile (in module impactIonization), 335
crossSectionVariable, 312
crossSectionVariable (in module binaryDissociation), 320
crossSectionVariable (in module binaryElastic), 318
crossSectionVariable (in module binaryExcitation), 317
crossSectionVariable (in module binaryIonization), 316
crossSectionVariable (in module binaryRecombination), 314
crossSectionVariable (in module chargeExchange), 322
crossSectionVariable (in module electronAttachment), 338
crossSectionVariable (in module impactElastic), 328
crossSectionVariable (in module impactExcitation), 330
crossSectionVariable (in module impactIonCollisions), 332
crossSectionVariable (in module impactIonization), 335
crossSectionVariable (in module negativeIonDetachment), 337
crossSectionVariable (in module threeBodyRecombination), 326
crossSectionVariable (in module twoColumnFile), 279
CurlCurl() (built-in function), 583
curlUpdater (module), 105
curlUpdaterCoordProd (module), 108
currDeps (in module Species), 192
current (in module planarPtclEmitter), 236

currentDensityFunc (in module planarPtclEmitter), 236
cutCellPosGen (module), 256
cutDir (in module fieldOnMovingPlane), 367
cutVal (in module fieldOnMovingPlane), 367
cylEdgeToNodeVec (module), 109

D

D (in module RxnRate-expPolyCrossSection), 278
dataset (in module importFromFileUpdater), 122
DatasetAttrib (in module speciesBinning), 394
datasetLowerBounds (in module importFromFileUpdater), 123
datasetUpperBounds (in module importFromFileUpdater), 123
decay (module), 291
decayLifetime (module), 281
Decomp (module), 79
decomp0 (in module Decomp), 80
decomp1 (in module Decomp), 80
decomp2 (in module Decomp), 80
delaySteps (in module fluxPtclEmitter), 232
delta, 385
densGrad_x (in module maxwellian), 430
densGrad_y (in module maxwellian), 430
densGrad_z (in module maxwellian), 430
density (in module bitRevDensSrc), 222
density (in module bitRevDensSrcVW), 223
density (in module fluxPtclEmitter), 232
density (in module gaussDensSrc), 234
density (in module gridDenSrcVW), 225
density (in module maxwellian), 430
density (in module planarPtclEmitter), 235
density (in module randdenssrc), 220
depField (in module ScalarDepositor), 187
depField (in module VectorDepositor), 188
depleteBackgroundGas (in module impactElastic), 327
depleteBackgroundGas (in module impactExcitation), 329
depleteBackgroundGas (in module impactIonCollisions), 331
depleteBackgroundGas (in module impactIonization), 334
depleteBackgroundGas (in module negativeIonDetachment), 336
depositCurrentFromCorner (in module secondaryEmitter), 244
depositCurrentThoughFieldsNotUsed (in module Species), 192
depositCurrentToCorner (in module absSavTriCutCell), 210
desiredHistory, 384
desiredResid (in module epetraUpdater), 116
desiredResid (in module poissonUpdater), 139
detachedElectronSpecies (in module negativeIonDetachment), 337
detachedNeutralSpecies (in module negativeIonDetachment), 337
detachmentThreshold (in module negativeIonDetachment), 337
deyMittraUpdater (module), 111
deyMittraYee() (built-in function), 507
deyMittraYeeInterpol() (built-in function), 508
diagFac (in module epetraUpdater), 116
diagVar, 377
dielectricConstXX (in module setEpsilonUpdater), 141
dielectricConstXY (in module setEpsilonUpdater), 141
dielectricConstXZ (in module setEpsilonUpdater), 142
dielectricConstYX (in module setEpsilonUpdater), 141
dielectricConstYY (in module setEpsilonUpdater), 141
dielectricConstYZ (in module setEpsilonUpdater), 141
dielectricConstZX (in module setEpsilonUpdater), 142
dielectricConstZY (in module setEpsilonUpdater), 142
dielectricConstZZ (in module setEpsilonUpdater), 141
DielectricShape, 134
differencing (in module curlUpdater), 106
differencing (in module curlUpdaterCoordProd), 108
differencing (in module gradVecUpdater), 121
differencing (in module Multifield-divUpdater), 112
diffuseBndry (module), 213
dimension (in module globalvariables), 71
dir (in module gridBoundaryFunc), 444, 445
dir (in module gridFunc), 442
dir (in module Multifield-malUpdater), 133
dir (in module speciesDataOnPlane), 382
direction (in module cosineFlattop), 415
direction (in module cosineRamp), 416
direction (in module diffuseBndry), 213
direction (in module fieldOnMovingPlane), 367
direction (in module leakychannel), 423
direction (in module potentialUpdater), 140
direction (in module radialCosChannel), 425
direction (in module secElec), 239
direction (in module specularBndry), 214
direction (in module sputter), 248
direction (in module transparentBndry), 215
direction (in module userDefinedSecElec), 249
direction (in module userFuncSecElec), 250
dirichletBC() (built-in function), 579
dirichletFromField() (built-in function), 580
displayModes, 591
dissociationEnergy (in module dissociativeIonization), 290
dissociationEnergy (in module electronImpactDissociation), 294
dissociativeIonization (module), 289
dissociativeRecombination (module), 293
distributedRandom (module), 435

divUpdaterCoordProd (module), 112
dmAdiUpdaters() (built-in function), 499
dmFrac (in module GridBoundary), 82
dMYee() (built-in function), 507
dnSortMax (in module globalvariables), 73
dnSortMin (in module globalvariables), 73
doComparison (in module distributedRandom), 435
doComparison (in module randOAFunc), 409
doShiftLoad (in module bitRevDensSrcVW), 224
doShiftLoad (in module fluxPtclEmitter), 232
doShiftLoad (in module gaussDensSrc), 234
doShiftLoad (in module gridDenSrcVW), 225
doShiftLoad (in module planarPtclEmitter), 235
downShiftDir (in module globalvariables), 73
downShiftPos (in module globalvariables), 74
dt (in module globalvariables), 71
dtCoefficients (in module curlUpdater), 106
dtCoefficients (in module curlUpdaterCoordProd), 108
dtCoefficients (in module linIterUpdater), 131
dtCoefficients (in module STFuncUpdater), 144
dtCoefficients (in module unaryFieldOpUpdater), 146
dtCoefficients (in module unaryScalarOpUpdater), 176
dtCoefficients (in module yeeAmpereDielVecUpdater), 150
dtFar (in module kirchhoffSurfaceIntegral), 370
dualCellInteriorness (in module gridBoundaryFunc), 443
dualEdgeInteriorness (in module gridBoundaryFunc), 444
dualEdgesInteriorness (in module gridBoundaryFunc), 443
dualFaceInteriorness (in module gridBoundaryFunc), 444
dualFacesInteriorness (in module gridBoundaryFunc), 443
dummyUpdater (module), 113
dumpField (in module ComboEmField), 93
dumpOnly (in module Field), 98
dumpPeriodicity (in module Field), 97
dumpPeriodicity (in module globalvariables), 71
dumpPeriodicity (in module GridBoundary), 82
dumpPeriodicity (in module Species), 193
dumpPotential (in module yeeStaticEmField), 91
dumpSteps (in module globalvariables), 71
dumpTrack (in module Fluid), 191

E

eAuxFieldName (in module PmlRegion), 181
edgeBordersDmNeglectedFace (in module gridBoundaryFunc), 445
edgeCenterPos (in module gridFunc), 442
edgeInteriorness (in module gridBoundaryFunc), 444
edgeLengthFrac (in module gridBoundaryFunc), 445
edgesInteriorness (in module gridBoundaryFunc), 443
edgeToNodeVec (module), 114
eFieldName (in module PmlRegion), 180
elecField, 86
electonIonization (module), 288
electronAttachment (in module Fluid-neutralGas), 347
electronAttachment (module), 286, 338
electronImpactDissociation (module), 294
electrons (in module binaryDissociation), 319
electrons (in module binaryIonization), 315
electrons (in module binaryRecombination), 313
electrons (in module fieldIonization), 324
electrons (in module nullFieldIonization), 345
electronScatter (module), 282
element, 592
emField (in module Species), 192
EmField (module), 83
EMField-funcEmField (module), 87
emField1 (in module ComboEmField), 93
emField2 (in module ComboEmField), 93
EMfieldEnergy (module), 361
emissionDirection (in module secondaryEmitter), 244
emissionOffset (in module cutCellPosGen), 256
emissionProb (in module secElec), 239
emissionProb (in module userFuncSecElec), 251
emissionQuad (in module beamVelocityGen), 260
emissionVelocityInUnitsOfC (in module userFuncSecElec), 251
emitBasedOnLocalForce, 254
emitDim (in module fluxPtclEmitter), 231
emitDistanceShift (in module fluxPtclEmitter), 231
emitForPrimariesWithoutSurfaceNormal (in module secondaryEmitter), 245
emitIntoGridBoundaryInterior (in module secondaryEmitter), 244
emitPlaneLocation (in module fluxPtclEmitter), 231
emitSign, 254
emitSign (in module bitRevSlabPosGen), 256
emitSign (in module gridPosGen), 258
emitterBoundary (in module cutCellPosGen), 256
emitterMask (in module cutCellPosGen), 256
emitterType (in module fieldEmitterVelGen), 262
emittingSurface (in module secElec), 239
emittingSurface (in module secFieldEmitter), 240
emittingSurface (in module userFuncSecElec), 251
emMultiField (module), 87
emType (in module yeeConductorUpdater), 159
endAmplitude (in module cosineFlattop), 415
endAmplitude (in module cosineRamp), 416
endAmplitude (in module radialCosChannel), 426
endFlattop (in module cosineFlattop), 415
endPosition (in module acceleratingVoltage), 360
endPosition (in module cosineFlattop), 415
endPosition (in module cosineRamp), 416
endRadius (in module radialCosChannel), 426
energy (in module fieldIonization), 325
energy (in module nullFieldIonization), 345

energyLoss (in module binaryElastic), 282
energyLoss (in module chargeExchange), 285
energyWritePeriod (in module PmlRegion), 181
enforceZeroNetCharge (in module yeeStaticEmField), 91
entryToBool() (built-in function), 461
envBoris, 198
envelopeFields, 198
envelopeMultiField, 198
epetraUpdater (module), 114
epsilon (in module Fluid), 190
esSolveDirichlet() (built-in function), 515
esSolveDirichletFromField() (built-in function), 516
eulerFluid (in module Fluid), 190
excAtomSpecies (in module impactExcitation), 330
excitationEnergy (in module binaryExcitation), 284
excitationThreshold (in module impactExcitation), 330
expandAboveTopInComponentDir (in module FieldUpdater), 103
expandAboveTopInNonComponentDir (in module FieldUpdater), 103
expandBelowBottomInComponentDir (in module FieldUpdater), 103
expandBelowBottomInNonComponentDir (in module FieldUpdater), 104
expandToTopInComponentDir (in module FieldUpdater), 103
expandToTopInNonComponentDir (in module FieldUpdater), 103
expanFunc (in module leakychannel), 423
exponent (in module powerLawCrossSection), 277
exportSpecies.py command line option
- -N, –numOutFiles, (int), 620
- -O <outname>, –outputFileName=<outname>, (string, optional), 620
- -P, –applyPeriod, (int), 620
- -R, –randomizePtcls, (int), 620
- -S <spname>, –speciesName=<spname>, (string, required), 619
- -d, –dumpNum, (int), 619
- -o <outbasename>, –outputBaseName=<outbasename>, (string), 620
- -s <simname>, –simulationName=<simname>, (string, required), 619
- -w, –overwrite, (flag), 620

expression (in module feedbackSTFunc), 418
expression (in module historySTFunc), 422
expression (in module OAFunc-expression), 411
Expression (in module secFieldEmitter), 240
expression (in module STFunc-expression), 417
expression (in module UserFunc-expression), 436
Expression (in module userFuncExpression), 428
Expression (in module userFuncSecElec), 250
exteriorGridBoundaryDirichletBC() (built-in function), 513
extractModes.py command line option
- -b <bd>, –beginDump=<bd>, (int, optional), 620
- -c <con>, –construct=<con>, (int), 621
- -e <ed>, –endDump=<ed>, (int, optional), 620
- -f <fld>, –field=<fld>, (string, required), 620
- -m <nummodes>, –nModes=<nummodes>, (int, required), 620
- -r <ranpts>, –numberRandomPts=<ranpts>, (int), 620
- -s <simname>, –simulationName=<simname>, (string, required), 620
- -t <samptype>, –sampleType=<samptype>, (int, optional), 620
- -u <unipts>, –numberUniformPts=<unipts>, (int), 620
- -w, –overwrite, (flag), 621

extractModesViaOperator.py command line option
- -B <B>, –magneticField=<B>, (string, required), 623
- -C <con>, –construct=<con>, (int, optional), 623
- -E <E>, –electricField=<E>, (string, required), 623
- -S <cs>, –cellSamples=<cs>, (string, required), 623
- -c <cut>, –cutoff=<cut>, (float, required), 623
- -d <dr>, –dumpRange=<dr>, (string, required), 623
- -o <op>, –operator=<op>, (string, required), 623
- -s <simname>, –simulationName=<simname>, (string, required), 622
- -w, –overwrite, (flag), 623

F

faceAreaFrac (in module gridBoundaryFunc), 444
faceCenterPos (in module gridFunc), 442
faceInsideCentroid (in module gridBoundaryFunc), 444
faceInteriorness (in module gridBoundaryFunc), 444
faceIsDmNeglected (in module gridBoundaryFunc), 445
faceOutsideCentroid (in module gridBoundaryFunc), 444
facesAreDmNeglected (in module gridBoundaryFunc), 444
facesInteriorness (in module gridBoundaryFunc), 443
faceToNodeVec (module), 117
factor (in module divUpdaterCoordProd), 113
factor (in module fieldSqrDiagUpdater), 157
factor (in module gradVecUpdater), 121
factor (in module gradVecUpdaterCoordProd), 122
factor (in module Multifield-divUpdater), 112
factor (in module poissonUpdater), 139
factor (in module potentialUpdater), 140
faradayUpdaterName (in module PmlRegion), 180
feedback (in module feedbackSTFunc), 418
feedback (in module historySTFunc), 422
feedbackHist, 384
feedbackHist (in module feedbackMeasured), 386

feedbackMeasured (module), 385
feedbackSTFunc (module), 417
field (in module acceleratingVoltage), 360
field (in module bLoop), 361
field (in module fieldArray), 387
field (in module fieldAtCoords), 362
field (in module fieldAtIndices), 365
field (in module fieldAverage), 363
field (in module fieldFunc), 440
field (in module History), 357
field (in module integratedSurfaceFieldSquared), 369
field (in module peakSurfaceFieldMagnitude), 372
field (in module pseudoPotential), 372
Field (module), 96
field2ScalarUpdater (module), 172
field_enhancement (in module fieldEmitterVelGen), 262
fieldArray (module), 387
fieldAtCoords (module), 362
fieldAtIndices (module), 364
fieldAverage (module), 363
fieldBinOpUpdater (module), 118
fieldEmitterVelGen (module), 261
fieldEnergy (module), 363
fieldFunc (module), 440
fieldIonization (module), 290, 323
fieldIonizationAverageADK (module), 280
fieldIonizationDCADK (module), 279
FieldMatrix (in module linIterUpdater), 131
fieldMomen (module), 365
FieldMultiUpdater (module), 154
fieldOnLine (module), 367
fieldOnMovingPlane (module), 366
fieldOnSemiCircle (module), 366
fields (in module EMfieldEnergy), 362
fields (in module fieldEnergy), 364
fields (in module fieldMomen), 365
fields (in module fieldOnLine), 368
fields (in module fieldOnMovingPlane), 367
fields (in module fieldOnSemiCircle), 366
fields (in module History), 357
fields (in module History-fieldPoyn), 368
fields (in module kirchhoffSurfaceIntegral), 370
fields (in module Species), 192
fieldScaleEndPoints (in module fieldScaleVelGen), 266
fieldScaleIncrement (in module fieldScaleVelGen), 266
fieldScaleValues (in module fieldScaleVelGen), 267
fieldScaleVelGen (module), 266
fieldSqrDiagUpdater (module), 156
FieldUpdater (module), 101
fieldValue (in module fieldFunc), 441
fieldVectorReader (module), 164
fieldVectorWriter (module), 166
file, 227
file (in module manualSrc), 228
file (in module twoColumnFile), 279
fileExtension, 227
filename, 411
fileName (in module importFromFileUpdater), 122
filename (in module LXcatFile), 349
fileNameBase, 227
fillGeoCad() (built-in function), 531
fillGeoFastCAD() (built-in function), 531
fillGeoPython() (built-in function), 532
filter() (built-in function), 517
finalize() (built-in function), 489
finalizeParticles() (built-in function), 482
finalizeUpdaters() (built-in function), 490, 491
flattopAmplitude (in module cosineFlattop), 415
floattype (in module globalvariables), 71
fluid (in module nullBgAbsorber), 341
Fluid (module), 189
Fluid-neutralGas (module), 346
fluidTemperature (in module nullBgAbsorber), 342
fluxDensityFunc (in module planarPtclEmitter), 236
fluxFileReadPeriod (in module fluxPtclEmitter), 231
fluxPtclEmitter (module), 231
fluxSpeciesDumpName (in module absSaveDump), 211
fluxSpeciesDumpName (in module fluxPtclEmitter), 231
fmax (in module OAFunc-expression), 411
fmin (in module OAFunc-expression), 411
force1D (in module electronAttachment), 338
force1D (in module impactElastic), 328
force1D (in module impactExcitation), 330
force1D (in module impactIonization), 334
force1D (in module nullSelfCombination), 340
forceField, 198
fourPassAndReloc() (built-in function), 573
fourPassAndRelocScalar() (built-in function), 574
frac (in module Multifield-malUpdater), 133
freeRel, 198
freeRelVW, 198
frequency (in module fieldIonization), 324
frequency (in module nullFieldIonization), 345
fullScatteringBox_ApplyB() (built-in function), 573
fullScatteringBox_ApplyD() (built-in function), 573
fullScatteringBox_Region() (built-in function), 571, 572
fullScatteringBox_Updaters() (built-in function), 571, 573
fullScatteringBox_Wave() (built-in function), 571, 572
functionOffset (in module CoordProdSTFuncStencilElement), 163
functionOffset (in module STFuncStencilElement), 163
functionOfTime (module), 387
functionVariables (in module secondaryEmitter), 244
funcVelGen (module), 267

G

gamma, 385

gammaBoost (in module boostLoader), 229
gammaVec (in module potentialUpdater), 140
gasKind (in module Fluid), 191
gaussDensSrc (module), 234
gaussian (module), 418
gaussianPulse (module), 419
gaussSource (module), 228
geoBiParabolicSlabXP() (built-in function), 520
geoBoxP() (built-in function), 519
geoConeXP() (built-in function), 522
geoCylinderXP() (built-in function), 520
geoEllipsoid() (built-in function), 526
geoEllipticalConeXP() (built-in function), 529
geoEllipticalCylinderXP() (built-in function), 526
geoHemiEllipsoidXP() (built-in function), 526
geoHemiSphereXP() (built-in function), 523
geometricQuantity (in module geometryUpdater), 120
geometryUpdater (module), 119
geoParaboloidXP() (built-in function), 523
geoPipeXP() (built-in function), 522
geoQuadrilateralSlabXP() (built-in function), 519
geoRndCylinderXP() (built-in function), 523
geoRndRectangleSlabXP() (built-in function), 526
geoSphere() (built-in function), 523
geoTorusX() (built-in function), 523
geoTriangleSlabXP() (built-in function), 520
getBoundaryOffsets() (built-in function), 463
getPrevStep (in module yeeStaticEmField), 92
getTagsFromAbsorber (in module speciesTrackTag), 381
getTagsFromSpecies (in module speciesTrackTag), 381
globalvariables (module), 71
gradVecUpdater (module), 121
gradVecUpdaterCoordProd (module), 121
grid (in module gridFunc), 442
Grid (module), 74
gridBndryFuncIsSmooth (in module GridBoundary), 82
gridBoundary (in module absCutCell), 207
gridBoundary (in module absSavCutCell), 209
gridBoundary (in module absSavStairStep), 212
gridBoundary (in module absSavTriCutCell), 210
gridBoundary (in module absStairStep), 212
gridBoundary (in module curlUpdater), 106
gridBoundary (in module cutCellPosGen), 257
gridBoundary (in module deyMittraUpdater), 111
gridBoundary (in module EMfieldEnergy), 362
gridBoundary (in module fieldFunc), 441
gridBoundary (in module gridBoundaryFunc), 443
gridBoundary (in module integratedSurfaceField-Squared), 369
gridBoundary (in module Multifield-divUpdater), 112
gridBoundary (in module MultiFiled-deyMittraConstrainUpdater), 110
gridBoundary (in module nodeFieldVectorReader), 165
gridBoundary (in module nodeFieldVectorWriter), 167
gridBoundary (in module nodeStencilFiller), 161
gridBoundary (in module nullBgAbsorber), 343
gridBoundary (in module peakSurfaceFieldMagnitude), 372
gridBoundary (in module secElec), 239
gridBoundary (in module secFieldEmitter), 240
gridBoundary (in module secondaryEmitter), 244
gridBoundary (in module stFuncNodeVectorWriter), 168
gridBoundary (in module unaryFieldOpUpdater), 146
gridBoundary (in module yeeAmpereUpdater), 151
gridBoundary (in module yeeFaradayUpdater), 153
GridBoundary (module), 81
gridBoundaryDirichletBC() (built-in function), 512
gridBoundaryFunc (module), 442
gridDenSrcVW (module), 225
gridDirection (in module CoordProdSTFuncStencilElement), 163
gridFunc (module), 442
gridLoader (module), 229
gridOffset (in module CoordProdSTFuncStencilElement), 164
gridPosGen (module), 258

H

halfSinePulse (module), 420
hasB, 86
hasB (in module ComboEmField), 93
hasB (in module emMultiField), 87
hasB (in module yeeStaticEmField), 88
haveNormalDeriv (in module kirchhoffSurfaceIntegral), 370
histCalc, 389
histCalc (in module functionOfTime), 388
histories (in module binaryOperation), 382
history, 373
history (in module historyFunc), 445
history (in module unaryOperation), 383
History (module), 356
History-fieldPoyn (module), 368
historyFunc (module), 445
historySTFunc (module), 421
hornAntenna() (built-in function), 500

I

I (in module RxnRate-expPolyCrossSection), 278
iComponents (in module speciesTrackTag), 381
ignoreProb (in module secElec), 239
ignoreProb (in module sputter), 248
ignoreProb (in module userFuncSecElec), 252
impactElastic (in module Fluid-neutralGas), 346
impactElastic (module), 326
impactExcitation (module), 328
impactIonCollisions (in module Fluid-neutralGas), 346
impactIonCollisions (module), 331

impactIonization (in module Fluid-neutralGas), 346
impactIonization (module), 287, 333
impactSpecies (in module electronAttachment), 338
impactSpecies (in module impactElastic), 327
impactSpecies (in module impactExcitation), 329
impactSpecies (in module impactIonCollisions), 331
impactSpecies (in module impactIonization), 334
impactSpecies (in module negativeIonDetachment), 336
impactSpecies (in module threeBodyRecombination), 325
importFromFileUpdater (module), 122
includeCylAxis (in module curlUpdaterCoordProd), 109
includeCylAxis (in module divUpdaterCoordProd), 113
includeCylAxis (in module gradVecUpdaterCoordProd), 122
includeDamping (in module linPlasDielcUpdater), 132
independentDumpIndicing (in module absSaveDump), 211
index (in module historyFunc), 445
indexAndResult (in module speciesBinning), 393
inElectrons (in module threeBodyRecombination), 325
inIons (in module chargeExchange), 321
inIons (in module threeBodyRecombination), 325
initBeam() (built-in function), 552
initBeamWithOpenBdry() (built-in function), 552
InitialCondition (in module Field), 98
initialCondition (in module Scalar), 172
InitialCondition (module), 98
initialDumpIndex (in module fluxPtclEmitter), 231
initialize_DDLD() (built-in function), 496
initialLoc (in module fieldOnMovingPlane), 367
initialPlanePos (in module speciesDataOnPlane), 382
inNeutrals (in module chargeExchange), 321
inNeutrals (in module impactIonization), 334
inNeutralsGasType (in module impactIonization), 334
input (in module fieldIonization), 324
input (in module nullFieldIonization), 345
Input (in module UserFunc-expression), 436
Input (module), 447
inputOrder (in module UserFunc-expression), 436
insidePermittivity (in module permittivityUpdater), 137
insidePermittivityDiag (in module permittivityUpdater), 137
insidePermittivityOffDiag (in module permittivityUpdater), 137
inSpecies (in module binaryDissociation), 319
inSpecies (in module binaryIonization), 315
inSpecies (in module binaryRecombination), 313
inSpeciesA, 311
inSpeciesA (in module binaryElastic), 318
inSpeciesA (in module binaryExcitation), 316
inSpeciesAtomicWeight, 311
inSpeciesB, 311
inSpeciesB (in module binaryElastic), 318
inSpeciesB (in module binaryExcitation), 316
inSpeciesName, 311
integratedSurfaceFieldSquared (module), 369
intElecMass (in module binaryDissociation), 319
intElecMass (in module binaryElastic), 318
intElecMass (in module binaryExcitation), 316
intElecMass (in module binaryIonization), 315
intElecMass (in module binaryRecombination), 313
interiorness (in module curlUpdater), 106
interiorness (in module Multifield-divUpdater), 112
interiorness (in module unaryFieldOpUpdater), 146
interiorness (in module yeeAmpereUpdater), 151
interiorness (in module yeeFaradayUpdater), 153
interiorosity (in module nodeFieldVectorReader), 165
interiorosity (in module nodeFieldVectorWriter), 167
interiorosity (in module stFuncNodeVectorWriter), 168
interpolation (in module Field), 97
interpolationOrder (in module fieldFunc), 441
interval, 592
intSpecMass (in module binaryDissociation), 319
inverseFunc (module), 422
invPermittivityField (in module permittivityUpdater), 137
ionCollType (in module impactIonCollisions), 331
ionizationEnergy (in module electonIonization), 289
ionizationEnergy (in module fieldIonizationAverageADK), 281
ionizationEnergy (in module fieldIonizationDCADK), 280
ionizationEnergy (in module impactIonization), 288
ionizationKind (in module fieldIonization), 324
ionizationKind (in module nullFieldIonization), 345
ionizationThreshold (in module impactIonization), 335
ionizedElectronSpecies (in module impactIonization), 334
ions (in module fieldIonization), 324
ions (in module nullFieldIonization), 345
ionSpecies (in module electronAttachment), 338
ionSpecies (in module impactIonization), 334
irrelevantDirs (in module cacheFunc), 413
isDissociation, 323
isEqualString() (built-in function), 461
isFlipped, 186
isHole, 200, 202
isNeutralGasFluid (in module impactElastic), 327
isNeutralGasFluid (in module impactExcitation), 329
isNeutralGasFluid (in module impactIonization), 334
isNeutralGasFluid (in module negativeIonDetachment), 336
isNotEqualInt() (built-in function), 461
iterativeSolver (module), 169

K

k (in module gaussianPulse), 419
k (in module halfSinePulse), 420

k (in module planeWavePulse), 424
k (in module sinePlaneWave), 426
kappaVelGen (module), 268
kappaX (in module amperePmlUpdater), 156
kappaX (in module perfDispPmlUpdater), 158
kappaY (in module amperePmlUpdater), 156
kappaY (in module perfDispPmlUpdater), 158
kappaZ (in module amperePmlUpdater), 156
kappaZ (in module perfDispPmlUpdater), 158
keepon (in module chirpWavePulse), 414
keepon (in module gaussianPulse), 420
keepon (in module halfSinePulse), 421
keepon (in module planeWavePulse), 425
keyword (in module History), 357
kind, 164, 166, 169, 259, 275
kind (in module cellSpecies), 204
kind (in module ComboEmField), 93
kind (in module Decomp), 80
kind (in module EmField), 84
kind (in module feedbackSTFunc), 418
kind (in module Field), 96
kind (in module fieldIonizationDCADK), 280
kind (in module FieldMultiUpdater), 155
kind (in module FieldUpdater), 102
kind (in module funcVelGen), 267
kind (in module Grid), 75
kind (in module GridBoundary), 81
kind (in module History), 357
kind (in module historySTFunc), 422
kind (in module InitialCondition), 99
kind (in module Input), 448
kind (in module iterativeSolver), 169
kind (in module MultiField), 95
kind (in module RxnProcess), 274
kind (in module Scalar), 172
kind (in module ScalarDepositor), 186
kind (in module scalarFunc), 426
kind (in module Species), 191
kind (in module VectorDepositor), 188
kind (in module yeeStaticEmField), 88, 91
kirchhoffSurfaceIntegral (module), 370

L

L_fwhm (in module gaussianPulse), 420
labels (in module Species), 193
laplacian() (built-in function), 581
laser (in module fieldEmitterVelGen), 264
laserPulseIntensityEnvelopeWidths (in module fieldEmitterVelGen), 264
laserPulseIntensityMagnitude (in module fieldEmitterVelGen), 264
laserPulseOrigin (in module fieldEmitterVelGen), 264
laserPulsePropagationDirection (in module fieldEmitterVelGen), 264
laserPulseStartTime (in module fieldEmitterVelGen), 264
laserPulseStopTime (in module fieldEmitterVelGen), 264
laserPulseWaveLength (in module fieldEmitterVelGen), 264
lastResortHistCalc, 389
lastResortHistCalc (in module functionOfTime), 388
launchWaveFromPort() (built-in function), 589
leakychannel (module), 423
leaveIncidentUnchanged (in module impactElastic), 328
leaveIncidentUnchanged (in module impactExcitation), 330
leaveIncidentUnchanged (in module impactIonCollisions), 331
leaveIncidentUnchanged (in module impactIonization), 334
leaveIncidentUnchanged (in module negativeIonDetachment), 336
length (in module Input), 448
lifetime, 323
lifetime (in module decayLifetime), 281
lightFrameEnvelopeUpdater (module), 124
lightFrameEnvForceUpdater (module), 125
lightSpeed (in module kirchhoffSurfaceIntegral), 370
linearApplyUpdater (module), 126
linearSolver() (built-in function), 515
linearSolveUpdater (module), 128
linIterUpdater (module), 130
linPlasDielcUpdater (module), 132
loadAfterInit, 254
loadAfterInit (in module boostLoader), 229
loadOnShift, 254
loadOnShift (in module boostLoader), 229
lowerBounds, 389
lowerBounds (in module absAndSav), 207
lowerBounds (in module absCutCell), 207
lowerBounds (in module absSavCutCell), 209
lowerBounds (in module absSaveDump), 211
lowerBounds (in module absSavStairStep), 212
lowerBounds (in module absSavTriCutCell), 210
lowerBounds (in module absStairStep), 212
lowerBounds (in module bitRevDensSrc), 222
lowerBounds (in module bitRevDensSrcVW), 223
lowerBounds (in module bLoop), 361
lowerBounds (in module diffuseBndry), 213
lowerBounds (in module fieldArray), 387
lowerBounds (in module fieldEnergy), 364
lowerBounds (in module fieldMomen), 365
lowerBounds (in module FieldUpdater), 103
lowerBounds (in module fluxPtclEmitter), 232
lowerBounds (in module gaussDensSrc), 234
lowerBounds (in module gridDenSrcVW), 225
lowerBounds (in module History-fieldPoyn), 368
lowerBounds (in module InitialCondition), 99

lowerBounds (in module integratedSurfaceFieldSquared), 369
lowerBounds (in module ParticleSink), 208
lowerBounds (in module peakSurfaceFieldMagnitude), 372
lowerBounds (in module planarPtclEmitter), 235
lowerBounds (in module randdenssrc), 221
lowerBounds (in module secondaryEmitter), 244
lowerBounds (in module specularBndry), 214
lowerBounds (in module transparentBndry), 215
lowerBounds (in module zeroWgtAbsorber), 216
lowerBoundsupperBounds (in module fieldOnLine), 368
lowerLimit (in module randGaussLimit), 407
lowerLimit0 (in module kappaVelGen), 268
lowerLimit1 (in module kappaVelGen), 268
lowerLimit2 (in module kappaVelGen), 268
lowerLimits (in module randKappa), 408
lowerSkinDepth (in module curlUpdater), 107
lowerSkinDepth (in module curlUpdaterCoordProd), 109
lowerSkinDepth (in module yeeAmpereUpdater), 152
lowerSkinDepth (in module yeeFaradayUpdater), 153
LXcatFile (module), 349

M

macroDensFunc (in module bitRevDensSrcVW), 224
macroDensFunc (in module randdenssrc), 220
macroFluxFunc (in module planarPtclEmitter), 236
macroPerDir (in module gridLoader), 230
macroPerDir (in module gridPosGen), 258
magField, 86
MALdampB_afterFaraday() (built-in function), 556
MALdampB_beforeFaraday() (built-in function), 554
MALdampE_afterAmpere() (built-in function), 556
MALdampE_beforeAmpere() (built-in function), 555
MALprofileDump() (built-in function), 553
manualSrc (module), 227
mask (in module cutCellPosGen), 256
mass (in module Fluid), 190
mass (in module Species), 192
massNumbers (in module linPlasDielcUpdater), 132
material (in module secElec), 238
material (in module userDefinedSecElec), 249
materialDensity (in module nullBgAbsorber), 343
materialMass (in module nullBgAbsorber), 341
materialName (in module nullBgAbsorber), 343
MatrixFiller (in module epetraUpdater), 117
matrixfiller (in module linearApplyUpdater), 127
maxApplyTime (in module InitialCondition), 99
maxcellxing (in module globalvariables), 72
maxcellxing (in module Species), 193
maxCellXings (in module Grid), 75
maximumTag (in module speciesTrackTag), 381
maxIntDepHalfWidth (in module Grid), 75
maxIter (in module yeeStaticEmField), 91
maxIterations (in module iterativeSolver), 170
maxIters (in module epetraUpdater), 115
maxIters (in module poissonUpdater), 139
maxNumEvals, 433
maxNumPoints (in module distributedRandom), 435
maxNumPoints (in module randOAFunc), 409
maxParabRadius (in module leakychannel), 423
maxRadius (in module leakychannel), 423
maxRelErr (in module distributedRandom), 435
maxRelErr (in module randOAFunc), 409
maxvalue (in module stretcher), 409
maxwellian (module), 430
mean, 405
mean (in module randGamma), 407
mean (in module randGauss), 406
mean (in module randGaussLimit), 406
measurePoint (in module pseudoPotential), 372
messageFields, 179
mindensity (in module neutralBoltzmannUpdater), 135
minDim (in module absAndSav), 207
minDim (in module absCutCell), 207
minDim (in module absSavCutCell), 209
minDim (in module absSaveDump), 211
minDim (in module absSavStairStep), 212
minDim (in module absSavTriCutCell), 210
minDim (in module absStairStep), 212
minDim (in module amperePmlUpdater), 155
minDim (in module CoordProdSTFuncStencilElement), 163
minDim (in module coordProdSTFuncStencilFiller), 162
minDim (in module diffuseBndry), 213
minDim (in module fieldVectorReader), 164
minDim (in module fieldVectorWriter), 166
minDim (in module linearApplyUpdater), 126
minDim (in module nodeFieldVectorReader), 165
minDim (in module nodeFieldVectorWriter), 167
minDim (in module nodeStencilFiller), 161
minDim (in module ParticleSink), 208
minDim (in module perfDispPmlUpdater), 157
minDim (in module phaseShiftVecUpdater), 138
minDim (in module potentialUpdater), 140
minDim (in module specularBndry), 214
minDim (in module StencilElement), 105
minDim (in module stencilFiller), 161
minDim (in module stFuncNodeVectorWriter), 168
minDim (in module STFuncStencilElement), 163
minDim (in module stFuncStencilFiller), 162
minDim (in module STFuncUpdater), 144
minDim (in module stFuncVectorWriter), 168
minDim (in module switchSpeciesSrc), 233
minDim (in module transparentBndry), 215
minDim (in module zeroWgtAbsorber), 216
minvalue (in module stretcher), 409
minWeight (in module nullSelfSplit), 340

mion (in module neutralBoltzmannUpdater), 135
MirrorAfterFaradaySetCurrentDensToZero() (built-in function), 559
MirrorAfterFaradaySetMagFieldToZero() (built-in function), 559
mlThreshold (in module epetraUpdater), 116
mode (in module Species), 193
mTwister (module), 405
mu (in module fieldEmitterVelGen), 264
multFunc (module), 424
MultiField (module), 94
Multifield-divUpdater (module), 111
Multifield-malUpdater (module), 133
MultiFiled-deyMittraConstrainUpdater (module), 110
multipleScattering (in module nullBgAbsorber), 342
multiplier (in module fieldEmitterVelGen), 262

N

n0 (in module neutralBoltzmannUpdater), 135
NAFunc, 422
name (in module stPyFunc), 427
ncnEpetraUpdaters() (built-in function), 560
ncnMatrix() (built-in function), 560
ndimZeroVector (in module gridBoundaryFunc), 444
needsJ (in module ComboEmField), 93
needsJ (in module emMultiField), 87
needsJ (in module yeeStaticEmField), 88
needsRho (in module ComboEmField), 93
needsRho (in module emMultiField), 87
negativeIonDetachment (in module Fluid-neutralGas), 347
negativeIonDetachment (module), 287, 335
neumannBC() (built-in function), 579
neutralBoltzmannUpdater (module), 134
neutralGas (in module electronAttachment), 338
neutralGas (in module impactElastic), 327
neutralGas (in module impactExcitation), 329
neutralGas (in module impactIonCollisions), 331
neutralGas (in module impactIonization), 334
neutralGas (in module negativeIonDetachment), 336
neutralGasTemp (in module impactElastic), 327
neutralGasTemp (in module impactExcitation), 329
neutralGasTemp (in module impactIonCollisions), 331
neutralGasTemp (in module impactIonization), 334
neutralGasTemp (in module negativeIonDetachment), 336
nLevels (in module yeeStaticEmField), 92
noCombThresholdKE (in module nullSelfCombination), 340
nodeFieldVectorReader (module), 165
nodeFieldVectorWriter (module), 167
nodeFuncLaplacian() (built-in function), 512
nodeInteriorness (in module gridBoundaryFunc), 443
nodeLaplacian() (built-in function), 511
nodePos (in module gridFunc), 442
nodeStencilFiller (module), 161
nodeToEdge (in module cylEdgeToNodeVec), 110
nodeToEdge (in module edgeToNodeVec), 114
nodeToFace (in module faceToNodeVec), 118
nominalDensity (in module Species), 192
nomMacroPtclsPerCellStep (in module bitRevSlabPosGen), 255
nomMacroPtclsPerStep, 254
nomMacroPtclsPerStep (in module cutCellPosGen), 256
nomMacroPtclsPerStep (in module fluxPtclEmitter), 232
nomMacroPtclsPerStep (in module gaussDensSrc), 234
noMove, 199
noMoveVW, 199
nomPtclsPerCell (in module Species), 192
nonRelES, 199
nonRelESEffMassExtd, 199
nonRelESVW, 200
nonUniformMacroPtclsFlag (in module planarPtclEmitter), 236
normal (in module fieldOnSemiCircle), 366
normalDir (in module open), 136
nspecies (in module linPlasDielcUpdater), 132
nsteps (in module globalvariables), 71
nullBgAbsorber (in module Fluid-neutralGas), 347
nullBgAbsorber (module), 341
nullFieldIonization (in module Fluid-neutralGas), 347
nullFieldIonization (module), 344
nullSelfCombination (module), 339
nullSelfSplit (module), 340
numberOfStepsForErrorAverage, 384
numberSequence, 405
numberSequence (in module distributedRandom), 435
numberSequence (in module randGamma), 407
numberSequence (in module randGauss), 406
numberSequence (in module randGaussLimit), 407
numberSequence (in module randKappa), 408
numberSequence (in module randOAFunc), 409
numBins, 592
numComponents (in module Field), 97
numComponents (in module fieldOnMovingPlane), 367
numComponents (in module fieldOnSemiCircle), 366
numMacroPPCRxns (in module Fluid), 191
numMacroPtclsPerCellPerStep (in module planarPtclEmitter), 235
numMacroPtclsPerStep (in module planarPtclEmitter), 235
numOrDenom (in module Multifield-malUpdater), 133
numPerDir (in module gridDenSrcVW), 225
numPhi (in module kirchhoffSurfaceIntegral), 370
numPhysPtclsPerStep (in module bitRevSlabPosGen), 255
numPSBins (in module nullSelfCombination), 340
numPtcls (in module gaussSource), 228

numPtclsInMacro (in module Species), 192
numPts (in module fieldOnSemiCircle), 366
numSurfacePoint (in module kirchhoffSurfaceIntegral), 370
numSurfaces (in module specularBndry), 214
numTheta (in module kirchhoffSurfaceIntegral), 370

O

OAFunc, 310, 312
OAFunc (in module binaryDissociation), 320
OAFunc (in module binaryElastic), 318
OAFunc (in module binaryExcitation), 317
OAFunc (in module binaryIonization), 315
OAFunc (in module binaryRecombination), 314
OAFunc (in module chargeExchange), 322
OAFunc (in module electronAttachment), 338
OAFunc (in module fieldIonization), 324
OAFunc (in module impactElastic), 328
OAFunc (in module impactExcitation), 330
OAFunc (in module impactIonCollisions), 332
OAFunc (in module impactIonization), 335
OAFunc (in module negativeIonDetachment), 337
OAFunc (in module nullBgAbsorber), 342
OAFunc (in module nullFieldIonization), 345
OAFunc (in module threeBodyRecombination), 326
OAFunc-expression (module), 411
offset (in module Field), 97
omega (in module fieldEmitterVelGen), 264
omega (in module gaussianPulse), 419
omega (in module halfSinePulse), 420
omega (in module lightFrameEnvelopeUpdater), 124
omega (in module planeWavePulse), 424
omega (in module sinePlaneWave), 426
open (module), 135
operation (in module binaryOperation), 383
operation (in module geometryUpdater), 120
operation (in module linIterUpdater), 131
operation (in module STFuncUpdater), 144
operation (in module unaryFieldOpUpdater), 145
operation (in module unaryOperation), 383
operation (in module unaryScalarOpUpdater), 176
optimize (in module UserFunc-expression), 437
Option (in module LXcatFile), 350
origin (in module chirpWavePulse), 414
origin (in module gaussian), 418
origin (in module gaussianPulse), 419
origin (in module halfSinePulse), 420
origin (in module planeWavePulse), 424
outerBoundsValue (in module LXcatFile), 351
outIons (in module chargeExchange), 321
outNeutrals (in module chargeExchange), 321
outNeutrals (in module threeBodyRecombination), 325
output (in module epetraUpdater), 116
output (in module poissonUpdater), 139
outsidePermittivity (in module permittivityUpdater), 137
outsidePermittivityDiag (in module permittivityUpdater), 137
outsidePermittivityOffDiag (in module permittivityUpdater), 137
outSpec1Mass (in module binaryDissociation), 319
outSpec2Mass (in module binaryDissociation), 319
outSpecies (in module binaryIonization), 315
outSpecies (in module binaryRecombination), 313
outSpecies1 (in module binaryDissociation), 319
outSpecies2 (in module binaryDissociation), 319
outSpeciesA, 311
outSpeciesB, 311
outSpeciesEnergyRatio (in module binaryDissociation), 320
overlap (in module Field), 97
overwriteTag (in module Species), 192

P

parabolicAntenna() (built-in function), 501
ParticleSource (module), 217
PartilceSink (module), 206
peakSurfaceFieldMagnitude (module), 371
perfDispCoeffs() (built-in function), 562
perfDispDx() (built-in function), 563
perfDispDy() (built-in function), 563
perfDispPMLs() (built-in function), 564
perfDispPmlUpdater (module), 157
perfDispPoissonSolver() (built-in function), 563
perfectDispersionFaradayUpdaters() (built-in function), 562
performLowPassFilter.py command line option
    -H, –historyName, (string, required), 625
    -O <outname>, –outputFileName=<outname>, (string, optional), 625
    -c <comp>, –component=<comp>, (int, optional), 625
    -f, –frequency, (float, required), 625
    -s <simname>, –simulationName=<simname>, (string, required), 625
    -w, –overwrite, (flag), 625
periodicDirs (in module Decomp), 80
permittivity, 186
permittivityField, 134
permittivityUpdater (module), 136
phase (in module chirpWavePulse), 414
phase (in module gaussianPulse), 419
phase (in module halfSinePulse), 420
phase (in module phaseShiftVecUpdater), 138
phase (in module planeWavePulse), 424
phase (in module sinePlaneWave), 426
phases (in module InitialCondition), 99
phaseShiftVecUpdater (module), 137
phaseVelocity (in module InitialCondition), 100

phi0 (in module neutralBoltzmannUpdater), 135
photonEnergy (in module fieldEmitterVelGen), 264
planarPtclEmitter (module), 235
planeVelocity (in module speciesDataOnPlane), 382
planeWavePulse (module), 424
PmlRegion (module), 180
point (in module fieldAtIndices), 365
poissonUpdater (module), 138
polarizationFlag (in module fieldIonization), 324
polarizationFlag (in module nullFieldIonization), 345
polation (in module fieldFunc), 441
position (in module fieldAtCoords), 362
position (in module kirchhoffSurfaceIntegral), 370
positionFunction (in module cutCellPosGen), 257
positionFunction (in module planarPtclEmitter), 236
PositionGenerator (in module boostLoader), 229
potentialUpdater (module), 139
power (in module Multifield-malUpdater), 133
powerLawCrossSection (module), 276
precond (in module epetraUpdater), 116
precond (in module yeeStaticEmField), 91
Preconditioner (in module iterativeSolver), 170
prime, 404
prime (in module mTwister), 405
printSampleValues (in module distributedRandom), 435
printSampleValues (in module randOAFunc), 409
probDist (in module distributedRandom), 435
probDist (in module randOAFunc), 408
problemType (in module epetraUpdater), 115
Process (in module LXcatFile), 350
Products (in module RxnProcess), 274
productSpecies, 323
pruneConsts (in module UserFunc-expression), 437
pruneCSEs (in module UserFunc-expression), 437
pruneIfs (in module UserFunc-expression), 437
pruneSelects (in module UserFunc-expression), 437
pseudoPotential (module), 372
ptclAbsorber (in module secElec), 238
ptclAbsorber (in module secFieldEmitter), 240
ptclAbsorber (in module secondaryEmitter), 243
ptclAbsorber (in module sputter), 248
ptclAbsorber (in module switchSpeciesSrc), 233
ptclAbsorber (in module userDefinedSecElec), 249
ptclAbsorber (in module userFuncSecElec), 250
ptclAbsorbers (in module speciesAbsPtclData), 374
ptclAbsorbers (in module speciesAbsPtclData2), 391
ptclAbsorbers (in module speciesBinning), 393
ptclAbsorbers (in module speciesCurrAbs), 376
ptclAttribute (in module speciesAbsPtclData), 374
ptclAttributes (in module speciesAbsPtclData2), 392
ptclAttributes (in module speciesBinning), 394
ptclCountType (in module secElec), 239
ptclCountType (in module secondaryEmitter), 244
ptclCountType (in module userFuncSecElec), 251
ptclSource (in module secondaryEmitter), 243
ptclSource (in module speciesCurrEmit), 376
ptclSources (in module speciesAbsPtclData2), 391
ptclSources (in module speciesBinning), 393
ptclSourceSpecies (in module speciesAbsPtclData2), 391
ptclSourceSpecies (in module speciesBinning), 393
ptclsPerCell, 254
ptclsPerCell (in module bitRevSlabPosGen), 255
ptclsPerCell (in module cutCellPosGen), 257
ptclSubGroupCapacity (in module cellSpecies), 205
pusher (in module cellSpecies), 204
putFieldOnSurfaceMesh.py command line option
    -O <outname>, –outputFieldName=<outname>, (string, optional), 626
    -b <bd>, –beginDump=<bd>, (int, optional), 626
    -e <ed>, –endDump=<ed>, (int, optional), 626
    -f <fldname>, –fieldName=<fldname>, (string, required), 626
    -g <geometry>, –geometryName=<geometry>, (string, required), 626
    -s <simname>, –simulationName=<simname>, (string, required), 626

Q

q0 (in module neutralBoltzmannUpdater), 135
quadCoef (in module leakychannel), 423
queryComponent (in module geometryUpdater), 120
queryGridBoundary (in module geometryUpdater), 120

R

radialCosChannel (module), 425
radiationLength (in module nullBgAbsorber), 342
radius (in module fieldOnSemiCircle), 366
radiusFarSphere (in module kirchhoffSurfaceIntegral), 370
radiusSphere (in module kirchhoffSurfaceIntegral), 370
randdenssrc (module), 220
randGamma (module), 407
randGauss (module), 406
randGaussLimit (module), 406
randKappa (module), 408
randOAFunc (module), 408
randomSeed, 260
randomSeed (in module binaryElastic), 282
randomSeed (in module binaryExcitation), 284
randomSeed (in module binaryReaction), 286
randomSeed (in module binaryRecombination), 292
randomSeed (in module chargeExchange), 285
randomSeed (in module decay), 291
randomSeed (in module dissociativeIonization), 290
randomSeed (in module dissociativeRecombination), 293
randomSeed (in module electonIonization), 289
randomSeed (in module electronAttachment), 286
randomSeed (in module electronImpactDissociation), 294

randomSeed (in module electronScatter), 283
randomSeed (in module fieldIonization), 291
randomSeed (in module globalvariables), 72
randomSeed (in module impactIonization), 288
randomSeed (in module negativeIonDetachment), 287
randomSeed (in module RxnProcess), 274
randomSeed (in module threeBodyRecombination), 293
randomSeeding (in module globalvariables), 72
ratioSH (in module Fluid), 190
Reactants (in module RxnProcess), 274
readAndCompute, 591
readComponent (in module fieldVectorWriter), 167
readComponent (in module nodeFieldVectorWriter), 167
readComponent (in module potentialUpdater), 140
readComponentBounds (in module fieldVectorWriter), 167
readComponentBounds (in module nodeFieldVectorWriter), 167
readComponents (in module epetraUpdater), 115
readComponents (in module field2ScalarUpdater), 173
readComponents (in module fieldBinOpUpdater), 119
readComponents (in module linearApplyUpdater), 126
readComponents (in module linearSolveUpdater), 128
readComponents (in module linIterUpdater), 131
readComponents (in module scalarFieldBinOpUpdater), 175
readComponents (in module unaryFieldOpUpdater), 145
readComponents (in module userFuncUpdater), 147
readFactors (in module epetraUpdater), 115
readField (in module field2ScalarUpdater), 172
readField (in module potentialUpdater), 140
readField (in module scalarFieldBinOpUpdater), 174
readFieldCompShifts (in module curlUpdater), 106
readFieldCompShifts (in module curlUpdaterCoordProd), 108
readFieldCompShifts (in module yeeAmpereUpdater), 151
readFieldFactors (in module curlUpdater), 106
readFieldFactors (in module curlUpdaterCoordProd), 108
readFieldPosVariables (in module userFuncUpdater), 147
readFields (in module amperePmlUpdater), 155
readFields (in module curlUpdater), 106
readFields (in module curlUpdaterCoordProd), 108
readFields (in module cylEdgeToNodeVec), 110
readFields (in module deyMittraUpdater), 111
readFields (in module divUpdaterCoordProd), 113
readFields (in module edgeToNodeVec), 114
readFields (in module epetraUpdater), 115
readFields (in module faceToNodeVec), 118
readFields (in module fieldBinOpUpdater), 118
readFields (in module fieldSqrDiagUpdater), 157
readFields (in module fieldVectorWriter), 166
readFields (in module gradVecUpdater), 121
readFields (in module gradVecUpdaterCoordProd), 122
readFields (in module lightFrameEnvelopeUpdater), 124
readFields (in module lightFrameEnvForceUpdater), 126
readFields (in module linearApplyUpdater), 126
readFields (in module linearSolveUpdater), 128
readFields (in module linIterUpdater), 131
readFields (in module linPlasDielcUpdater), 132
readFields (in module Multifield-divUpdater), 112
readFields (in module MultiFiled-deyMittraConstrainUpdater), 110
readFields (in module neutralBoltzmannUpdater), 135
readFields (in module nodeFieldVectorWriter), 167
readFields (in module open), 136
readFields (in module perfDispPmlUpdater), 157
readFields (in module poissonUpdater), 138
readFields (in module smooth1D), 142
readFields (in module unaryFieldOpUpdater), 145
readFields (in module userFuncUpdater), 147
readFields (in module yeeAmpereDielVecUpdater), 150
readFields (in module yeeAmpereUpdater), 151
readFields (in module yeeConductorUpdater), 159
readFields (in module yeeFaradayUpdater), 153
readFieldVarNames (in module userFuncUpdater), 147
readItersShiftInX (in module userFuncUpdater), 147
readItersShiftInY (in module userFuncUpdater), 147
readItersShiftInZ (in module userFuncUpdater), 147
readScalars (in module scalarBinOpUpdater), 173
readScalars (in module scalarFieldBinOpUpdater), 174
readScalars (in module unaryScalarOpUpdater), 176
readScalars (in module userFuncScalarUpdater), 177
readScalars (in module userFuncUpdater), 147
recombinationEnergy (in module threeBodyRecombination), 293
recordParticleData (in module absSavTriCutCell), 210
recordParticleData (in module ParticleSource), 218
reduceMethod, 389
reduceMethod (in module field2ScalarUpdater), 173
reduction (in module childLangmuirVelGen), 261
referencePoint (in module pseudoPotential), 372
reflectingBox() (built-in function), 580
reflectingBoxCylindrical() (built-in function), 580
reflectivity (in module fieldEmitterVelGen), 265
reflectPtcls (in module transparentBndry), 215
refPtclCharge (in module nullBgAbsorber), 342
refPtclMass (in module nullBgAbsorber), 342
region (in module fieldAverage), 363
region (in module permittivityUpdater), 137
Region (in module PmlRegion), 181
region (in module stRgnMask), 429
relativeValueFlag (in module fluxPtclEmitter), 232
relBoris, 200
relBorisBallisticVW, 200
relBorisCyl, 201
relBorisCylVW, 201
relBorisCylVWTagged, 201

relBorisDF, 201
relBorisEffMassExtd, 201
relBorisFuncVW, 202
relBorisTagged, 203
relBorisVW, 203
relBorisVWScale, 203
relBorisVWTagged, 203
relPermForm (in module PmlRegion), 182
removeFluxFilesAfterReadFlag (in module fluxPtclEmitter), 232
removeStoppedParticles (in module nullBgAbsorber), 343
restoreTimeFromField (in module MultiField), 96
result (in module fieldFunc), 440
result (in module gridBoundaryFunc), 443
result (in module gridFunc), 442
resultVecDescriptions, 390
resultVecDescriptions (in module functionOfTime), 388
rhoJFieldName (in module PmlRegion), 181
rhoJmagFieldName (in module PmlRegion), 181
rhojweighting (in module EmField), 84
rowFieldIndex (in module CoordProdSTFuncStencilElement), 163
rowFieldIndex (in module StencilElement), 105
rowFieldIndex (in module STFuncStencilElement), 163
rowInteriorosity (in module nodeStencilFiller), 161
runName, 592
rxnPhysics (in module RxnProcess), 274
RxnProcess (module), 274
RxnRate-expPolyCrossSection (module), 277

S

samplesPerCell (in module setEpsilonUpdater), 141
saveEmField() (built-in function), 584
saveHistories() (built-in function), 534
scalar (in module scalarFunc), 426
Scalar (module), 171
scalarBinOpUpdater (module), 173
ScalarDepositor (module), 186
scalarFieldBinOpUpdater (module), 174
scalarFunc (module), 426
scalars, 386
scaling (in module epetraUpdater), 116
scaling (in module fieldVectorReader), 165
scaling (in module fieldVectorWriter), 167
scaling (in module nodeFieldVectorReader), 165
scaling (in module nodeFieldVectorWriter), 167
scaling (in module poissonUpdater), 139
scaling (in module stFuncNodeVectorWriter), 168
scaling (in module stFuncVectorWriter), 168
scatterFlag, 200, 202
scattering (in module fieldEmitterVelGen), 265
scatteringBox() (built-in function), 570
scatteringBox1Dielectric() (built-in function), 570
scatteringBox2Dielectric() (built-in function), 570
scatterType (in module electronScatter), 283
secElec (module), 237
secFieldEmitter (module), 240
secondaryEmitter (module), 243
secondarySpecies (in module secElec), 239
secondarySpecies (in module secFieldEmitter), 240
secondarySpecies (in module userDefinedSecElec), 249
secondarySpecies (in module userFuncSecElec), 251
sectionBreaks, 77
secVSig (in module sputter), 248
seed, 260, 405
seed (in module distributedRandom), 435
seed (in module fluxPtclEmitter), 232
seed (in module gaussDensSrc), 234
seed (in module kappaVelGen), 268
seed (in module planarPtclEmitter), 236
seed (in module randGamma), 407
seed (in module randGauss), 406
seed (in module randGaussLimit), 407
seed (in module randKappa), 408
seed (in module randOAFunc), 408
seed (in module stretcher), 409
seed (in module sysRandom), 410
setAnalyticBackground_DDLD() (built-in function), 493
setDebye_DDLD() (built-in function), 495
setDrudeByCarrier_DDLD() (built-in function), 494
setDrudeByConductance_DDLD() (built-in function), 494
setEpsilonUpdater (module), 141
setGridData() (built-in function), 472, 473
setInfLimits_DDLD() (built-in function), 493
setMinMaxFromFile, 411
setPointsRandom, 591
setPointsUniform, 591
setTimingData() (built-in function), 461
sey (in module secondaryEmitter), 244
shiftPtclPosition, 227
shiftUpdaters (in module MultiField), 96
shortCircuitUnneededArgs (in module UserFuncexpression), 437
sigma (in module randGamma), 407
sigma (in module randGauss), 406
sigma (in module randGaussLimit), 406
sigma (in module yeeConductorUpdater), 159
sigmaFactorW (in module PmlRegion), 181
sigmaForm (in module PmlRegion), 181
sigmaFormW (in module PmlRegion), 181
sigmas (in module randKappa), 408
sigmaX (in module amperePmlUpdater), 155
sigmaX (in module perfDispPmlUpdater), 157
sigmaX (in module yeeConductorUpdater), 159
sigmaY (in module amperePmlUpdater), 155
sigmaY (in module perfDispPmlUpdater), 158

sigmaY (in module yeeConductorUpdater), 159
sigmaZ (in module amperePmlUpdater), 155
sigmaZ (in module perfDispPmlUpdater), 158
sigmaZ (in module yeeConductorUpdater), 159
SIM_DATA_PATH, 597
sinePlaneWave (module), 426
singleField (in module fieldEnergy), 364
skewness (in module chirpWavePulse), 414
skewness (in module halfSinePulse), 421
skewness (in module planeWavePulse), 425
skipFirst (in module divUpdaterCoordProd), 113
skipFirst (in module Multifield-divUpdater), 112
Slab (in module gridLoader), 230
smooth() (built-in function), 577
smooth1D (module), 142
smoothComp (in module smooth1D), 143
smoothDir (in module smooth1D), 143
smoother (in module yeeStaticEmField), 91
smoothyz() (built-in function), 577
solver (in module epetraUpdater), 115
Solver (in module lightFrameEnvelopeUpdater), 124
solver (in module poissonUpdater), 139
solverType, 171
sortPtcls (in module globalvariables), 72
Source (in module Field), 98
sourceLowerBounds (in module InitialCondition), 99
sourceLowerBounds (in module potentialUpdater), 140
sourceSpecies (in module secondaryEmitter), 244
sourceType (in module speciesCurrEmit), 376
sourceUpperBounds (in module InitialCondition), 99
sourceUpperBounds (in module potentialUpdater), 140
species, 592
species (in module History), 357
species (in module nullBgAbsorber), 341
species (in module nullSelfCombination), 339
species (in module nullSelfSplit), 340
species (in module speciesAbsPtclData), 374
species (in module speciesAbsPtclData2), 391
species (in module speciesBinning), 393
species (in module speciesCurrAbs), 376
species (in module speciesCurrEmit), 376
species (in module speciesDataOnPlane), 382
species (in module speciesEnergy), 377
species (in module speciesMomen), 378
species (in module speciesNumberOf), 379
species (in module speciesNumPhysical), 379
species (in module speciesRmsDistToAxis), 379
species (in module speciesRmsMomen), 380
species (in module speciesRmsPosition), 380
species (in module speciesTrackTag), 381
Species (module), 191
speciesAbsPtclData (module), 373
speciesAbsPtclData2 (module), 391
speciesAtomicNumber (in module nullBgAbsorber), 342
speciesBinning (module), 392
speciesCurrAbs (module), 375
speciesCurrEmit (module), 376
speciesDataOnPlane (module), 382
speciesEnergy (module), 377
speciesMomen (module), 378
speciesName (in module binaryDissociation), 319
speciesName (in module binaryIonization), 315
speciesNumberOf (module), 378
speciesNumPhysical (module), 379
speciesRmsDistToAxis (module), 379
speciesRmsMomen (module), 380
speciesRmsPosition (module), 380
speciesTrackTag (module), 381
specularBndry (module), 213
speed (in module fieldOnMovingPlane), 367
splitInterval (in module nullSelfSplit), 341
sputter (module), 247
sputterAtomSpecies (in module sputter), 248
sputterAtomType (in module sputter), 248
standardTimeStep() (built-in function), 570
startAmplitude (in module cosineFlattop), 415
startAmplitude (in module cosineRamp), 416
startAmplitude (in module radialCosChannel), 426
startFlattop (in module cosineFlattop), 415
startPosition (in module acceleratingVoltage), 360
startPosition (in module cosineFlattop), 415
startPosition (in module cosineRamp), 416
startRadius (in module radialCosChannel), 426
stencil (in module cellSpecies), 205
StencilElement (in module nodeStencilFiller), 161
StencilElement (in module stencilFiller), 161
StencilElement (module), 104
StencilElementMacro() (built-in function), 477, 557
stencilFiller (module), 161
stepPrintPeriodicity (in module globalvariables), 73
STFunc (in module cacheFunc), 413
STFunc (in module childLangmuirVelGen), 261
STFunc (in module coordProdSTFuncStencilFiller), 162
STFunc (in module funcVelGen), 267
STFunc (in module InitialCondition), 99
STFunc (in module inverseFunc), 423
STFunc (in module multFunc), 424
STFunc (in module open), 136
STFunc (in module stFuncNodeVectorWriter), 168
STFunc (in module STFuncUpdater), 144
STFunc (in module stFuncVectorWriter), 168
STFunc (in module sumFunc), 427
STFunc (in module unaryScalarOpUpdater), 176
STFunc (module), 412
STFunc-expression (module), 416
stFuncLaplacian() (built-in function), 582
stFuncNodeVectorWriter (module), 167
STFuncStencilElement (module), 163

STFuncStencilElementMacro() (built-in function), 478, 558
stFuncStencilFiller (module), 162
STFuncUpdater (module), 143
stFuncVectorWriter (module), 168
stoppingPower (in module nullBgAbsorber), 341
stPyFunc (module), 427
straggling (in module nullBgAbsorber), 342
stretcher (module), 409
stretchFunc (in module stretcher), 409
stRgnMask (module), 429
subselectparticles.py command line option
  -S <spname>, –speciesName=<spname>, (string, required), 627
  -c <comp>, –component=<comp>, (int, optional), 627
  -f <fl>, –flipLimits=<fl>, (int, required), 627
  -l <lb>, –lowerBounds=<lb>, (float, optional), 627
  -o <outspname>, –outputSpeciesName=<outspname>, (string, optional), 627
  -s <simname>, –simulationName=<simname>, (string, required), 627
  -u <ub>, –upperBounds=<ub>, (float, optional), 627
  -w, –overwrite, (flag), 627
subtractYeeFaraday (in module deyMittraUpdater), 111
sumFunc (module), 427
suppressEnergy (in module secElec), 239
suppressEnergy (in module secondaryEmitter), 245
suppressEnergy (in module userFuncSecElec), 251
surface (in module diffuseBndry), 213
surface (in module specularBndry), 214
surface (in module transparentBndry), 215
surfaceArea (in module gridBoundaryFunc), 444
surfaceCenter (in module gridBoundaryFunc), 443
surfaceCentroid (in module gridBoundaryFunc), 443
surfaceCharge() (built-in function), 514
surfaceOutwardArea (in module gridBoundaryFunc), 444
surfaceOutwardArea3D (in module gridBoundaryFunc), 444
surfaceOutwardUnitNormal (in module gridBoundaryFunc), 444
surfaceOutwardUnitNormal3D (in module gridBoundaryFunc), 444
surfacePotential() (built-in function), 514
surfNormalX (in module setEpsilonUpdater), 142
surfNormalY (in module setEpsilonUpdater), 142
surfNormalZ (in module setEpsilonUpdater), 142
susceptibilityDep, 198
sweepRate, 254
sweepVel (in module gridLoader), 230
switchSpecies (in module switchSpeciesSrc), 233
switchSpeciesSrc (module), 233
sysRandom (module), 409

T

T0 (in module binaryRecombination), 313
T0 (in module neutralBoltzmannUpdater), 135
T1 (in module binaryRecombination), 313
taggedParticle (in module cellSpecies), 205
tags (in module speciesTrackTag), 381
tagsFromFile (in module speciesTrackTag), 381
Tcol (in module binaryIonization), 315
temperature (in module fieldEmitterVelGen), 262, 263
tempExpFactor (in module threeBodyRecombination), 325
Term (in module UserFunc-expression), 436
thicknessFunc (in module nullBgAbsorber), 342
threeBodyRecombination (module), 292, 325
threshold (in module nullSelfCombination), 339
threshold (in module yeeStaticEmField), 92
thresholdEnergy (in module binaryElastic), 318
thresholdEnergy (in module binaryExcitation), 317
thresholdEnergy (in module binaryIonization), 315
thresholdEnergy (in module binaryReaction), 285
thresholdEnergy (in module dissociativeRecombination), 293
thresholdEnergy (in module electronAttachment), 286
thresholdEnergy (in module negativeIonDetachment), 287
thresholdFunc (in module nullSelfSplit), 340
thresoldEnergy (in module binaryDissociation), 319
timeConstant, 384
timeInterval (in module kirchhoffSurfaceIntegral), 370
timeIsIrrelevant (in module cacheFunc), 413
timeWindow, 373
timingAnalysisPeriodicity (in module globalvariables), 73
toDtFrac, 179
tolerance (in module iterativeSolver), 170
tolerance (in module yeeStaticEmField), 91
toZero (in module RxnRate-expPolyCrossSection), 278
transparency (in module transparentBndry), 215
transparentBndry (module), 215
turnOn() (built-in function), 589
twoColumnFile (module), 278
type (in module Input), 448
typeOfCollision (in module LXcatFile), 349
types (in module Input), 448

U

unaryFieldOpUpdater (module), 144
unaryOperation (module), 383
unaryScalarOpUpdater (module), 175
unbiasedSelector, 308
uniformVector (in module Input), 448
unstableSpecies, 323
untrackedPhoton (in module decay), 291
update_DDLD() (built-in function), 496

updateDebugType, 200, 202
updateFunction (in module userFuncScalarUpdater), 177
updateFunction (in module userFuncUpdater), 147
updateOrder, 273
updatePeriod, 273
updaters, 179
updateStepOrder (in module MultiField), 96
upperBounds, 389
upperBounds (in module absAndSav), 207
upperBounds (in module absCutCell), 207
upperBounds (in module absSavCutCell), 209
upperBounds (in module absSaveDump), 211
upperBounds (in module absSavStairStep), 212
upperBounds (in module absSavTriCutCell), 210
upperBounds (in module absStairStep), 212
upperBounds (in module bitRevDensSrc), 222
upperBounds (in module bitRevDensSrcVW), 224
upperBounds (in module bLoop), 361
upperBounds (in module diffuseBndry), 213
upperBounds (in module fieldArray), 387
upperBounds (in module fieldEnergy), 364
upperBounds (in module fieldMomen), 365
upperBounds (in module FieldUpdater), 103
upperBounds (in module fluxPtclEmitter), 232
upperBounds (in module gaussDensSrc), 234
upperBounds (in module gridDenSrcVW), 225
upperBounds (in module History-fieldPoyn), 368
upperBounds (in module InitialCondition), 99
upperBounds (in module integratedSurfaceFieldSquared), 369
upperBounds (in module ParticleSink), 208
upperBounds (in module peakSurfaceFieldMagnitude), 372
upperBounds (in module planarPtclEmitter), 235
upperBounds (in module randdenssrc), 221
upperBounds (in module secondaryEmitter), 244
upperBounds (in module specularBndry), 214
upperBounds (in module transparentBndry), 215
upperBounds (in module zeroWgtAbsorber), 216
upperLimit (in module randGaussLimit), 407
upperLimit0 (in module kappaVelGen), 268
upperLimit1 (in module kappaVelGen), 268
upperLimit2 (in module kappaVelGen), 268
upperLimits (in module randKappa), 408
upperOrLower (in module Multifield-malUpdater), 133
upperSkinDepth (in module curlUpdater), 107
upperSkinDepth (in module curlUpdaterCoordProd), 109
upperSkinDepth (in module yeeAmpereUpdater), 152
upperSkinDepth (in module yeeFaradayUpdater), 153
useCornerMove, 254
useCornerMove (in module absCutCell), 207
useCornerMove (in module absSavCutCell), 209
useEffectiveMasses, 200, 202
useGridBndryRestore (in module globalvariables), 73
useHistoryRestore (in module globalvariables), 73
useMemory (in module UserFunc-expression), 437
useObliqueReflection (in module diffuseBndry), 213
usePositionFuncInEmitDir (in module planarPtclEmitter), 236
userDefinedSecElec (module), 249
UserFunc (in module OAFunc-expression), 411
UserFunc (in module UserFunc-expression), 436
UserFunc-expression (module), 435
userFuncExpression (module), 428
userFuncScalarUpdater (module), 177
userFuncSecElec (module), 250
userFuncUpdater (module), 146
userFuncVelGen (module), 269
useRunNameAsPrefix (in module absSaveDump), 211
useRunNameAsPrefix (in module fluxPtclEmitter), 231
useSegmentedMove (in module Species), 193
useSequentialTags (in module Species), 193
useSmallSigmaApproximation (in module PmlRegion), 182
useThermalDist (in module negativeIonDetachment), 337
useUniPml (in module PmlRegion), 181
useVecUpdater (in module curlUpdater), 106
useVecUpdater (in module curlUpdaterCoordProd), 108
useVecUpdater (in module yeeAmpereUpdater), 151
useVecUpdater (in module yeeFaradayUpdater), 153

V

value (in module constant), 434
value (in module CoordProdSTFuncStencilElement), 164
value (in module StencilElement), 105
value (in module STFuncStencilElement), 163
variable (in module OAFunc-expression), 411
variable (in module powerLawCrossSection), 277
variable (in module RxnRate-expPolyCrossSection), 278
variableWeightParticle (in module cellSpecies), 205
VAtheory (in module decay), 291
vbar (in module beamVelocityGen), 260
vbar (in module bitRevDensSrcVW), 224
vbar (in module fluxPtclEmitter), 232
vbar (in module gaussDensSrc), 234
vbar (in module gaussSource), 228
vbar (in module gridDenSrcVW), 225
vbar (in module kappaVelGen), 268
vbar (in module planarPtclEmitter), 235
vbar (in module randdenssrc), 221
vbar (in module transparentBndry), 215
vdrift_x (in module maxwellian), 430
vdrift_y (in module maxwellian), 430
vdrift_z (in module maxwellian), 430
VectorDepositor (module), 187
vectorreader (in module linearApplyUpdater), 127
vectorwriter (in module linearApplyUpdater), 127
velocity (in module acceleratingVoltage), 360

velocity (in module gaussian), 418
velocityDim (in module maxwellian), 430
VelocityGenerator (in module boostLoader), 229
VelocityGenerator (in module gridLoader), 230
velocityGenerator (in module userFuncVelGen), 269
velocityIsLocal, 260
velocityIsLocal (in module fieldEmitterVelGen), 262
velocitySequence_0 (in module bitRevDensSrc), 222
velocitySequence_0 (in module randdenssrc), 220
velocitySequence_1 (in module bitRevDensSrc), 222
velocitySequence_2 (in module bitRevDensSrc), 222
velocitySequence_3 (in module bitRevDensSrc), 222
velOverC (in module open), 136
verbosity (in module globalvariables), 72
verbosity (in module RxnProcess), 274
vg (in module chirpWavePulse), 414
vg (in module gaussianPulse), 420
vg (in module halfSinePulse), 421
vg (in module planeWavePulse), 425
voidGeoCad() (built-in function), 531
voidGeoPython() (built-in function), 532
vsig (in module beamVelocityGen), 260
vsig (in module bitRevDensSrcVW), 224
vsig (in module diffuseBndry), 213
vsig (in module fluxPtclEmitter), 232
vsig (in module gaussDensSrc), 234
vsig (in module gaussSource), 228
vsig (in module gridDenSrcVW), 225
vsig (in module kappaVelGen), 268
vsig (in module planarPtclEmitter), 235
vsig (in module randdenssrc), 221
vsig (in module transparentBndry), 215
vthermal (in module maxwellian), 430
vthermNeutral (in module chargeExchange), 321
vthermNeutrals (in module threeBodyRecombination), 325

W

waistDisplacement (in module chirpWavePulse), 414
waistDisplacement (in module gaussianPulse), 420
waistDisplacement (in module halfSinePulse), 421
warningLevel, 390
warningLevel (in module functionOfTime), 388
warningLevel (in module speciesBinning), 394
weight_max (in module nullSelfCombination), 339
weightCurrents (in module Fluid), 190
weightFunc (in module bitRevDensSrcVW), 224
weightFunc (in module gridDenSrcVW), 225
weightFunc (in module planarPtclEmitter), 236
weightIndex, 592
weightMinimum (in module planarPtclEmitter), 236
width (in module gaussian), 418
widths (in module chirpWavePulse), 414
widths (in module gaussianPulse), 419
widths (in module halfSinePulse), 421
widths (in module planeWavePulse), 425
work_function (in module fieldEmitterVelGen), 262, 263
write1CellVolume() (built-in function), 462
writeComponent (in module fieldBinOpUpdater), 119
writeComponent (in module fieldVectorReader), 165
writeComponent (in module nodeFieldVectorReader), 165
writeComponent (in module potentialUpdater), 140
writeComponent (in module scalarFieldBinOpUpdater), 175
writeComponentBounds (in module fieldVectorReader), 165
writeComponentBounds (in module nodeFieldVectorReader), 165
writeComponents (in module epetraUpdater), 115
writeComponents (in module geometryUpdater), 120
writeComponents (in module importFromFileUpdater), 123
writeComponents (in module linearApplyUpdater), 127
writeComponents (in module linearSolveUpdater), 128
writeComponents (in module linIterUpdater), 131
writeComponents (in module STFuncUpdater), 144
writeComponents (in module unaryFieldOpUpdater), 145
writeComponents (in module userFuncUpdater), 147
writeEquationToFile (in module linearApplyUpdater), 127
writeEquationToFile (in module linearSolveUpdater), 128
writeFactors (in module epetraUpdater), 115
writeField (in module Multifield-malUpdater), 133
writeField (in module potentialUpdater), 140
writeField (in module scalarFieldBinOpUpdater), 174
writeFieldPosVariables (in module userFuncUpdater), 147
writeFields (in module amperePmlUpdater), 155
writeFields (in module curlUpdater), 106
writeFields (in module curlUpdaterCoordProd), 108
writeFields (in module cylEdgeToNodeVec), 110
writeFields (in module deyMittraUpdater), 111
writeFields (in module divUpdaterCoordProd), 113
writeFields (in module dummyUpdater), 114
writeFields (in module edgeToNodeVec), 114
writeFields (in module epetraUpdater), 115
writeFields (in module faceToNodeVec), 118
writeFields (in module fieldBinOpUpdater), 118
writeFields (in module fieldVectorReader), 165
writeFields (in module geometryUpdater), 120
writeFields (in module gradVecUpdater), 121
writeFields (in module gradVecUpdaterCoordProd), 122
writeFields (in module importFromFileUpdater), 122
writeFields (in module lightFrameEnvelopeUpdater), 124
writeFields (in module lightFrameEnvForceUpdater), 126
writeFields (in module linearApplyUpdater), 126
writeFields (in module linearSolveUpdater), 128

writeFields (in module linIterUpdater), 131
writeFields (in module linPlasDielcUpdater), 132
writeFields (in module Multifield-divUpdater), 112
writeFields (in module MultiFiled-deyMittraConstrainUpdater), 110
writeFields (in module neutralBoltzmannUpdater), 135
writeFields (in module nodeFieldVectorReader), 165
writeFields (in module open), 136
writeFields (in module perfDispPmlUpdater), 157
writeFields (in module phaseShiftVecUpdater), 138
writeFields (in module poissonUpdater), 138
writeFields (in module setEpsilonUpdater), 141
writeFields (in module smooth1D), 143
writeFields (in module STFuncUpdater), 144
writeFields (in module unaryFieldOpUpdater), 145
writeFields (in module userFuncUpdater), 147
writeFields (in module yeeAmpereDielVecUpdater), 150
writeFields (in module yeeAmpereUpdater), 151
writeFields (in module yeeConductorUpdater), 159
writeFields (in module yeeFaradayUpdater), 153
writeIndicesFromCoords() (built-in function), 462
writeMovingWindow() (built-in function), 489, 490
writePeriod (in module fieldSqrDiagUpdater), 157
writeScalars (in module field2ScalarUpdater), 173
writeScalars (in module scalarBinOpUpdater), 173
writeScalars (in module unaryScalarOpUpdater), 176
writeScalars (in module userFuncScalarUpdater), 177

X

xbar (in module gaussSource), 228
xComponents (in module speciesTrackTag), 381
xdirection (in module specularBndry), 214
Xrad (in module binaryRecombination), 313
xsig (in module gaussSource), 228
xsmth() (built-in function), 576
xsurface (in module specularBndry), 214

Y

ydirection (in module specularBndry), 214
Yee() (built-in function), 510
yeeAmpereDielVecUpdater (module), 150
yeeAmpereUpdater (module), 151
yeeConductorUpdater (module), 158
yeeFaradayUpdater (module), 152
yeeStaticEmField (module), 88
yieldDataFile (in module userDefinedSecElec), 249
ysurface (in module specularBndry), 214
yzsmth() (built-in function), 576

Z

zdirection (in module specularBndry), 214
zeroFieldValue (in module fieldFunc), 441
zeroWgtAbsorber (module), 216
zsurface (in module specularBndry), 214