GSim Customization

Release 1.0.0rc1

Tech-X Corporation

Dec 02, 2025

CONTENTS

1 Overview 1

2 Macros 3

3 Analyzers 11

CHAPTER

ONE

OVERVIEW

This guide shows how to customize GSim by adding macros and analyzers.

GSim is an arbitrary dimensional, electromagnetics and plasma simulation code consisting of two major components:

- GSimComposer, the graphical user interface.
- Vorpal [NC04], the GSim Computational Engine.

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

CHAPTER

TWO

MACROS

2.1 Basics of Macro Definition and Usage

A macro file contains one or more macros. A macro is invoked as a parameterized token with the result being that the body of the macro is placed into the input file. In this section we dicuss the basics of macros, including their definition and use.

2.1.1 Macros in Brief

In its simplest form, a macro provides a way to substitute a code snippet from an input file:

```
<macro snippet>
  line1
  line2
  line3
 </macro>
```

In this example, every occurrence of the code named snippet in the input file will now be replaced by the three lines defined between the <macro> and </macro> tags.

For example, you could define a macro to set up a laser pulse like this:

```
<macro  myLaser>
  EmField
  <BoundaryCondition LaserPulseBC>
    ... some regular boundary conditions ...
  </BoundaryCondition>
</macro>
```

You could then call your myLaser macro within the input file like this:

```
<Field  exampleField>
  kind = myLaser
</Field>
```

The GSim engine (Vorpal) will expand the input file use of your macro into:

```
<Field  exampleField>
  kind = EmField
  <BoundaryCondition LaserPulseBC>
    ... some regular boundary conditions ...
```

(continues on next page)

(continued from previous page)

```
  </BoundaryCondition>
</Field>
```

2.1.2 Macro Parameters

Macros can take parameters, allowing variables to be passed into and used by the macro. Parameters are listed in parentheses after the macro name in the macro declaration, as in this example:

```
<macro box(lx, ly, lz, ux, uy, uz)>
  lowerBounds = [lx ly lz]
  upperBounds = [ux uy uz]
</macro>
```

Once a macro is defined, it can be used by calling it and providing values or symbols for the parameters. The macro will substitute the parameter values into the body provided. Calling the example above with parameters defined like this:

```
$ NX = 10
$ NY = 20
$ NZ = 30
box(0, 0, 0, NX, NY/2, NZ)
```

will create the following code fragment in the processed input file:

```
lowerBounds = [0 0 0]
upperBounds = [10 10 30]
```

Note: The parameter substitution happened in the scope of the caller. Parameters do not have scope outside of the macro in which they are defined.

2.1.3 Macro Overloading

As with symbols, macros can be overloaded within a scope. The particular instance of a macro that is used is determined by the number of parameters provided at the time of instantiation. This enables the user to write macros with different levels of parameterization:

```
<macro circle(x0, y0, r)>
  r**2 - ((x-x0)**2 + (y-y0)**2)
</macro>
<macro circle(r)>
  circle(0, 0, r)
</macro>
```

Looking at the example above, whenever the macro circle is used with a single parameter, it creates a circle around the origin; if you use the macro with 3 parameters, you can specify the center of the circle.

The macro substitution does not occur until the macro instantiation is actually made. This means that you do not have to define the 3-parameter circle prior to defining the 1-parameter circle, even though the 1-parameter circle refers to the 3-parameter circle. It is only necessary that the first time the 1-parameter circle is instantiated that the 3-parameter circle has already been defined, otherwise you will receive an error.

2.1.4 Defining Functions Using Macros

Macros can be particularly useful for defining complex mathematical expressions, such as defining functions in STFunc blocks with `kind = expression`. However, one must be careful because of the substitution rules. For example, a macro that defines a Gaussian could be written,

```
<macro badGauss(A, x, sigma)>
  A * exp(-x**2/sigma)
</macro>
```

While this is a legitimate macro, an instantiation of the macro via:

```
badGauss(A0+5, x-3, 2*sigma)
```

will result in:

```
A0+5*exp(-x+3**3/2*sigma)
```

which is probably not the expected result. One alternative is to put parentheses around the parameters whenever they are used in the macro.

```
<macro betterGauss(A, x, sigma)>
  ((A) * exp(-(x)**2/(sigma)))
</macro>
```

This will ensure that the expressions in parameters will not cause any unexpected side effects. The downside of this approach, however, is that the macro text is hard to read due to all the parentheses. To overcome this issue, GSim provides a mechanism to automatically introduce the parentheses around arguments by using a function block

```
<function goodGauss(A, x, sigma)>
  A * exp(-x**2/sigma)
</function>
```

The previous example will produce the same output as the betterGauss macro, but without requiring the additional parentheses in the macro text.

2.1.5 More About Parameters

In the previous examples, parameters were always single tokens or simple expressions. However, the preprocessor allows you to pass parameters that span multiple lines. This can be particularly useful for writing larger macros. An example of multiple line parameter passing would be defining a general particle source. This example below shows a macro defining a general species:

```
<macro ions(name, charge, extra)>
  <Species name>
  kind = relBoris
  emField = emSum
  charge = charge extra
  <ParticleSink leftAbsorber>
    kind = absorber
    lowerBounds = [-1 -1 -1]
    upperBounds = [ 0 NY1 NZ1]
  </ParticleSink>
```

(continues on next page)

(continued from previous page)

```
  </Species>
</macro>
```

The parameter extra can be an arbitrary string such as:

```
ions(species1, 1.6e-19, "mass = 1e-28")
```

or it can be an empty string, if no additional information is needed:

```
ions(species2, 1.6e-19, "")
```

In addition, you can add entire input file blocks to this parameter. Assume we have a macro called loader, defined as follows:

```
<macro loader(ionDens)>
  <ParticleSource ptcl_loader>
    kind = randDensSrc
    lowerBounds = [ -0.05 -0.05 -0.2]
    upperBounds = [ 0.05 0.05 0.2]
    density = ionDens
    vbar = [0. 0. 0.]
    vsig = [V_ion_rms V_ion_rms V_ion_rms ]
    <STFunc macroDensFunc>
      kind = expression
      expression = H(0.1 - sqrt(x*x + y*y))
    </STFunc>
  </ParticleSource>
</macro>
```

Using this macro with the ions macro defined previously, we can now create an ion species with a source via a single line:

```
ions(species3, 1.6e-19, loader(1e18))
```

2.1.6 Importing Macros from Files

It is also possible to import a macro file that contains your own custom macros. This is useful when reusing one or more custom macros over multiple simulations. For example, physical constant definitions or commonly-used geometry setups can be stored in files that can then be reused. The macro file must have a .mac extension on it to be imported as a local macro, and it must be in either the directory of your .pre file, or its directory must be in the environment variable, `TXPP_PATH`.

To extend the example above, say the macro myLaser is in the file Lasers.mac. The input file would look like this:

```
$ import Lasers.mac

<Field  exampleField>
  kind = myLaser
</Field>
```

Vorpal will expand the input file use of your macro into:

```
<Field  exampleField>
  kind = EmField
  <BoundaryCondition LaserPulseBC>
    ... some regular boundary conditions ...
  </BoundaryCondition>
</Field>
```

The macro definition would remain the exact same. As long as the macro file is imported properly, it is just like having it defined explicitly in the input file.

Files are imported via the import keyword:

```
$ import FILENAME
```

where FILENAME represents the name of the file to be included. GSim applies the standard rules for token substitution to any tokens after the import token. Quotes around the filename are optional and computed filenames are possible.

2.1.7 Conditionals

The Vorpal preprocessor includes both flow control and conditional statements, similar to other scripting languages. These features allow the user a great deal of flexibility when creating input files.

The most general form for a conditional is

```
$ if (COND1)
  ...
$ elseif (COND2)
  ...
$ elseif (COND3)
  ...
$ else
  ...
$ endif
```

where there is a stanza starting with `$ if`, zero or more stanzas beginning with `$ elseif`, and zero or one stanza beginning with `$ else`. As in general usage, when a stanza is reached where the conditional evaluates to True, those lines are pre-processed and the resulting lines are inserted into the input file. If a conditional statement does not evaluate to True, then that branch of the conditional statement is skipped by the pre-processor. If no branches evaluate to True, then the lines after the `else` (if present) are processed. Conditionals can be arbitrarily nested. Use of parentheses around testing condition expressions is important when also using the boolean operators `not`, `and`, or `or`.

Most valid Python expressions can be inserted for the conditional test, but this is an area continuously undergoing improvement, so there may be some volatility. In particular, it is desired to have a conditional test that evalutes in Python to either True or False. As an example, for a variable that is undefined, both `$ if (undefvar)` and `$ if not (undefvar)` branches will be skipped by the pre-processor because neither evaluates to True. For the moment, other unexpected behavior can occur when checking for empty strings. The only valid way to do this for now is to use `$ if (isEqualString(s1, ""))`. The *isEqualString* should generally be used for strings to be careful, but most string comparisons, except for those involving empty strings, work in the manner expected for Python evaluation. Support for empty string comparisons using standard Python syntax will be supported in the future.

Example Conditional Statements

```
$ if (NDIM == 2)
$   dt = 1/(c * sqrt(1/dx**2+1/dy**2))
$ else
$   dt = 1/(c * sqrt(1/dx**2 + 1/dy**2 + 1/dz**2))
$ endif
```

A conditional statement can also use Boolean operators:

```
$ A = 0
$ B = 0
$ C = 1
#
# Below, D1 is 1 if A, B, or C are non-zero. Otherwise D1=0:
$ D1 = (A) or (B) or (C)
#
# Below, D2 is 1 if A is non-zero or if both B and C are non-zero.
# Otherwise D2=0:
$ D2 = (A) or ( (B) and (C) )
#
# This can be also be written as an if statement:
$ if (A) or ( (B) and (C) )
$   D3 = 1
$ else $
$   D3 = 0
$ endif
```

2.1.8 Repetition

For repeated execution, Vorpal provides while loops; these take the form:

```
$ while (COND)
  .
  .
  .
$ endwhile
```

which repeatedly inserts the loop body into the output. For example, to create 10 stacked circles using the circle macro from above, you could use:

```
$ n = 10
$ while (n > 0) circle(n)
$   n = n - 1
$ endwhile
```

2.1.9 Recursion

Macros can be called recursively. E.g. the following computes the Fibonacci numbers:

```
<macro fib(a)>
  $ if (a < 2)
    a
  $ else
    fib(a-1)+fib(a-2)
  $ endif
</macro>
fib(7)
```

Note: There is nothing preventing you from creating infinitely recursive macros; if terminal conditions are not given for the recursion, infinite loops can occur.

2.1.10 Requires

When writing reusable macros, the best practice is for macro authors to help ensure that the user can be prevented from making obvious mistakes. One such mechanism is the requires directive, which terminates translation if one or more symbols are not defined at the time. This allows users to write macros that depend on symbols that are not passed as parameters. For example, the following code snippet will not be processed if the symbol `NDIM` has not been previously defined:

```
<macro circle(r)>
  $ requires NDIM
  $ if (NDIM == 2) r**2 - x**2 - y**2
  $ endif
  $ if (NDIM == 3) r**2 - x**2 - y**2 - z**2
  $ endif
</macro>
```

2.1.11 String Concatenation

One task that is encountered often during the simulation process is naming groups of similar blocks, e.g. similar species. Macros can allow us to concatenate strings to make this process more clean and simple. However, based on the white-spacing rules, strings may be concatenated with a space between them. For example,

```
$ a = hello
$ b = world
a b
will result in
hello world
```

The space insertion is not done, however, if the last character of the first string is not a letter or a number, or if the first character of the second string is not a letter. We can avoid this rule altogether by using the `concatenate` macro:

```
concatenate(hello, world)
```

in which case, the result will always be:

```
helloworld
```

The `concatenate` macro is located in the file listUtilities.mac, and is always available at the top level for importation.

CHAPTER

THREE

ANALYZERS

3.1 Custom Analyzers

You may eventually find that you would like to analyze data from a simulation run in a way that is not possible with the analyzers that come with GSimComposer. For this, you need to write a custom analyzer. In this section we describe the process of creating custom analyzers written in Python.

An analyzer is any executable that analyzes data generated by the computational engine (Vorpal). Analyzers can be written in any language, and they can produce output just to the log window, or they can produce data files that can be visualized in the Visualization tab. If custom analyzers are designed with specific syntactical structures, then Composer can automatically integrate these analyzers into the analysis tab, providing for improved workflow management, such as adding graphical widgets corresponding to command line parameters and simplifying file reading and writing.

3.1.1 Creating Custom Python Analyzer Scripts

Given below are the key parts to enable easy compatibility with GSimComposer. These instructions are intended for use with GSim-1.0 or later. For earlier versions of GSim, please consult the appropriate documentation.

Step 1: Import the Standard Modules used for Analysis Scripts

Put the following lines at the top of your analysis script:

```
import sys
import VpAnalyzer
```

You may also choose to import additional standard or custom Python modules (like numpy or scipy) at the top level. If possible, it is recommended that other modules get imported within the main() function (see *Step 8: Provide a main() Function*) as part of a try/except block for error checking.

The VpAnalyzer module defines classes that contain functionality for reading and parsing command line options and arguments, reading and writing of valid VsHdf5-compliant files, and convenience functions related to validation and printing of command line argument inputs. Using the functions in the VpAnalyzer module supplants use of TxPyUtils and VsHdf5 directly, thus simplifying the interfaces for custom analyzers.

Step 2: Write a Class that Inherits from VpAnlayzer

Define a class in Your Custom Analyzer module that inherits from the VpAnalyer class.

```
class className(VpAnalyzer.VpAnalyzer):
  def __init__(self):
    super(className, self).__init__()
```

Deriving from the VpAnalyzer base class gives your analyzer access to all of the VpAnalyzer functionality and internally defined attributes.

Step 3: Add Description Attributes

Define two class attributes that describe the purpose of the analyzer and what the output of the analyzer is. These string attributes will be used to provide structured help and for integration with the Composer interface.

```
self.setAnalyzerDescription('This script performs ...')
self.setAnalyzerOutputDescription('This script prints out formatted text... \n
It also creates a VizSchema-compliant Hdf5 file that contains...')
```

Step 4: Adding Command Line Options and Flags

Adding command line options and flags to your analyzer will allow the user to input parameters to be used in the analysis.

Note: The following examples for Command Line Options (simulation name) and Command Line Flag (overwrite) should be included in custom analyzers.

Adding Command Line Options

The positional arguments for the VpAnalyzer.addCommandLineOption() function are

1. `short option`

   A string starting with a single dash followed by a single character.

2. `long option`

   A string starting with two dashes followed by a longer, descriptive name. Note that the long description must be a valid Python variable name, e.g. does not start with a number, is not a reserved Python word, etc.

3. `option description`

   A string describing what the command line option is; typically a few sentences.

4. `option type`

   A string declaring the primitive type of the parameter. This is either 'string' or 'float' or 'int.'

5. `default value`

   The default value for the parameter. For no default value, use "None."

6. `required or optional`

   Whether or not this parameter is necessary for the analysis to run. A Boolean: True or False.

```
self.addCommandLineOption('-s', '--simulationName', 'Name of the simulation.', 'string',␣
↪None, True)
```

The long option string is converted to a variable of the appropriate type when command line arguments are parsed by VpAnalyzer, which is available as an class attribute. So in the example above, there will be a Python variable referenced as self.simulationName in the class with a value equal to the string passed to the analyzer either as `-s` sim or as `--simulationName` sim.

Adding Command Line Flags

A command line flag is like a command line option, except the command line flag is Boolean. A command line flag can be used as a 'switch' to turn on/off certain aspects of the analysis. The positional arguments for the VpAnalyzer.addCommandLineFlag() are

1. `short flag`

   A string starting with a single dash followed by a single character.

2. `long flag`

   A string starting with two dashes followed by a longer, descriptive name.

3. `flag description`

   A string describing the flag; typically a few sentences.

```
self.addCommandLineFlag('-w', '--overwrite', 'Whether a dataset or group should be␣
↪overwritten if it already exists.')
```

The "long flag" string is converted to a Python Boolean variable (True/False), and so must be a valid Python variable, similarly to the "long option" string. The value of the variable will be True if the flag is specified on the command line either as the short or long flag, and will be False if not specified on the command line. The `--overwrite` flag should be included in your list of command line flags. This particular flag indicates to the VsHdf5 file writers whether or not datasets should be overwritten if they already exist in an output file. VsHdf5 by default will not overwrite existing datasets, so the results of your analyzer may not be written into the output file unless this flag is passed on the command line.

Step 5: Add Validation of Command Line Options

This is an opportunity to perform error checking on passed command line arguments. For instance, if an option must be non-negative, e.g. a frequency, then the value can be validated here.

```
def validateInput(self):
  if self.frequency <= 0.0:
    print('\n[moduleName] Error Command-line argument "frequency" must be greater than ␣
↪0.0.\n')
    self.printHelp()
    sys.exit(9)
```

Note that this function is a class member of the custom analyzer class that you are providing, and it's behavior supersedes the behavior of the abstract function/method VpAnalyzer.validateInput().

Step 6: Write Helper Functions

Helper functions are separate functions that will be called by analyze().

```
def SMOOTH(self, data, freq, dt):
  LENGTH = len(data)
  LENGTH_INDICES = range(LENGTH)
  PERIOD = 1/(freq*dt)
  [...] etc.
```

Step 7: Write the analyze() Function

The analyze() function is a required class function (see *Custom Analyzers Reference & Examples* for more specific details about writing an analyze() function). This function is where the actual analysis is performed. The analyze() function is called from the analyzer main() function and has access to all variables set through parsing of command line options and flags, as well as any other class functions that have been defined.

```
def analyze(self):
  [...] do analysis...
```

All VsHdf5-compatible file reading and writing that is performed in analyze() or other defined class functions should use the VpAnalyzer convenience class functions instead of direct calls to VsHdf5 functions. An instance of the VpAnalyzer base class contains an attribute that is an instance of the VsHdf5 class. The following VpAnalyzer class functions pass arguments to the owned VsHdf5 instance, thus adding an opaque interface layer to VsHdf5 that separates VsHdf5 data structures and file I/O from the VpAnalyzer class.

Step 8: Provide a main() Function

This is where execution of the analyzer is launched when invoked on the command line. The main() function should be similar in structure to the following:

```
def main():
  global os, glob
  import os, glob
  global numpy
  try:
    import numpy
  except:
    print('[className] Could not import numpy. Please make sure it is in your Python path
↪')
    print('Python path is: ')
    print(sys.path)
    sys.exit(1)

  classInstance = className()
  classInstance.parseArgs(sys.argv)
  classInstance.validateInput()
  classInstance.analyze()
  sys.exit(0)
```

The first part of main() is importing modules that are used in the analyze() or other class-defined functions. Non-system modules should be imported in a `try: except:` block to catch errors, similar to what is shown above for importing numpy. Modules imported in main() need to be declared as global prior to being imported in order to be used in class

functions. Optionally, importing modules can be done at the top-level scope of the analyzer module instead of in the main() function.

The second part of main() first instantiates an analyzer class object, parses the arguments to the analyzer that were passed on the command line, validates the command line arguments (optional), and performs the analysis.

Step 9: Make the Analyzer Executable as a Python Script

The following two-line stanza should be placed an the end of the analyzer. Also ensure that the analyzer is executable (permissions 755 are recommended on unix-type systems).

```
if (__name__ == "__main__"):
  main()
```

Finally, to make the analyzer executable from the command line, one should ensure that the first line of the file is:

```
#!/usr/bin/env python
```

3.2 Custom Analyzers Reference & Examples

This section will provide more details of what you might need to write your own analyze() function. In the *How to Import, Read, and Write a Data Group with VsHdf5* section below, several VsHdf5 class functions are listed which you will be able to use to import, read, and write data if you inherit from the VpAnalyzer class (*Write a Class that Inherits from VpAnalyzer*).

In *Examples* are examples of code you may find in the analyze() function.

3.2.1 How to Import, Read, and Write a Data Group with VsHdf5

The VpAnalyzer class uses VsHdf5 Python class functions for GSim file reading and writing. VsHdf5 ensures that metadata describing the type of datasets and other simulation information is added to simulation data so that it can be correctly visualized in Composer. GSim dumps simulation data in Hdf5 format with the extension ".h5". It is good practice to make analyzers create files with the extension ".vsh5" in order to differentiate datasets that are simulation results versus those that are analysis results. This will protect against accidentally deleting simulation results that are not recoverable unless the simulation is re-run.

There are a number of convenient functions included in VpAnalyzer that handle VizSchema file reading and writing, and these should be used in your analyze() function. There should not be any need to import VsHdf5 directly into your analysis module, nor should you need to import PyTables. There are three types of functions. First, declaring Objects returns an empty object, (either a group or dataset) with the specified name. This is useful if you are creating a new dataset or group that will be written out to a file.

```
def History(name=None)
def Field(name=None)
def Limits(name=None)
def TimeGroup(name=None)
def Mesh(name=None, kind='uniformCartesian')
def RunInfo(name=None)
def Group(name=None)
def Dataset(name=None)
```

Second are functions for reading VsHdf5 objects from an existing file. This is useful for loading in GSim dump data.

```
def getHistory(fileName=None, historyName=None, name=None, location='/')
def getField(fileName=None, fieldName=None, name=None, location='/')
def getLimits(fileName=None, name=None)
def getTimeGroup(fileName=None, name=None)
def getMesh(fileName=None, name=None)
def getRunInfo(fileName=None, name=None)
def getGroup(fileName=None, groupName=None, name=None)
def getDataset(fileName=None, datasetName=None, name=None)
```

Finally, there are functions for writing VsHdf5 objects into a new or existing file.

```
def writeHistory(fileName=None, historyName=None, data=None, meshName=None,␣
↪overwrite=False, location='/')
def writeField(fileName=None, data=None, fieldName=None, meshName=None, limitsName=None,␣
↪   timeGroupName=None, offset='nodal', dumpTime=0.0, overwrite=False)
def writeLimits(fileName=None, limitsName=None, overwrite=False)
def writeTimeGroup(fileName=None, timeGroupName=None, dumpTime=None, dumpStep=None,␣
↪overwrite=False)
def writeMesh(fileName=None, mesh=None, overwrite=False)
def writeRunInfo(fileName=None, runInfo=None, overwrite=False)
def writeGroup(fileName=None, groupName=None, overwrite=False)
def writeDataset(fileName=None, datasetName=None, overwrite=False)
```

3.2.2 Examples

Example 1: Create a 1D Data Set

Create a 1-D structured mesh and assign frequency values to that mesh's dataset. Then define a History with values for each mesh point (frequency), and write that History to a file. Note that in this example, "self.overwrite" will be True if the `--overwrite` or `-w` flag was passed as a command line argument, or False otherwise. Similarly, `--simulationName` is a command line argument.

```
fileName = self.simulationName + '_SParameters.vsh5'
mesh = self.Mesh(name='SParameters', kind='structured')
mesh.assignDataset(Freq)
self.writeMesh(fileName=fileName, mesh=mesh, overwrite=self.overwrite)

pointDataHist = self.History(name='S11')
pointDataHist.assignDataset(S11)
pointDataHist.assignAttribute('vsCentering', 'nodal')
pointDataHist.writeHistory(fileName=fileName, meshName=mesh.name, overwrite=self.
↪overwrite)
```

Example 2: Read Time Step and Dump Periodicity

Read a TimeGroup from a GSim dump file, and extract the simulation time and dump step from that group. Note that all of the get functions return both the object, and the attributes of that object.

```
timeGroup, attrs = self.getTimeGroup(fileName=speciesFileName)
time = timeGroup.getDumpTime()
step = timeGroup.getDumpStep()
```

Example 3: Extract Data from a History

Read a History from a GSim History dump file, and extract the data array from that History. Note that "hist" is a VsHdf5 object, not the actual dataset. To extract the dataset from the object, one can use the usual slicing operators, as in the example below, or by direct reference to the dataset, that is an attribute of the object: "data = hist.dataset". data is a possibly multi-dimensional numpy array.

```
fileName = self.simulationName + '_History.h5'
hist, attrs = self.getHistory(fileName=fileName, historyName=self.historyName)
data = hist[:]
```

3.3 Import your own analysis script

If you have created your own analysis script, you can import it for use in VSimComposer by clicking on the button **Import Analyzer** at the bottom left of the **Analyze** Tab. The pop-up dialog that follows will open to your home user directory.

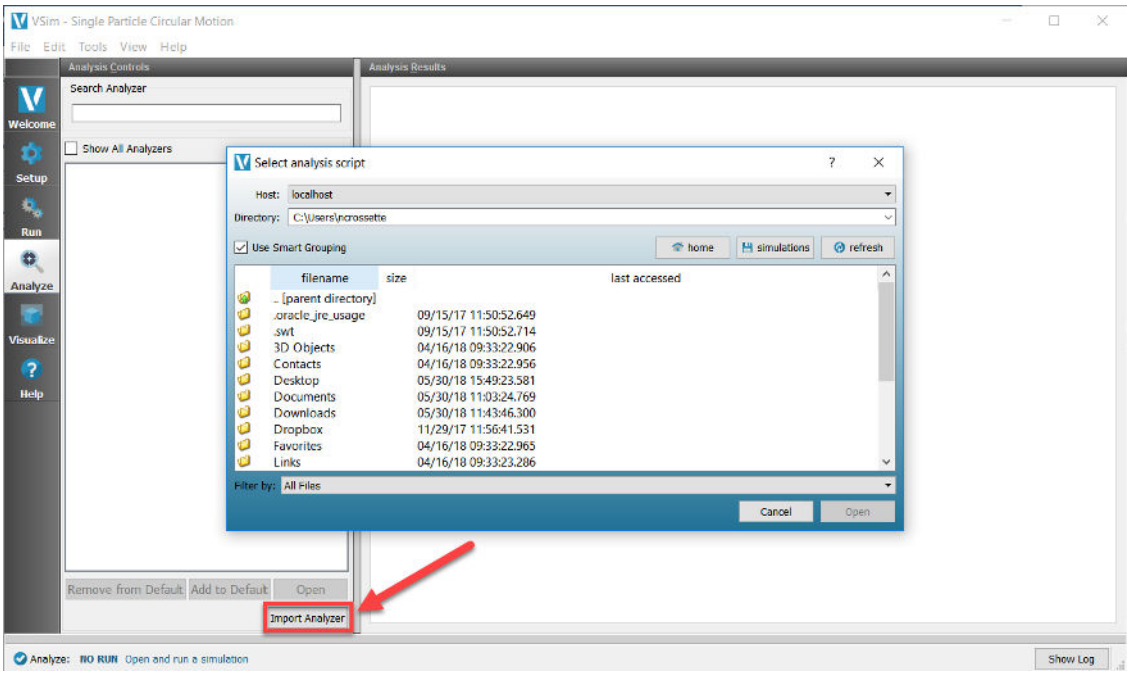

Fig. 3.1: Select a Custom Script

Navigate to the location of your analyzer, then click **Open**. On Windows, this will copy your analyzer to the *Documents/txcorp/GSim1.0/analyzers* directory if the default options were chosen during installation.

Alternatively, you can add your analyzer script to *txcorp/GSim1.0/analyzers* prior to opening GSim. If the analyzer is added to the *analyzers* directory while the GSimComposer is open, the Composer will need to be closed and re-opened for the added analyzer to appear.

On Windows, the analyzers distributed with GSim are available at:

```
C:\Program Files\Tech-X\GSim-1.0\Contents\engine\bin
```

if the default options were chosen during installation.

3.4 VsHdf5 User Guide

3.4.1 VsHdf5 Introduction

VsHdf5 is a collection of Python classes that provide an interface between VizSchema and Hdf5. These classes make it easy to read and write VizSchema-compliant files, which may be the result of simulation code dumps or analysis scripts. VizSchema-compliant files may be visualized in Composer, or other visualization software, such as VisIt.

3.4.2 Requirements

VsHdf5 has been developed using Python 2.7. It works with Python 2.6 as well, and probably Python 3.0 although that is currently not tested. VsHdf5 must be able to import pytables and numpy. When importing VsHdf5, if these modules cannot be loaded, an exception is thrown. VsHdf5 supports VizSchema version 2.1.

3.4.3 Usage

To use VsHdf5, first make sure that VsHdf5 (and numpy and pytables) are in your PYTHONPATH. Then simply use:

```
import VsHdf5
```

in your Python script.

3.4.4 Reading in VizSchema data from a file

To read an object from an existing VizSchema-compliant file in your Python script, simply create the object and pass in the name of the file and the name of the object in the file:

```
obj = VsHdf5.Object(fileName=filename, name=objectname)
```

where Object could be one of:

- Dataset
  - Field
  - History
  - Particles
- Group
  - RunInfo
  - TimeGroup

  - Limits
- Mesh
  - StructuredMesh
  - UnstructuredMesh
  - UniformCartesianMesh
  - RectilinearMesh

So for instance, to read particles from a Vorpal dump file from a species called electrons, you could write:

```
e = VsHdf5.Particles(fileName='baseName_electrons_1.h5', name='electrons')
```

This will create an object called 'e' in Python, that contains both the data in the dataset 'electrons' in the file 'baseName_electrons_1.h5', as well as the attributes associated with that data. The data can be accessed as

```
e.dataset
```

or

```
e[:]
```

both of which are a numpy array, and can be sliced and manipulated with the usual numpy operators. You can get a list of the attributes that were stored in the dataset in the file by invoking

```
attrs = e.attributeList
```

which will return a dictionary with key/value pairs representing the attributes.

For objects that do not have datasets (those that are Hdf5 groups with only attributes in the file), then dataset = None.

Optionally, you can specify a location in the Hdf5 file from which to read the data or group:

```
e = VsHdf5.Particles(fileName='baseName_electrons_1.h5', name='electrons', location = '/
↪fluids')
```

If not specified, the default location of 'root' is used ('/').

You can also read in a dataset and attributes directly after the object has been created:

```
e = VsHdf5.Particles()
array, attrs = e.readParticles('baseName_electrons_1.h5', 'electrons')
```

In this invocation, array will contain the numpy dataset, and attrs will contain the dictionary of attributes.

Certain objects often only occur once in a given file, such as Limits, Meshes, and RunInfo objects. For these types of objects, you can read from a file without specifying the name of the group or dataset that you want to read. VsHdf5 will search through the file until it finds a group or object of the correct type, and read it in for you. So you could read the (single) Limits group in a field file by invoking:

```
l=VsHdf5.Limits(fileName='gamma2D01_universe_0.h5')
```

The name of the Limits group in the file is contained in the data member

```
l.name
```

Required Attributes

Some objects require certain attributes to be VizSchema compliant. Typically, if you read an object from a simulation dump file, then all of the required attributes will be there. You can change individual attribute by using the methods

```
object.removeAttribute(attName)
object.assignAttribute(attName, attValue)
```

There are additional optional attributes for VizSchema. See the VizSchema documentation and VsHdf5.py docstrings for more details. The following attributes are required for the objects below. In some cases, there are default values.

Meshes (Hdf5 Group or Dataset):

Meshes are important types of objects for VizSchema, and deserve more attention here. Meshes may be datasets or groups in a Hdf5 file, depending on the kind of mesh. Currently, VsHdf5 supports reading and writing of mesh objects called StructuredMesh, UnstructuredMesh, UniformCartesianMesh, and RectilinearMesh. Each of these kinds require different attributes (see below). There is also a generic mesh object that makes it easier to read a mesh from a file. If you invoke:

```
m = VsHdf5.Mesh(fileName='gamma2D01_nodalE_0.h5')
```

for instance, then VsHdf5 will search for the (single) group or dataset that is a mesh in the file, will determine the kind of mesh, and will cast itself into that particular kind. So here, for instance, since the gamma2D01 simulation was in Cylindrical coordinates (ZR), we find that the mesh object is a RectilinearMesh.

```
In [5]: m = VsHdf5.Mesh(fileName='gamma2D01_nodalE_0.h5')
In [6]: m
Out[6]: <VsHdf5.RectilinearMesh instance at 0x104b88170>
```

Rectilinear Meshes (Hdf5 Group)

meshName
: Name of the group to be written in the file. Default = objectName

axis0data
: Dataset representing the coordinates of the mesh in direction 0

axis1data
: Dataset representing the coordinates of the mesh in direction 1. Required if simulation is 2-Dimensional.

axis2data
: Dataset representing the coordinates of the mesh in direction 2. Required if simulation is 3-Dimensional.

axis0Name
: Name of the dataset corresponding to the first axis. Default = axis0

axis1Name
: Name of the dataset corresponding to the second axis. Default = axis1

axis2Name
: Name of the dataset corresponding to the third axis. Default = axis2

limits
: Name of a limits group that gives the spatial limits of the simulation

UniformCartesian Meshes (Hdf5 Group)

meshName
Name of the group to be written in the file. Default = objectName

lowerBounds
Array of physical coordinates representing the left, bottom, back corner of the simulation domain

upperBounds
Array of physical coordinates representing the right, top, front corner of the simulation domain

numCells
Array of integers giving the number of computational cells in each direction

startCell
Cell index of lowerBounds, default = [0,0,0]

Structured Meshes (Hdf5 Dataset)

meshName
Name of the group to be written in the file. Default = objectName

Unstructured Meshes (Hdf5 Group)

meshName
Name of the group to be written in the file. Default = objectName

pointsData
Dataset of physical coordinates of vertexes of the mesh (reals)

edgesData
Dataset of connectivity between vertexes of the mesh (ints)

facesData
Dataset of connectivity between vertexes of the mesh (ints)

polygonsData
Dataset of connectivity between vertexes of the mesh (ints)

pointsName
Name of the dataset containing vertex coordinates. Default = "points"

polygonsName
Name of the dataset containing vertex connectivity. Default = "polygons"

Note: Only one of edgesData, facesData, or polygonsData is needed. pointsData is always required.

Fields (Hdf5 Dataset)

fieldName
Name of the dataset to be written in the file. Default = objectName

mesh
Name of a mesh group that the field is defined upon

limits
Name of a limits group that gives the spatial limits of the simulation

timeGroup
Name of a time group that indicates the simulation time and step

offset
Centering of the data. default = 'none'

dumpTime
Simulation time when the data was dumped. default = 0.0

Histories (Hdf5 Dataset)

historyName
Name of the dataset to be written in the file. Default = objectName

mesh
Name of a mesh group that the field is defined upon, typically a 1-Dimensional time series

Particles (Hdf5 Dataset)

particlesName
Name of the dataset to be written in the file. Default = objectName

charge
The charge of the particle species

mass
The mass of the particle species

numPtclsInMacro
The number of physical particles per simulation particle

limits
Name of a limits group that gives the spatial limits of the simulation

timeGroup
Name of a time group that indicates the simulation time and step

numSpatialDims
Number of spatial dimensions. default = 3

vsNumSpatialDims
Number of spatial dimensions. default = 3

dumpTime
Simulation time when the data was dumped. default = 0.0

RunInfo (Hdf5 Group)

runInfoName
Name of the group to be written in the file. Default = objectName

TimeGroups (Hdf5 Group)

timeGroupName
Name of the group to be written in the file. Default = objectName

dumpTime
Simulation time when the data was dumped.

3.4.5 Reading in an entire file

It is also possible to parse an entire VizSchema-compliant file. To do this, invoke:

```
f = VsHdf5.VsFileReader(fileName=fileName)
```

This will create an object that contains all of the datasets and groups in the file that have VizSchema attributes. This does not guarantee that the file is VizSchema compliant however. In particular,

```
f.objectDict
```

is a Python dictionary with key equal to the name of the object (Field, Mesh, etc.) and value equal to a VsHdf5 object of the appropriate type. As with reading in individual objects, the objects in the dictionary will contain both datasets and attributes as appropriate to the type of object.

Now you can read in a group by calling it's name explicitly

```
electrons = f.objectDict['electrons']
limits = f.objectDict['globalGridGlobalLimits']
```

3.4.6 External links

VsHdf5 supports reading and writing of external links from/to files.

To write a link to an external file, use:

```
object.writeExternalLink(fileName, name, target, location='/')
```

fileName
is the name of the file to write the link into

name
is the name of the link in that file

target
is the name of the external file that contains the actual dataset or group

Location
is an optional parameter that specifies the location of the link "name" in the file "fileName", in case this is not the root group.

"object" is any VsHdf5 class

Note that the target file may or may not exist. So for instance, if you invoked:

```
mesh.writeExternalLink('outVsHdf5.vsh5', 'goodLink', 'ww02_nodalExternalB_99.h5', '/
↪globalGridGlobal')
```

This would create an external link in the file outVsHdf5.vsh5 called goodLink, that would point to the group /globalGridGlobal in the file ww02_nodalExternalB_99.h5. Subsequently, accessing the Hdf5 node goodLink would be as if the group /globalGridGlobal was actually in the file outVsHdf5.vsh5.

To read an external link from a file, simply read it as if it were a group or dataset object in the target file, e.g.

```
emesh = VsHdf5.Mesh(fileName='outVsHdf5.vsh5', name='goodLink')
```

will transparently dereference the external link called "goodLink" in the file "outVsHdf5.vsh5", and create the VsHdf5 Mesh object (appropriately cast to the proper type) as if the mesh group was in the file "outVsHdf5.vsh5". If the link is broken, namely if it points to a file with a fully qualified path that does not exist, then VsHdf5 will look for a file with the same name in the current directory. If that file in turn does not exist, or the group or dataset can not be read from an existing target file, then an error is reported and the object is not read.

3.4.7 Special Accessor methods

Some attributes are commonly used in analysis scripts, and so they have their own accessor methods. The value of any attribute can be found by

```
att = object.attribute(key)
```

where key is the name of the attribute. If the attribute does not exist in the object, then att = None.

For most objects (dataset and group types), where applicable:

```
object.getType(): return the value of 'vsType'.
object.getKind(): return the value of 'vsKind'.
object.getLowerBounds(): return the value of 'vsLowerBounds'.
object.getUpperBounds(): return the value of 'vsUpperBounds'.
object.getNumCells(): return the value of 'vsNumCells'.
object.getStartCell(): return the value of 'vsStartCell'.
object.getCentering(): return the value of 'vsCentering'.
object.getDumpTime(): return the value of 'time'.
```

For TimeGroup objects:

```
object.getTime(): return the value of 'vsTime'.
object.getDumpStep(): return the value of 'vsStep'.
```

For Particles objects:

```
object.getCharge(): return the value of 'charge'.
object.getMass(): return the value of 'mass'.
object.getNumPtclsInMacro(): return the value of 'numPtclsInMacro'.
object.getNumSpatialDims(): return the value of 'vsNumSpatialDims'.
```

3.4.8 Writing a VizSchema-compliant file from data

To write VizSchema-compliant data into a file, simply call the VsHdf5 objects write method, such as:

```
e.writeParticles(filename)
```

Similarly for fields (writeField), meshes (writeMesh), etc. See below.

Some objects require that certain attributes be defined. In this case, if you try to write the object to a file without defining those attributes, then a warning will be written to stdout and the object will not be written to a file. Similarly, if a dataset or group with the same name as the object to be written already exists in the file, it will not be overridden, and a warning will be printed.

If you have created a dataset that you would like to write into a file in a VizSchema-compliant fashion, you can do this by constructing the needed objects by hand (as opposed to reading them in from an existing file). Consider the following example:

```
lb=numpy.double(0.)
ub=numpy.double(1.)
nc=numpy.int(100)
ts = VsHdf5.UniformCartesianMesh(name='timeSeries')
ts.writeMesh('outputFile.vsh5', lowerBounds=lb, upperBounds=ub, numCells=nc)

array = numpy.zeros(100)
d = VsHdf5.History(name='myHistory')
d.assignDataset(array)
#[add any required attributes, e.g.]
d.assignAttribute('vsType','variable')
d.assignAttribute('vsMesh','timeSeries')
d.writeHistory('outputFile.vsh5')
```

This will create a file called 'outputFile.vsh5' with a 10x10 dataset called 'myHistory', with the attributes vsType=variable and vsMesh=timeSeries. Notice that we first created a mesh object that is the 1-Dimensional time series on which the history is defined. Also note that arrays and scalar numerical values that are passed to write methods must be numpy objects, not Python lists.

3.4.9 Example of Writing a VizSchema-compliant file from existing data

```
import VsHdf5

# read in the file
f=VsHdf5.VsFileReader(fileName='ECDriftQuad_electrons_1.h5')

#print file contents
print f.objectDict

electrons = f.objectDict['electrons']
limits = f.objectDict['globalGridBlobalLimits']
runInfo = f.objectDict['runInfo']
timeGroup = f.objectDict['time']

#print time group attributes
print timeGroup.attributeList
```

(continues on next page)

(continued from previous page)

```
#print the dump time and dump step from the time group
print timeGroup.getDumpTime()
print timeGroup.getDumpStep()

#overwrite the attribute "vsStep" with a new value of 1
timeGroup.assignAttribute('vsStep',1)

#overwrite the attribute "vsTime" with a new value of 1e-1
timeGroup.assignAttribute('vsTime', 1e-1)

#print new time group attributes
print timeGroup.attributeList

#give the name of the new output file
of = 'outfile.vsh5'

#write the new timeGroup data to a file and copy the rest of the file as old
electrons.writeParticles(of,timeGroup='time')
runInfo.writeRunInfo(of)
limits.writeLimits(of)
timeGroup.writeTimeGroup(of)
```