USimReferenceManual

Release 3.0.1

Tech-X Corporation

May 03, 2018

CONTENTS

1 Introduction **1**

2 Macros **3**
2.1 Mathphys Macro . . . 3
2.2 Grid Macro . . . 3
2.3 Euler Macro . . . 8
2.4 Ideal MHD Macro . . . 15
2.5 Anisotropic Conductivity Macro . . . 21

3 Grid **25**
3.1 cart (1d, 2d, 3d) . . . 25
3.2 bodyFitted (1d, 2d, 3d) . . . 26
3.3 unstructured . . . 28

4 DataStruct **31**
4.1 bin . . . 31
4.2 dynVector . . . 32
4.3 nodalArray . . . 32

5 DataStructAlias **33**

6 UpdateStep **35**

7 UpdateSequence **37**

8 Updater **39**
8.1 initialize (1d, 2d, 3d) . . . 39
8.2 linearCombiner (1d, 2d, 3d) . . . 41
8.3 uniformCombiner (1d, 2d, 3d) . . . 41
8.4 combiner (1d, 2d, 3d) . . . 42
8.5 dynVectorOperator . . . 43
8.6 equation (1d, 2d, 3d) . . . 45
8.7 computePrimitiveState(1d, 2d, 3d) . . . 45
8.8 vertexJetUpdater (1d, 2d, 3d) . . . 46
8.9 firstOrderMusclUpdater (1d, 2d, 3d) . . . 48
8.10 classicMusclUpdater (1d, 2d, 3d) . . . 50
8.11 unstructMusclUpdater (1d, 2d, 3d) . . . 52
8.12 thirdOrderMusclUpdater (1d, 2d, 3d) . . . 54
8.13 vector (1d, 2d, 3d) . . . 56
8.14 diffusion (1d, 2d, 3d) . . . 58
8.15 navierStokesViscousOperator (1d, 2d, 3d) . . . 60

8.16 kOmegaOperator (1d, 2d, 3d) . . . 63
8.17 kEpsilonOperator (1d, 2d, 3d) . . . 65
8.18 generalizedOhmsLaw (1d, 2d, 3d) . . . 67
8.19 resitiveOperator (1d, 2d, 3d) . . . 69
8.20 multiUpdater (1d, 2d, 3d) . . . 71
8.21 implicitMultiUpdater (1d, 2d, 3d) . . . 72
8.22 localOdeIntegrator (1d, 2d, 3d) . . . 76
8.23 timeStepRestrictionUpdater (1d, 2d, 3d) . . . 77
8.24 boundaryEntityGenerator (1d, 2d, 3d) . . . 79
8.25 characteristicCellLength (1d, 2d, 3d) . . . 79
8.26 entityGenerator (1d, 2d, 3d) . . . 80
8.27 minDistanceToWall (1d, 2d, 3d) . . . 80
8.28 operatorEntityGenerator (1d, 2d, 3d) . . . 81
8.29 paintEntity (1d, 2d, 3d) . . . 82
8.30 binCells (1d, 2d, 3d) . . . 82
8.31 fieldAtPoint (1d, 2d, 3d) . . . 83
8.32 intCombinedFields (1d, 2d, 3d) . . . 83
8.33 lineIntegral (1d, 2d, 3d) . . . 84
8.34 maxCombinedFields (1d, 2d, 3d) . . . 85
8.35 surfaceIntegral (1d, 2d, 3d) . . . 86
8.36 surfaceVariables (1d, 2d, 3d) . . . 87
8.37 nanChecker (1d, 2d, 3d) . . . 89
8.38 pressureDensityCorrector (1d, 2d, 3d) . . . 90
8.39 valueCorrector (1d, 2d, 3d) . . . 91

9 Time Integrator **93**

10 Preconditioner **95**

11 Hyperbolic Equations **97**
11.1 eulerEqn . . . 97
11.2 realGasEqn . . . 99
11.3 realGasEosEqn . . . 100
11.4 tenMomentEqn . . . 101
11.5 multiSpeciesSingleVelocityEqn . . . 103
11.6 mhdDednerEqn . . . 104
11.7 mhdDednerEosEqn . . . 108
11.8 gasDynamicMhdDednerEqn . . . 112
11.9 simpleTwoTemperatureMhdDednerEqn . . . 117
11.10 twoTemperatureMhdDednerEqn . . . 122
11.11 maxwellEqn . . . 127
11.12 maxwellDednerEqn . . . 129
11.13 gasDynamicMaxwellDednerEqn . . . 132
11.14 twoFluidEqn . . . 138
11.15 userDefinedEqn . . . 139

12 Algebraic Equations **143**
12.1 eulerSym . . . 144
12.2 mhdSym . . . 148
12.3 maxwellSym . . . 150
12.4 multiSpeciesSym . . . 151
12.5 twoFluidSym . . . 152
12.6 exprHyperSrc . . . 153
12.7 mhdSrc . . . 154
12.8 tenMomentFluidSrc . . . 158

12.9 twoFluidSrc . . . 160
12.10 idealGasVariables . . . 166
12.11 idealGasComputeVariables . . . 167
12.12 propaceosVariables . . . 169
12.13 propaceosComputeVariables . . . 171
12.14 sesameVariables . . . 173
12.15 sesameComputeVariables . . . 175
12.16 vanDerWaalsVariables . . . 177
12.17 vanDerWaalsComputeVariables . . . 179
12.18 bremsPowerSrc . . . 180
12.19 radiationAbsorption . . . 181
12.20 radiationEmission . . . 182
12.21 coilFieldEqn . . . 182
12.22 current . . . 183
12.23 lorentzForce . . . 184
12.24 wireFieldEqn . . . 186
12.25 computeChargeError . . . 187
12.26 hyperbolicCleanSym . . . 188
12.27 collisionFrequency . . . 188
12.28 conductivityTensor . . . 191
12.29 momentumEnergyExchange . . . 192
12.30 NFluidSrc . . . 194
12.31 reactionTableRhs . . . 195
12.32 temperatureRelaxation . . . 197
12.33 transportCoeffSrc . . . 198

13 Boundary Conditions **207**
13.1 copy (1d, 2d, 3d) . . . 207
13.2 eulerBc (1d, 2d, 3d) . . . 208
13.3 functionBc (1d, 2d, 3d) . . . 208
13.4 generalBc (1d, 2d, 3d) . . . 209
13.5 maxwellBc (1d, 2d, 3d) . . . 210
13.6 mhdBc (1d, 2d, 3d) . . . 210
13.7 periodicCartBc (1d, 2d, 3d) . . . 211
13.8 simpleBc (1d, 2d, 3d) . . . 211
13.9 sufaceEvaporation (1d, 2d, 3d) . . . 212
13.10 tenMomentBc (1d, 2d, 3d) . . . 212

14 Time Step Restriction **215**
14.1 cyclotronFrequency (1d, 2d, 3d) . . . 215
14.2 frequency (1d, 2d, 3d) . . . 216
14.3 hyperbolic (1d, 2d, 3d) . . . 217
14.4 plasmaFrequency (1d, 2d, 3d) . . . 260
14.5 positiveValue (1d, 2d, 3d) . . . 261
14.6 quadratic (1d, 2d, 3d) . . . 262
14.7 whistlerWave (1d, 2d, 3d) . . . 263

15 Multi-Species Data Files **265**
15.1 Multi-Species Chemical Reactions . . . 265
15.2 Multi-Species Specific Heat At Constant Pressure . . . 267
15.3 Multi-Species Energy of Formation, Molecular Weight, Molecular Diameter and Degrees of Freedom 269

Index **271**

CHAPTER

ONE

INTRODUCTION

USim Reference is a quick-reference manual for USim users to look up specific USim features and code block syntax for use in editing a USim input file. To learn about the complete USim simulation process, including details regarding input file format and the USim tutorials, or see examples of using USim to simulate real-world physics models, please refer to USim-in-depth.

CHAPTER

TWO

MACROS

2.1 Mathphys Macro

This macro can be imported to an input file with $ import mathphys

mathphys: In many of the input file examples that are supplied in USimComposer, 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. Refer to the mathphys file under `Macros` in USimComposer to see which constants are available.

2.2 Grid Macro

This macro file can be imported to an input file with `$ import grid.mac`.

This collection of macros can be used to add different types of grids to the input file.

Contents

- *addGrid Macro*
- *addCylindricalGrid Macro*
- *addBodyFittedGrid Macro*
- *addCylindricalBodyFittedGrid Macro*
- *addExodusGrid Macro*
- *addExodusTetrahedralGrid Macro*
- *addCylindricalExodusTetrahedralGrid Macro*
- *addCylindricalExodusGrid Macro*
- *addGmshGrid Macro*
- *addGmshTetrahedralGrid Macro*
- *addCylindricalGmshGrid Macro*
- *addCylindricalGmshTetrahedralGrid Macro*
- *addGridVariable Macro*
- *addGridPreExpression Macro*
- *addGridExpression Macro*
- *addEntityMaskVariable Macro*
- *addEntityMaskPreExpression Macro*
- *addEntityMaskExpression Macro*
- *createNewEntityFromMask (newEntityNameVar) Macro*
- *createNewEntityFromMask (newEntityNameVar, entityToCreateFromVar) Macro*

2.2.1 addGrid Macro

addGrid (lowerBounds, upperBounds, numCells, periodicDirections): Add a structured Cartesian grid

addGrid Macro Parameters

lowerBounds: Vector of coordinates for lower edge of grid, lowerBounds = [XMIN YMIN ZMIN]

upperBounds: Vector of coordinates for upper edge of grid, upperBounds = [XMAX YMAX ZMAX]

numCells: Vector of number of cells in grid, numCells = [NX NY NZ]

periodicDirections: List of directions that are periodic

```
periodicDirections = [ 0 ] (x-direction periodic)
periodicDirections = [ 0 1 ] (x,y-directions periodic)
periodicDirections = [ 0 1 2 ] (x,y,z-directions periodic)
```

2.2.2 addCylindricalGrid Macro

addCylindricalGrid (lowerBounds, upperBounds, numCells, periodicDirections): Add a structured cylindrical grid

addCylindricalGrid Macro Parameters

lowerBounds: Vector of coordinates for lower edge of grid, lowerBounds = [XMIN YMIN ZMIN]

upperBounds: Vector of coordinates for upper edge of grid, upperBounds = [XMAX YMAX ZMAX]

numCells: Vector of number of cells in grid, numCells = [NX NY NZ]

periodicDirections: List of directions that are periodic

```
periodicDirections = [ 0 ] (x-direction periodic)
periodicDirections = [ 0 1 ] (x,y-directions periodic)
periodicDirections = [ 0 1 2 ] (x,y,z-directions periodic)
```

2.2.3 addBodyFittedGrid Macro

addBodyFittedGrid (lowerBounds, upperBounds, numCells, periodicDirections): Add a body-fitted cartesian grid

addBodyFittedGrid Macro Parameters

lowerBounds: Vector of coordinates for lower edge of grid, lowerBounds = [XMIN YMIN ZMIN]

upperBounds: Vector of coordinates for upper edge of grid, upperBounds = [XMAX YMAX ZMAX]

numCells: Vector of number of cells in grid, numCells = [NX NY NZ]

periodicDirections: List of directions that are periodic

```
periodicDirections = [ 0 ] (x-direction periodic)
periodicDirections = [ 0 1 ] (x,y-directions periodic)
periodicDirections = [ 0 1 2 ] (x,y,z-directions periodic)
```

2.2.4 addCylindricalBodyFittedGrid Macro

addCylindricalBodyFittedGrid (lowerBounds, upperBounds, numCells, periodicDirections): Add a body-fitted cylindrical grid

addCylindricalBodyFittedGrid Macro Parameters

lowerBounds: Vector of coordinates for lower edge of grid, lowerBounds = [XMIN YMIN ZMIN]

upperBounds: Vector of coordinates for upper edge of grid, upperBounds = [XMAX YMAX ZMAX]

numCells: Vector of number of cells in grid, numCells = [NX NY NZ]

periodicDirections: List of directions that are periodic

```
periodicDirections = [ 0 ] (x-direction periodic)
periodicDirections = [ 0 1 ] (x,y-directions periodic)
periodicDirections = [ 0 1 2 ] (x,y,z-directions periodic)
```

2.2.5 addExodusGrid Macro

addExodusGrid (name): Add a unstructured grid in ExodusII format

addExodusGrid Macro Parameters

name: Name of grid WITHOUT extension

2.2.6 addExodusTetrahedralGrid Macro

addExodusTetrahedralGrid (name): Add a unstructured grid composed of tetrahedra in ExodusII format

addExodusTetrahedralGrid Macro Parameters

name: Name of grid WITHOUT extension

2.2.7 addCylindricalExodusTetrahedralGrid Macro

addCylindricalExodusTetrahedralGrid (name): Add a unstructured cylindrical grid composed of tetrahedra in ExodusII format

addCylindricalExodusTetrahedralGrid Macro Parameters

name: Name of grid WITHOUT extension

2.2.8 addCylindricalExodusGrid Macro

addCylindricalExodusGrid (name): Add a unstructured cylindrical grid in ExodusII format

addCylindricalExodusGrid Macro Parameters

name: Name of grid WITHOUT extension

2.2.9 addGmshGrid Macro

addGmshGrid (name): Add a unstructured grid in Gmsh format

addGmshGrid Macro Parameters

name: Name of grid WITHOUT extension

2.2.10 addGmshTetrahedralGrid Macro

addGmshTetrahedralGrid (name): Add a unstructured grid composed of tetrahedra in Gmsh format

addGmshTetrahedralGrid Macro Parameters

name: Name of grid WITHOUT extension

2.2.11 addCylindricalGmshGrid Macro

addCylindricalGmshGrid (name): Add a unstructured cylindrical grid in Gmsh format

addCylindricalGmshGrid Macro Parameters

name: Name of grid WITHOUT extension

2.2.12 addCylindricalGmshTetrahedralGrid Macro

addCylindricalGmshTetrahedralGrid (name): Add a unstructured cylindrical grid composed of tetrahedra in Gmsh format

addCylindricalGmshTetrahedralGrid Macro Parameters

name: Name of grid WITHOUT extension

2.2.13 addGridVariable Macro

addGridVariable (varName, varValue): Specify a variable for defining a body-fitted grid

addGridVariable Macro Parameters

varName: Name to assign quantity that is independent of grid position

varValue: Value to assign quantity that is independent of grid position

2.2.14 addGridPreExpression Macro

addGridPreExpression (expression): Specify a preExpression for defining a body-fitted grid

addGridPreExpression Macro Parameters

expression: A mathematical expression to evaluate. f=f(preExpression,variable,t,x,y,z)

2.2.15 addGridExpression Macro

addGridExpression (expression): Specify an Expression for defining a body-fitted grid

addGridExpression Macro Parameters

expression: A mathematical expression to evaluate. f=f(preExpression,variable,t,x,y,z)

2.2.16 addEntityMaskVariable Macro

addEntityMaskVariable (newEntityNameVar, varName, varValue): Specify a variable for defining a mask on the grid

addEntityMaskVariable Macro Parameters

newEntityNameVar: Name of the new entity

varName: Name to assign quantity that is independent of grid position

varValue: Value to assign quantity that is independent of grid position

2.2.17 addEntityMaskPreExpression Macro

addEntityMaskPreExpression (newEntityNameVar, expression): Specify a preExpression for defining a mask on the grid

addEntityMaskPreExpression Macro Parameters

expression: A mathematical expression to evaluate. f=f(preExpression,variable,t,x,y,z)

2.2.18 addEntityMaskExpression Macro

addEntityMaskExpression (newEntityNameVar, expression): Specify an Expression for defining a mask on the grid

addEntityMaskExpression Macro Parameters

expression: A mathematical expression to evaluate. f=f(preExpression,variable,t,x,y,z)

2.2.19 createNewEntityFromMask (newEntityNameVar) Macro

createNewEntityFromMask (newEntityNameVar): Create a new entity within the grid based on mask function

createNewEntityFromMask Macro Parameters

newEntityNameVar: Name of the new entity

2.2.20 createNewEntityFromMask (newEntityNameVar, entityToCreateFromVar) Macro

createNewEntityFromMask (newEntityNameVar, entityToCreateFromVar): Create a new entity within the grid based on mask function

createNewEntityFromMask Macro Parameters

newEntityNameVar: Name of the new entity

entityToCreateFromVar: Name of the entity to create the new entity from

2.3 Euler Macro

This macro file can be imported to an input file with `$ import euler.mac`.

This collection of macros can be used to define quantities required for the Euler equations to input files.

Contents

- *initializeFluidSimulation Macro*
- *createFluidSimulation Macro*
- *addVariable Macro*
- *addPreExpression Macro*
- *addExpression Macro*
- *finiteVolumeScheme Macro*
- *addGravitationalAcceleration Macro*
- *addBoundaryConditionVariable (name, varName, varValue) Macro*
- *addBoundaryConditionVariable (name, entityName, varName, varValue) Macro*
- *addBoundaryConditionPreExpression (name, expression) Macro*
- *addBoundaryConditionPreExpression (name, entityName, expression) Macro*
- *addBoundaryConditionExpression (name, expression) Macro*
- *addBoundaryConditionExpression (name, entityName, expression) Macro*
- *boundaryCondition (type) Macro*
- *boundaryCondition (type, entityName) Macro*
- *boundaryCondition (name, type, entityName) Macro*
- *timeAdvance Macro*
- *diffusionTimeAdvance Macro*
- *implicitTimeAdvance Macro*
- *addOutputDiagnostic (name) Macro*
- *addOutputDiagnostic (name, numberOfComponents, isVector) Macro*
- *addOutputDiagnosticParameter Macro*
- *addOutputDiagnosticPreExpression Macro*
- *addOutputDiagnosticExpression Macro*
- *runFluidSimulation Macro*

2.3.1 initializeFluidSimulation Macro

`initializeFluidSimulation` (dimensionality,tStart,tEnd,numFrames,cflNum,gammaIn,writeRestartIn,debugIn):
Define quanties required for the Euler equations.

initializeFluidSimulation Macro Parameters

`dimensionality`: 1,2,3. Number of dimensions for the simulation

`tStart`: Start time for simulation

`tEnd`: End time for simulation

`numFrames`: Number of data outputs

`cflNum`: Cfl limit, typically $\Delta t = cflNum * \Delta x / V_{max}$

`gammaIn`: Adiabatic index for ideal gas eqn. of state. Pressure = (gammaIn - 1.0) * density * internal energy

`writeRestartIn`: Output data required for simulation restart

`debugIn`: Run simulation in debug mode

2.3.2 createFluidSimulation Macro

createFluidSimulation (): Define and add the various data structures, initial conditions and primitive variable computations required for the Euler equations.

2.3.3 addVariable Macro

addVariable (varName, varValue): Specify a variable in the initial condition

addVariable Macro Parameters

varName: Name to assign the quantity that is independent of grid position

varValue: Value to assign the quantity that is independent of grid position

2.3.4 addPreExpression Macro

addPreExpression (expression): Specify a preExpression in the initial condition

addPreExpression Macro Parameters

expression: The mathematical expression to evaluate. f=f(preExpression,variable,t,x,y,z)

2.3.5 addExpression Macro

addExpression (expression): Specify an Expression in the initial condition

addExpression Macro Parameters

expression: The mathematical expression to evaluate. f=f(preExpression,variable,t,x,y,z)

2.3.6 finiteVolumeScheme Macro

finteVolumeScheme (diffusive): Add a finite volume scheme for solving the Euler equations

finiteVolumeScheme Macro Parameters

diffusive: True/False. Utilize a diffusive, but robust scheme to solve the system

2.3.7 addGravitationalAcceleration Macro

addGravitationalAcceleration (gravitationalAcceleration): Add a gravitational acceleration source block to the finiteVolumeScheme

addGravitationalAccleration Macro Parameters

gravitationalAcceleration: Acceleration to apply in negative y-direction

2.3.8 addBoundaryConditionVariable (name, varName, varValue) Macro

`addBoundaryConditionVariable` (name, varName, varValue): Specify a variable on a userSpecified boundary condition on the ghost entity

addBoundaryConditionVariable Macro Parameters

`name`: The type of the boundary condition to apply. Must be *userSpecified*.

`varName`: Name to assign the quantity that is independent of grid position

`varValue`: Value to assign the quantity that is independent of grid position

2.3.9 addBoundaryConditionVariable (name, entityName, varName, varValue) Macro

`addBoundaryConditionVariable` (name, entityName, varName, varValue): Specify a variable on a user-Specified boundary condition on the ghost entity

addBoundaryConditionVariable Macro Parameters

`name`: The type of the boundary condition to apply. Must be *userSpecified*.

`entityName`: The boundary entity to apply boundary condition on

`varName`: Name to assign the quantity that is independent of grid position

`varValue`: Value to assign the quantity that is independent of grid position

2.3.10 addBoundaryConditionPreExpression (name, expression) Macro

`addBoundaryConditionPreExpression` (name, expression): Specify a preExpression on a userSpecified boundary condition on the ghost entity.

addBoundaryConditionPreExpression Macro Parameters

`name`: The type of the boundary condition to apply. Must be *userSpecified*.

`expression`: The mathematical expression to evaluate. f=f(preExpression,variable,t,x,y,z)

2.3.11 addBoundaryConditionPreExpression (name, entityName, expression) Macro

`addBoundaryConditionPreExpression` (name, entityName, expression): Specify a preExpression on a userSpecified boundary condition on the ghost entity.

addBoundaryConditionPreExpression Macro Parameters

name: The type of the boundary condition to apply. Must be *userSpecified*.

entityName: The boundary entity to apply boundary condition on

expression: The mathematical expression to evaluate. f=f(preExpression,variable,t,x,y,z)

2.3.12 addBoundaryConditionExpression (name, expression) Macro

addBoundaryConditionExpression (name, expression): Specify an Expression on a userSpecified boundary condition on the ghost entity.

addBoundaryConditionExpression Macro Parameters

name: The type of the boundary condition to apply. Must be *userSpecified*.

expression: The mathematical expression to evaluate. f=f(preExpression,variable,t,x,y,z)

2.3.13 addBoundaryConditionExpression (name, entityName, expression) Macro

addBoundaryConditionExpression (name, entityName, expression): Specify an Expression on a userSpecified boundary condition on the ghost entity.

addBoundaryConditionExpression Macro Parameters

name: The type of the boundary condition to apply. Must be *userSpecified*.

entityName: The boundary entity to apply boundary condition on

expression: The mathematical expression to evaluate. f=f(preExpression,variable,t,x,y,z)

2.3.14 boundaryCondition (type) Macro

boundaryCondition (type): Apply a boundary condition for the Euler equations on the ghost entity

boundaryCondition Macro Parameters

type: The type of the boundary condition to apply. Can be one of: *periodic*, *copy*, *userSpecified*, *wall*, *noInflow*, *noSlip*.

2.3.15 boundaryCondition (type, entityName) Macro

boundaryCondition (type, entityName): Apply a boundary condition for the Euler equations on the ghost entity

boundaryCondition Macro Parameters

type: The type of the boundary condition to apply. Can be one of: *periodic*, *copy*, *userSpecified*, *wall*, *noInflow*, *noSlip*.

entityName: The boundary entity to apply boundary condition on

2.3.16 boundaryCondition (name, type, entityName) Macro

`boundaryCondition` (name, type, entityName): Apply a boundary condition for the Euler equations on the ghost entity

boundaryCondition Macro Parameters

`name`: The name to assign to the boundary condition

`type`: The type of the boundary condition to apply. Can be one of: *periodic*, *copy*, *userSpecified*, *wall*, *noInflow*, *noSlip*.

`entityName`: The boundary entity to apply boundary condition on

2.3.17 timeAdvance Macro

`timeAdvance` (order): Advance simulation in time using an explicit Runge-Kutta method

timeAdvance Macro Parameters

`order`: The order of explicit Runge-Kutta method. Can be first, second, third, fourth

2.3.18 diffusionTimeAdvance Macro

`diffusionTimeAdvance` (order): Advance simulation in time using an explicit Super-Time Step method for diffusion problems

diffusionTimeAdvance Macro Parameters

`order`: The order of explicit Super-Time Step method. Can be first, second

2.3.19 implicitTimeAdvance Macro

`implicitTimeAdvance` (order): Advance simulation in time using an implicit method for Poisson type problems

implicitTimeAdvance Macro Parameters

`order`: The order of implicit method. Currently only None is supported

2.3.20 addOutputDiagnostic (name) Macro

`addOutputDiagnostic` (name): Add a scalar output diagnostic

addOutputDiagnostic Macro Parameters

`name`: The name of the output diagnostic

2.3.21 addOutputDiagnostic (name, numberOfComponents, isVector) Macro

`addOutputDiagnostic` (name, numberOfComponents, isVector): Add a multi-component output diagnostic

addOutputDiagnostic Macro Parameters

`name`: The name of the output diagnostic

`numberOfComponents`: The number of components for the output diagnostic

`isVector`: True/False. Is the output diagnostic a vector quantity.

2.3.22 addOutputDiagnosticParameter Macro

`addOutputDiagnosticParameter` (name, varName, varValue): Specify a paramter in an output diagnostic

addOutputDiagnosticParameter Macro Parameters

`name`: The type of the output diagnostic

`varName`: Name to assign the quantity that is independent of grid position

`varValue`: Value to assign the quantity that is independent of grid position

2.3.23 addOutputDiagnosticPreExpression Macro

`addOutputDiagnosticPreExpression` (name, expression): Specify a preExpression in an output diagnostic. Available pre-defined quantities are

```
preDefined = [rho rhoVx rhoVy rhoVz En Vx Vy Vz P ]
```

addOutputDiagnosticPreExpression Macro Parameters

`name`: The type of the output diagnostic

`expression`: The mathematical expression to evaluate. f=f(preExpression,preDefined,variable,t,x,y,z)

2.3.24 addOutputDiagnosticExpression Macro

`addOutputDiagnosticExpression` (name, expression): Specify an Expression in an output diagnostic. Available pre-defined quantities are

```
preDefined = [rho rhoVx rhoVy rhoVz En Vx Vy Vz P ]
```

addOutputDiagnosticExpression Macro Parameters

`name`: The type of the output diagnostic

`expression`: The mathematical expression to evaluate. f=f(preExpression,preDefined,variable,t,x,y,z)

2.3.25 runFluidSimulation Macro

runFluidSimulation (): Evaluate all macros added to the .pre file and generate the .in file

2.4 Ideal MHD Macro

This macro file can be imported to an input file with `$ import idealmhd.mac`.

This collection of macros can be used to define quantities required for the MHD equations.

Contents

- *initializeFluidSimulation Macro*
- *createFluidSimulation Macro*
- *addVariable Macro*
- *addPreExpression Macro*
- *addExpression Macro*
- *finiteVolumeScheme (diffusive) Macro*
- *finiteVolumeScheme (diffusive,basementPressureIn,basementDensityIn) Macro*
- *addGravitationalAcceleration Macro*
- *addBoundaryConditionVariable (name, varName, varValue) Macro*
- *addBoundaryConditionVariable (name, entityName, varName, varValue) Macro*
- *addBoundaryConditionPreExpression (name, expression) Macro*
- *addBoundaryConditionPreExpression (name, entityName, expression) Macro*
- *addBoundaryConditionExpression (name, expression) Macro*
- *addBoundaryConditionExpression (name, entityName, expression) Macro*
- *boundaryCondition (type) Macro*
- *boundaryCondition (type, entityName) Macro*
- *timeAdvance Macro*
- *diffusionTimeAdvance Macro*
- *implicitTimeAdvance Macro*
- *addOutputDiagnostic (name) Macro*
- *addOutputDiagnostic (name, numberOfComponents, isVector) Macro*
- *addOutputDiagnosticParameter Macro*
- *addOutputDiagnosticPreExpression Macro*
- *addOutputDiagnosticExpression Macro*
- *runFluidSimulation Macro*

2.4.1 initializeFluidSimulation Macro

initializeFluidSimulation (dimensionality,tStart,tEnd,numFrames,cflNum,gammaIn,muIn,writeRestartIn,debugIn):
Define quanties required for the Euler equations.

initializeFluidSimulation Macro Parameters

dimensionality: 1,2,3. Number of dimensions for the simulation

tStart: Start time for simulation

tEnd: End time for simulation

numFrames: Number of data outputs

cflNum: Cfl limit, typically $\Delta t = cflNum * \Delta x / V_{max}$

gammaIn: Adiabatic index for ideal gas eqn. of state. Pressure = (gammaIn - 1.0) * density * internal energy

muIn: Permeability of free space

writeRestartIn: Output data required for simulation restart

debugIn: Run simulation in debug mode

2.4.2 createFluidSimulation Macro

createFluidSimulation (): Define and add the various data structures, initial conditions and primitive variable computations required for the MHD equations.

2.4.3 addVariable Macro

addVariable (varName, varValue): Specify a variable in the initial condition

addVariable Macro Parameters

varName: Name to assign the quantity that is independent of grid position

varValue: Value to assign the quantity that is independent of grid position

2.4.4 addPreExpression Macro

addPreExpression (expression): Specify a preExpression in the initial condition

addPreExpression Macro Parameters

expression: The mathematical expression to evaluate. f=f(preExpression,variable,t,x,y,z)

2.4.5 addExpression Macro

addExpression (expression): Specify an Expression in the initial condition

addExpression Macro Parameters

expression: The mathematical expression to evaluate. f=f(preExpression,variable,t,x,y,z)

2.4.6 finiteVolumeScheme (diffusive) Macro

finteVolumeScheme (diffusive): Add a finite volume scheme for solving the MHD equations

finiteVolumeScheme Macro Parameters

diffusive: True/False. Utilize a diffusive, but robust scheme to solve the system

2.4.7 finiteVolumeScheme (diffusive,basementPressureIn,basementDensityIn) Macro

`finteVolumeScheme` (diffusive,basementPressureIn,basementDensityIn): Add a finite volume scheme for solving the MHD equations with a density, pressure floor

finiteVolumeScheme Macro Parameters

`diffusive`: True/False. Utilize a diffusive, but robust scheme to solve the system

`basementPressureIn`: Floor value for pressure

`basementDensityIn`: Floor value for density

2.4.8 addGravitationalAcceleration Macro

`addGravitationalAcceleration` (gravitationalAcceleration): Add a gravitational acceleration source block to the finiteVolumeScheme

addGravitationalAcceleration Macro Parameters

`gravitationalAcceleration`: Acceleration to apply in negative y-direction

2.4.9 addBoundaryConditionVariable (name, varName, varValue) Macro

`addBoundaryConditionVariable` (name, varName, varValue): Specify a variable on a userSpecified boundary condition on the ghost entity

addBoundaryConditionVariable Macro Parameters

`name`: The type of the boundary condition to apply. Must be *userSpecified*.

`varName`: Name to assign the quantity that is independent of grid position

`varValue`: Value to assign the quantity that is independent of grid position

2.4.10 addBoundaryConditionVariable (name, entityName, varName, varValue) Macro

`addBoundaryConditionVariable` (name, entityName, varName, varValue): Specify a variable on a user-Specified boundary condition on the ghost entity

addBoundaryConditionVariable Macro Parameters

`name`: The type of the boundary condition to apply. Must be *userSpecified*.

`entityName`: The boundary entity to apply boundary condition on

`varName`: Name to assign the quantity that is independent of grid position

`varValue`: Value to assign the quantity that is independent of grid position

2.4.11 addBoundaryConditionPreExpression (name, expression) Macro

addBoundaryConditionPreExpression (name, expression): Specify a preExpression on a userSpecified boundary condition on the ghost entity.

addBoundaryConditionPreExpression Macro Parameters

name: The type of the boundary condition to apply. Must be *userSpecified*.

expression: The mathematical expression to evaluate. f=f(preExpression,variable,t,x,y,z)

2.4.12 addBoundaryConditionPreExpression (name, entityName, expression) Macro

addBoundaryConditionPreExpression (name, entityName, expression): Specify a preExpression on a userSpecified boundary condition on the ghost entity.

addBoundaryConditionPreExpression Macro Parameters

name: The type of the boundary condition to apply. Must be *userSpecified*.

entityName: The boundary entity to apply boundary condition on

expression: The mathematical expression to evaluate. f=f(preExpression,variable,t,x,y,z)

2.4.13 addBoundaryConditionExpression (name, expression) Macro

addBoundaryConditionExpression (name, expression): Specify an Expression on a userSpecified boundary condition on the ghost entity.

addBoundaryConditionExpression Macro Parameters

name: The type of the boundary condition to apply. Must be *userSpecified*.

expression: The mathematical expression to evaluate. f=f(preExpression,variable,t,x,y,z)

2.4.14 addBoundaryConditionExpression (name, entityName, expression) Macro

addBoundaryConditionExpression (name, entityName, expression): Specify an Expression on a userSpecified boundary condition on the ghost entity.

addBoundaryConditionExpression Macro Parameters

name: The type of the boundary condition to apply. Must be *userSpecified*.

entityName: The boundary entity to apply boundary condition on

expression: The mathematical expression to evaluate. f=f(preExpression,variable,t,x,y,z)

2.4.15 boundaryCondition (type) Macro

`boundaryCondition` (type): Apply a boundary condition for the MHD equations on the ghost entity

boundaryCondition Macro Parameters

`type`: The type of the boundary condition to apply. Can be one of: *periodic*, *copy*, *userSpecified*, *wall*, *noInflow*, *noSlip*.

2.4.16 boundaryCondition (type, entityName) Macro

`boundaryCondition` (type, entityName): Apply a boundary condition for the Euler equations on the ghost entity

boundaryCondition Macro Parameters

`type`: The type of the boundary condition to apply. Can be one of: *periodic*, *copy*, *userSpecified*, *wall*, *noInflow*, *noSlip*.

`entityName`: The boundary entity to apply boundary condition on

2.4.17 timeAdvance Macro

`timeAdvance` (order): Advance simulation in time using an explicit Runge-Kutta method

timeAdvance Macro Parameters

`order`: The order of explicit Runge-Kutta method. Can be first, second, third, fourth

2.4.18 diffusionTimeAdvance Macro

`diffusionTimeAdvance` (order): Advance simulation in time using an explicit Super-Time Step method for diffusion problems

diffusionTimeAdvance Macro Parameters

`order`: The order of explicit Super-Time Step method. Can be first, second

2.4.19 implicitTimeAdvance Macro

`implicitTimeAdvance` (order): Advance simulation in time using an implicit method for Poisson type problems

implicitTimeAdvance Macro Parameters

`order`: The order of implicit method. Currently only None is supported

2.4.20 addOutputDiagnostic (name) Macro

addOutputDiagnostic (name): Add a scalar output diagnostic

addOutputDiagnostic Macro Parameters

name: The name of the output diagnostic

2.4.21 addOutputDiagnostic (name, numberOfComponents, isVector) Macro

addOutputDiagnostic (name, numberOfComponents, isVector): Add a multi-component output diagnostic

addOutputDiagnostic Macro Parameters

name: The name of the output diagnostic

numberOfComponents: The number of components for the output diagnostic

isVector: True/False. Is the output diagnostic a vector quantity.

2.4.22 addOutputDiagnosticParameter Macro

addOutputDiagnosticParameter (name, varName, varValue): Specify a paramter in an output diagnostic

addOutputDiagnosticParameter Macro Parameters

name: The type of the output diagnostic

varName: Name to assign the quantity that is independent of grid position

varValue: Value to assign the quantity that is independent of grid position

2.4.23 addOutputDiagnosticPreExpression Macro

addOutputDiagnosticPreExpression (name, expression): Specify a preExpression in an output diagnostic. Available pre-defined quantities are

```
preDefined = [rho rhoVx rhoVy rhoVz En Vx Vy Vz P Pb divB Jx Jy Jz]
```

addOutputDiagnosticPreExpression Macro Parameters

name: The type of the output diagnostic

expression: The mathematical expression to evaluate. f=f(preExpression,preDefined,variable,t,x,y,z)

2.4.24 addOutputDiagnosticExpression Macro

addOutputDiagnosticExpression (name, expression): Specify an Expression in an output diagnostic. Available pre-defined quantities are

```
preDefined = [rho rhoVx rhoVy rhoVz En Vx Vy Vz P Pb divB Jx Jy Jz ]
```

addOutputDiagnosticExpression Macro Parameters

name: The type of the output diagnostic

expression: The mathematical expression to evaluate. f=f(preExpression,preDefined,variable,t,x,y,z)

2.4.25 runFluidSimulation Macro

runFluidSimulation (): Evaluate all macros added to the .pre file and generate the .in file

2.5 Anisotropic Conductivity Macro

This macro file can be imported to an input file with `$ import anisotropicConductivity.mac`.

This collection of macros can be used to add the computation of anisotropic conductivity tensor to MHD input files.

Contents

- *addAnisotropicConductivity ( ) Macro*
- *addAnisotropicConductivity (DIFFUSION_CFL) Macro*
- *addKPerpendicularParameter Macro*
- *addKPerpendicularPreExpression Macro*
- *addKPerpendicularExpression Macro*
- *addKParallelParameter Macro*
- *addKParallelPreExpression Macro*
- *addKParallelExpression Macro*
- *anisotropicDiffusionScheme Macro*
- *diffusionTimeAdvance Macro*

2.5.1 addAnisotropicConductivity ( ) Macro

addAnisotropicConductivity ( ): Adds computation of anisotropic conductivity tensor to MDH input files.

2.5.2 addAnisotropicConductivity (DIFFUSION_CFL) Macro

addAnisotropicConductivity (DIFFUSION_CFL): Adds computation of anisotropic conductivity tensor to MDH input files.

addAnisotropicConductivity Macro Parameters

DIFFUSION_CFL: Apply a separate CFL condition to the diffusion solve, specified by the value of DIFFUSION_CFL

2.5.3 addKPerpendicularParameter Macro

addKPerpendicularParameter (varName, varValue): Specify a variable for computing kPerpendicular

addKPerpendicularParameter Macro Parameters

varName: Name to assign the quantity that is independent of grid position

varValue: Value to assign the quantity that is independent of grid position

2.5.4 addKPerpendicularPreExpression Macro

addKPerpendicularPreExpression (expression): Specify a preExpression for computing kPerpendicular

addKPerpendicularPreExpression Macro Parameters

expression: The mathematical expression to evaluate. f=f(preExpression,variable,t,x,y,z)

2.5.5 addKPerpendicularExpression Macro

addKPerpendicularExpression (expression): Specify an Expression for computing kPerpendicular

addKPerpendicularExpression Macro Parameters

expression: The mathematical expression to evaluate. f=f(expression,variable,t,x,y,z)

2.5.6 addKParallelParameter Macro

addKParallelParameter (varName, varValue): Specify a variable for computing kParallel

addKParallelParameter Macro Parameters

varName: Name to assign the quantity that is independent of grid position

varValue: Value to assign the quantity that is independent of grid position

2.5.7 addKParallelPreExpression Macro

addKParallelPreExpression (expression): Specify a preExpression for computing kParallel

addKParallelPreExpression Macro Parameters

expression: The mathematical expression to evaluate. f=f(preExpression,variable,t,x,y,z)

2.5.8 addKParallelExpression Macro

addKParallelExpression (expression): Specify an Expression for computing kParallel

addKParallelExpression Macro Parameters

expression: The mathematical expression to evaluate. f=f(expression,variable,t,x,y,z)

2.5.9 anisotropicDiffusionScheme Macro

anisotroipcDiffusionScheme ( ): Add a finite volume discretization of an anisotropic diffusion operator, $\nabla \cdot \kappa \nabla \Phi$

2.5.10 diffusionTimeAdvance Macro

diffusionTimeAdvance (order): Add an explicit Super-Time Step scheme for evolving the total energy of the gas with an anisotropic conductivity tensor

diffusionTimeAdvance Macro Parameters

order: Order of the method. Can be 'first', 'second'.

CHAPTER

THREE

GRID

Defines the simulation grid for USim. An example *Grid* block is shown below:

```
<Grid domain>
   kind = cart1d
   ghostLayers = 2
   lower = [0.0]
   upper = [1.0]
   cells = [512]
</Grid>
```

The common parameters accepted by this updater block are listed below:

`kind` (string) All *Grid* blocks take a string *kind* that species the type of USim simulation grid. The different kinds of grid available in USim are:

3.1 cart (1d, 2d, 3d)

Used for defining a grid with regular spacing in the x, y, and z directions.

3.1.1 Parameters

`lower` (float vector) Defines the lower x, y, z coordinates in the form lower=[XLOWER, YLOWER, ZLOWER]. In 1D only 1 component is required, in 2D 2 components, in 3D 3 components. Extra components are ignored.

`upper` (float vector) Defines the upper x, y, z coordinates in the form upper=[XUPPER, YUPPER, ZUPPER]. In 1D only 1 component is required, in 2D 2 components, in 3D 3 components. Extra components are ignored.

`cells` (integer vector) Defines the number of cells in the domain, cells=[CELLSX, CELLSY, CELLSZ]. Extra components are ignored, and in 1D only the first component is used, similarly in 2D only the first 2 components are used and in 3D the first 3 components are used.

`isRadial` (boolean) Defines whether or not coordinates are radial (r, θ, z) or Cartesian (x, y, z).

`periodicDirs` (integer vector) Define the directions where periodic boundary conditions will be applied. perioddicDirs = [0, 1] tells USim that the grid is periodic in the X and Y directions if a periodicCartBc is called.

`ghostLayers` (integer) Tell USim how many ghost layers the grid should use. The default value is 1.

`writeGeom` (boolean) Tell USim whether or not to write out geometrical data. Defaults to false.

`writeConn` (boolean) Tell USim whether or not to write out connectivity data. Defaults to false.

writeHalos (boolean) Tell USim whether the halo data and grid should be dumped. Defaults to false. Useful for debugging.

3.1.2 Example

Sample code block

```
<Grid domain>
  kind = cart2d
  ghostLayers = 2
  lower = [0.0, 0.0]
  upper = [1.0, 1.0]
  cells = [CELLX, CELLSY]
  periodicDirs = [0]
  isRadial = false
</Grid>
```

3.2 bodyFitted (1d, 2d, 3d)

Used for defining block structured body fitted grid by defining the vertices of each cell. Maps a regular Cartesian grid to a more complex grid with curved boundaries. Instructions on generating body fitted meshes are given in usimbase-tutorial-lesson-4.

3.2.1 Parameters

Vertices (block) Defines the coordinates of the vertices of the grid

lower (float vector) Defines the lower mapping x, y, z coordinates in the form lower=[XLOWER, YLOWER, ZLOWER]. In 1D only 1 component is required, in 2D 2 components, in 3D 3 components. Extra components are ignored.

upper (float vector) Defines the upper mapping x, y, z coordinates in the form upper=[XUPPER, YUPPER, ZUPPER]. In 1D only 1 component is required, in 2D 2 components, in 3D 3 components. Extra components are ignored.

When combined with the Vertex, kind = funcVertCalc, the vertex mapping in the x direction ranges from XLOWER to XUPPER and the true x position is xnew = f(x) for x between XLOWER and XUPPER

cells (integer vector) Defines the number of cells in the domain, cells=[CELLSX, CELLSY, CELLSZ]. Extra components are ignored, and in 1D only the first component is used, similarly in 2D only the first 2 components are used and in 3D the first 3 components are used.

isRadial (boolean) Defines whether or not coordinates are radial (r, θ, z) or Cartesian (x, y, z). Defaults to false or cartesian coordinates

ghostLayers (integer) Tell USim how many ghost layers the grid should use. The default value is 1.

writeGeom (boolean) Tell USim whether or not to write out geometrical data. Defaults to false.

writeConn (boolean) Tell USim whether or not to write out connectivity data. Defaults to false.

writeHalos (boolean) Tell USim whether the halo data and grid should be dumped. Defaults to false. Useful for debugging.

3.2.2 Example

Sample code block

```
<Grid domain>
  kind = bodyFitted2d

  lower = [0.0, 0.0]
  upper = [1.0, 1.0]
  cells = [R_CELLS, $Z_INLET_CELLS+Z_CURVE_CELLS$]
  ghostLayers = 2

  isRadial = 1

  <Vertices vertices>
    kind = funcVertCalc

    <Function f>
      kind = exprFunc

      z_inlet = Z_INLET

      z_inlet_cells = $Z_INLET_CELLS*1.0$
      z_curve_cells = $Z_CURVE_CELLS*1.0$

      r_inner = R_INNER
      r_outer = R_OUTER
      r_cells = $R_CELLS*1.0$

      dz_inlet = $Z_INLET/Z_INLET_CELLS$

      circ_rad = CIRC_RAD

      # cell spacings in computational space
      dzc = $1.0/(Z_INLET_CELLS+Z_CURVE_CELLS)$
      drc = $1.0/R_CELLS$
      zc_inlet = $1.0*Z_INLET_CELLS/(Z_INLET_CELLS+Z_CURVE_CELLS)$

      preExprs = [ \
      "r = x", \
      "z = y", \
      "iz = rint(z/dzc)", \
      "ir = rint(r/drc)", \
      "zp_inlet = if (iz<=z_inlet_cells, dz_inlet*iz, 0.0)", \
      "rp_inlet = if (iz<=z_inlet_cells, r_inner+ir*(r_outer-r_inner)/r_cells, 0.0)", \
      "rr = r_inner + r*(r_outer-r_inner)", \
      "zz = 0.5*_pi*(z-zc_inlet)/(1.0-zc_inlet)", \
      "rp_curve = if (iz>z_inlet_cells, rr*cos(zz), 0.0)", \
      "zp_curve = if (iz>z_inlet_cells, rr*sin(zz) + z_inlet, 0.0)", \
      "rp = rp_curve+rp_inlet", \
      "zp = zp_curve+zp_inlet" \
      ]

      exprs = ["rp", "zp"]

    </Function>
  </Vertices>
</Grid>
```

3.3 unstructured

Reads an unstructured grid generated by an external tool into USim. Currently USim does not do its own decomposition so it is assumed that the decomposition data is stored in the grid. Details on generating grids using GMSH and CUBIT/Trelis are given in usimbase-tutorial-lesson-5.

3.3.1 Parameters

`Creator` (block) Defines the what type of grid will be read in. USim currently supports 2 different types of grids: Exodus meshes (frequently generated by CUBIT or Trelis):

```
<Creator ctor>
  kind = exodus
  ndim = 3
  file = NElementCube_1000.g
</Creator>
```

or

```
<Creator ctor>
  kind = exodus
  ndim = 3
  file = NElementCube_1000.exo
</Creator>
```

and meshes generated with gmsh:

```
<Creator ctor>
  kind = gmsh
  ndim = 2
  file = rampgeom.msh
</Creator>
```

`isRadial` (boolean) Defines whether or not coordinates are radial (r, θ, z) or Cartesian (x, y, z). Defaults to false or cartesian coordinates

`ghostLayers` (integer) Tell USim how many ghost layers the grid should use. The default value is 1.

`writeGeom` (boolean) Tell USim whether or not to write out geometrical data. Defaults to false.

`writeConn` (boolean) Tell USim whether or not to write out connectivity data. Defaults to false.

`writeHalos` (boolean) Tell USim whether the halo data and grid should be dumped. Defaults to false. Useful for debugging.

3.3.2 Example

Sample code block

```
<Grid domain>
  kind = unstructured

  ghostLayers = 2

  <Creator ctor>
    kind = gmsh
    ndim = 2
```

```
        file = rampgeom.msh
      </Creator>
    </Grid>
```

CHAPTER

FOUR

DATASTRUCT

Basic data structure of USim. Updaters perform operations on DataStructs and write out to DataStructs. An example *DataStruct* block is shown below:

```
<DataStruct q>
  kind = nodalArray
  onGrid = domain
  numComponents = 9
</DataStruct>
```

The parameters accepted by this updater block are listed below:

`onGrid` (string) All data structures take a string that tells the data structure which grid it is defined on.

`writeOut` (boolean) Tells USim whether to write out data from the *DataStruct* or not.

`kind` (string) All DataStruct blocks take a string *kind* that specifies the type of DataStruct. The different kinds of DataStruct available in USim are:

4.1 bin

A bin is a data structure for grouping unstructured grid elements into a regular grid. The bin superimposes a regular grid over a structured or unstructured grid. Each element of the bin stores data about which cells within the structured or unstructured grid are contained within its boundaries. The data structure is used for such things as computing line integrals on unstructured meshes since it reduces the search time for finding what cell a particular point belong in.

4.1.1 Parameters

`scale` (float) Estimate for how much larger (in length) a typical bin element is than the average grid size. If domain is a regular grid with size 100X100 and scale = 2.0 then the bin would be a regular grid with size 50X50 with the same extents as the domain.

4.1.2 Example

```
<DataStruct cellBin>
  kind = bin
  onGrid = domain
  scale = 2.0
</DataStruct>
```

4.2 dynVector

A dynVector is a single vector that has the same value on all processor domains. The dynVector changes with time and its value is recorded at every time step in the output files when writeOut = true. An example of the use of of the dynVector would be storing the total energy of the system at every time step.

4.2.1 Parameters

`numComponents` (integer) Defines the number of components in the dynVector.

4.2.2 Example

```
<DataStruct totalMass>
  kind = dynVector
  onGrid = domain
  numComponents = 1
</DataStruct>
```

4.3 nodalArray

A nodalArray is a distributed array. This means during parallel runs the array is distributed across the different MPI domains. The nodalArray knows how to synchronize domain boundaries.

4.3.1 Parameters

`numComponents` (integer) Defines the number of components in the distributed array. Solving the Euler equations requires 5 variables per grid point so in this case numComponents=5.

`useEpetraVector` (boolean) Use an Epetra compatible version of the nodal component array. Required if the data structured is used as an input/output vector for *implicitMultiUpdater (1d, 2d, 3d)*

4.3.2 Example

```
<DataStruct q>
  kind = nodalArray
  onGrid = domain
  numComponents = 18
</DataStruct>
```

CHAPTER

FIVE

DATASTRUCTALIAS

DataStructAlias is a pointer to a *DataStruct*. The *DataStructAlias* can be used everywhere a *DataStruct* can be used. An example *DataStructAlias* block is shown below:

```
<DataStructAlias electronDensity>
  kind = nodalArray
  target = q
  componentRange = [0,1]
  writeOut = false
</DataStructAlias>
```

The parameters accepted by this updater block are listed below:

`kind` (string) The kind of *DataStruct*. This must be nodalArray.

`target` (string) The *DataStruct* that the *DataStructAlias* is pointing to. *DataStructAlias* only works with *DataStruct* of kind = nodalArray

`componentRange` (integer vector) The vector must have 2 components. The first component specifies the starting index of *DataStruct* that the *DataStructAlias* points to. The second value is the upper limit that the *DataStructAlias* points to - *DataStructAlias* can access up to, but not included the index of the second component.

An example follows. Suppose we have the *DataStruct*

```
<DataStruct q>
  kind = nodalArray
  onGrid = domain
  numComponents = 9
</DataStruct>
```

with *DataStructAlias*

```
<DataStructAlias electronDensity>
  kind = nodalArray
  target = q
  componentRange = [3,5]
  writeOut = false
</DataStructAlias>
```

The *DataStructAlias* points to element 3, 4 of *DataStruct*.

`writeOut` (boolean) Tells USim whether to write out data from the *DataStructAlias* or not.

CHAPTER

SIX

UPDATESTEP

Defines an update step. An update step is a sequence of updaters with a possible synchronization that occurs at the end of the update steps. Synchronization is used in parallel runs for updating ghost cell values along the specified domains. An example *UpdateStep* code block is given below:

```
<UpdateStep bcStep>
  updaters = [bcLeft, bcRight, bcTop, bcBottomIon, bcBottomElectron, bcBottomEm]
  syncVars = [qnew]
</UpdateStep>
```

In this code block all the boundary condition updaters are called then "qnew" is synchronized across MPI barriers. The parameters for this *UpdateStep* have the following meanings:

`updaters` (string vector, required) Defines the list of updaters called in this update step. The updaters are called in the order they are presented in the list.

`syncVars` (string vector, optional) Defines a list of nodalArrays that are synchronized accross MPI boundaries at the completion of the update step.

`operation` (string, optional) Used in combination with *multiUpdater (1d, 2d, 3d)* and *implicitMultiUpdater (1d, 2d, 3d)*. Accepted values are *integrate* or *operate*. When *integrate* is used, integration is performed imediately after it is called. When *operate* is called an operation is performed on the newly integrated values (for each sub step of the runge-kutta method). If *operation* is not used then the updaters are simply evaluated.

CHAPTER

SEVEN

UPDATESEQUENCE

Update sequence takes a series of update steps and processes them in order. An example *UpdateSequence* is shown below:

```
<UpdateSequence sequence>
  startOnly = [initStep]
  restartOnly = [restoreStep]
  loop = [restrictions, hyperStep, correctionStep, bcStep, copyStep]
  writeOnly = [pressureStep]
</UpdateSequence>
```

The parameters for this *UpdateSequence* have the following meanings:

`startOnly` (string vector) Defines a list of *UpdateSteps* to apply only at the beginning of the simulation.

`restartOnly` (string vector) Defines a list of *UpdateSteps* to apply only in the restore phase of a restarted simulation.

`loop` (string vector) Defines a list of *UpdateSteps* that are continually looped over until the simulation completes.

`writeOnly` (string vector) Defines a list of UpdateSteps to apply only at data output time.

CHAPTER

EIGHT

UPDATER

Updaters are the fundamental computation infrastructure in USim. Given a set of input data structures, *in*, an Updater computes a set of output data structures *out* according to a set of rules defined by the *kind* of Updater. A simple updater based on a *combiner (1d, 2d, 3d)* Updater for computing the gas and magnetic pressure of a magnetohydrodynamic plasma is given below:

```
 <Updater pressCalc>
   kind = combiner1d

   onGrid = domain
# input array
   in = [q]

# ouput data-structures
   out = [pressure]

# labels for components in the input q array
   indVars_q = ["rho", "rhou", "rhov", "rhow", "Er","bx","by","bz", "psi"]

# Adiabatic index, or ratio of specific heats
   gasGamma = $GAS_GAMMA$
# Permeability of free space
   mu0 = $MU0$
   preExprs = ["pr = (gasGamma-1)*(Er - (0.5*(rhou^2+rhov^2+rhow^2)/rho)-\
                     (0.5/mu0)*(bx*bx+by*by+bz*bz))", \
               "pm = (0.5/mu0)*(bx*bx+by*by+bz*bz)" ]
   exprs = ["pr", "pm"]
 </Updater>
```

The following parameters are common to all *Updater* blocks:

`onGrid` (string, required) All updaters take a string that says which grid the updater is applied to. This grid corresponds to that which the input *nodalArray* is defined on

`kind` (string, required) All *updater* blocks take a string *kind* that species the type of updater block.

The following Updater *kind* attributes can be specified to perform simple operations (initialize, copy, transform) on *nodalArray* and *dynVector* data structures:

8.1 initialize (1d, 2d, 3d)

Initializes a *nodalArray* according to a user specified function block.

8.1.1 Data

`out` (string vector, required) Outputs 1 to N are *nodalArray*s which are initialized according to the function of this updater. If multiple *out*s are specified, then each *nodalArray* must have the same number components.

8.1.2 Parameters

There are no additional parameters for this kind of Updater

8.1.3 Sub-Blocks

`Function` (block, required) exprFunc for specifying the initial condition.

8.1.4 Example

The following code block specificies the initial condition for a magnetized shock tube for a two temperature plasma:

```
<Updater init>
  kind = initialize1d
  onGrid = domain
  out = [q]

  <Function func>
    kind = exprFunc

    pr = 1.0
    pl = 0.1

    rhor = 1.0
    rhol = 0.125
    mu0 = MU0
    gas_gamma = GAMMA

    preExprs = [ \
    "rho = if (x > 0.0, rhol, rhor)", \
    "P = if(x > 0.0, pl, pr)", \
    "bx = 0.75*sqrt(mu0)", \
    "by = if(x>0.0, -1.0*sqrt(mu0), 1.0*sqrt(mu0))", \
    "bz = 0.0",\
    "phi = 0.0",\
  ]

    exprs = ["rho", "0.0", "0.0", "0.0", \
             "P/(gas_gamma-1)+(0.5/mu0)*(bx*bx+by*by)",\
             "bx","by","bz","phi","0.5*P/(gas_gamma-1)"]
  </Function>
</Updater>
```

8.2 linearCombiner (1d, 2d, 3d)

Sum a user specified list of input *nodalArray*s, where each component can be scaled by a scalar factor and store the ouput in a *single* user-specifed *nodalArray*. All input and output data structures must have the same number of components.

The linearCombiner accepts the following parameters, in addition to those required by *Updater*:

8.2.1 Data

in (string vector, required) Input 1 to N are input *nodalArray*s to be summed. Each component can be scaled by a scalar factor.

out (string vector, required) Output is a *nodalArray* which will contain the sum of the *in*puts.

8.2.2 Parameters

coeffs (float vector, required) A vector of *n* scalars, such that $\mathrm{out} = \sum_{s=0}^{n-1} c_s \mathrm{in}_s$

8.2.3 Example

The following code block copies input *nodalArray* *qnew* to output *nodalArray* *q*:

```
<Updater copier>
  kind = linCombinerUpdater
  onGrid = domain

  in = [qnew]
  out = [q]

  coeffs = [1.0]
</Updater>
```

8.3 uniformCombiner (1d, 2d, 3d)

Performs an identical arithmetic operation on all components of a set of input *nodalArray*s and stores the ouput in a *single* user-specifed *nodalArray* using an expression evaluator. All input and output data structures must have the same number of components. The expression evaluator recognizes positions *x*, *y*, *z* and time *t*, along with the current timestep, *dt*, and the cell volume, *dVolume*. The expression evaluator checks the user supplied expression for validity and errors on finding undefined expressions.

The uniformCombiner accepts the following parameters, in addition to those required by *Updater*:

8.3.1 Data

in (string vector, required) Inputs 1 to N are input *nodalArray*s which will be supplied to the expression evaluator.

out (string vector, required) Output is a *nodalArray* which will contain the evaluation.

8.3.2 Parameters

indVars_inName (string vector, required) For each input variable an "indVars" string vector must be defined. So if *in = [q1,k2]* where *q1* and *electricField* are both *nodalArray* then the uniformCombiner block must define *indVars_q1 = ["q1"]* and *indVars_k2 = ["k2"]*. Note that the labels *"q1"* and *"k2"* are arbitrary; the requirement is that there is a single unique name for each input data structure, irrespective of the number of components of that data structure.

exprs (string vector, required) Strings must be put in quotes. The strings are evaluated and placed in the output array. Only one string can be supplied; this same expression is applied to all components of the input arrays uniformly. Available command are defined by the muParser (http://muparser.sourceforge.net/)

preExprs (string vector, optional) Strings must be put in quotes. The preExprs is used to compute quantities based on indVars that can later be used in the *exprs* to evaluate the output. Available commands are defined by the muParser (http://muparser.sourceforge.net)

other (strings, optional) In addition, an arbitrary number of constants can be defined that can then be used in evaluating expression in both *preExprs* and *exprs*.

8.3.3 Example

The code block below demonstrates the addition of two input *nodalArray* and placing the result into one single output *nodalArray*:

```
<Updater computeQ2>
  kind = uniformCombiner1d
  onGrid = domain

  in = [q1, k2]
  out = [q2]

  indVars_q1 = ["q1"]
  indVars_k2 = ["k2"]

  exprs = ["q1+dt*k2"]
</Updater>
```

8.4 combiner (1d, 2d, 3d)

Performs arithmetic operations on a set of input *nodalArray*s and stores the ouput in a *single* user-specifed *nodalArray* using an expression evaluator. The expression evaluator recognizes positions *x*, *y*, *z* and time *t*, along with the current *dt* and the cell volume, *dVolume*. The expression evaluator checks the user supplied expression for validity and errors on finding undefined expressions.

The combiner accepts the following parameters, in addition to those required by *Updater*:

8.4.1 Data

in (string vector, required) Input 1 to N are input *nodalArray*s which will be supplied to the expression evaluator.

dynVectors (string vector) Input 1 to N are input *dynVector*s which will be supplied to the expression evaluator.

out (string vector, required) Output is a *nodalArray* which will contain the output of the evaluation.

8.4.2 Parameters

`indVars_inName` (string vector, required) For each input variable an "indVars" string vector must be defined. So if *in = [magneticField, electricField]* where *magneticField* and *electricField* are each 3-component *nodalArray*s then the combiner block must define *indVars_magneticField = ["bx", "by", "bz"]* and *indVars_electricField = ["ex", "ey", "ez"]*. Note that the labels *"bx", "by", "bz"* and *"ex", "ey", "ez"* are arbitrary; the requirement is that there is a unique name for each component of each input data structure.

`dynVars_` (string vector) For each dynVector variable a "dynVars" string vector must be defined. So if *dynVectors = [a, b]* where *a* and *b* are each 3-component *dynVector*s then the combine block must define *dynVectorVars_a = ["a1","a2","a3"]* and *dynVectorVars_b = ["b1","b2","b3"]*. Note that the labels *"a1","a2","a3"* and *"b1","b2","b3"* are arbitrary; the requirement is that there is a unique name for each component of each input data structure.

`exprs` (string vector, required) Strings must be put in quotes. The strings are evaluated and placed in the output array. The number of strings must be identical to the number of components in the output array. Available command are defined by the muParser (http://muparser.sourceforge.net/)

`preExprs` (string vector, optional) Strings must be put in quotes. The preExprs is used to compute quantities based on indVars that can later be used in the *exprs* to evaluate the output. Available commands are defined by the muParser (http://muparser.sourceforge.net)

`other` (strings, optional) In addition, an arbitrary number of constants can be defined that can then be used in evaluating expression in both *preExprs* and *exprs*.

8.4.3 Example

The following code block demonstrates the use of a combiner to compute the cross product of two 3-component *nodalArray*s (*magneticField* and *electricField*) using *preExprs*, and store the output multiplied by a user-specifed constant (*factor*) in a 3-component *nodalArray* (*eCrossB*)

```
<Updater computeECrossB>
  kind = combiner2d
  onGrid = domain

  in = [magneticField, electricField]
  out = [eCrossB]

  indVars_magneticField = ["bx", "by", "bz"]
  indVars_electricField = ["ex", "ey", "ez"]

  factor = 10.0

  preExprs = ["Sx = (ey*bz-ez*by)", "Sy = (ez*bx-ex*bz)", "Sz =
  (ex*by-ey*bx)"]

  exprs = ["factor*Sx", "factor*Sy", "factor*Sz"]
</Updater>
```

8.5 dynVectorOperator

Performs arithmetic operations on a set of input *dynVector*s and stores the ouput in a *single* user-specifed *dynVector* using an expression evaluator. The expression evaluator recognizes time *t*, along with the current timestep

dt. The expression evaluator checks the user supplied expression for validity and errors on finding undefined expressions.

The dynVectorOperator accepts the following parameters, in addition to those required by *Updater*:

8.5.1 Data

in (string vector, required) Input 1 to N are input *dynVector*s which will be supplied to the expression evaluator.

out (string vector, required) Output is a *dynVector* which will contain the evaluation.

8.5.2 Parameters

indVars_inName (string vector, required) For each input variable an *indVars* string vector must be defined. So if *in = [charge, current]* where *charge* and *current* are each 1-component *dynVector*s then the dynVectorOperator block must define *indVars_charge = ["Q"]* and *indVars_current = ["I"]*. Note that the labels *"Q"* and *"I"* are arbitrary; the requirement is that there is a unique name for each component of each input data structure.

exprs (string vector, required) Strings must be put in quotes. The strings are evaluated and placed in the output array. The number of strings must be identical to the number of components in the output array. Available command are defined by the muParser (http://muparser.sourceforge.net/)

preExprs (string vector, optional) Strings must be put in quotes. The preExprs is used to compute quantities based on indVars that can later be used in the *exprs* to evaluate the output. Available commands are defined by the muParser (http://muparser.sourceforge.net)

other (strings, optional) In addition, an arbitrary number of constants can be defined that can then be used in evaluating expression in both *preExprs* and *exprs*.

8.5.3 Example

The following code block demonstrates the use of a dynVectorOperator to compute update the current associated with the flow of charge. Two input *dynVector*s (*charge* and *current*) are combined in an exprs and the output is stored in a *dynVector* (*current*). Note that a *dynVector* can be both an input data structure *and* and output data structure for this updater.

```
<Updater integrateCurrent>
  kind = dynVectorOperator
  in = [charge, current]
  out = [current]

  indVars_charge = ["Q"]
  indVars_current = ["I"]
  C = CAPACITANCE
  bgL = L0

  exprs = ["I-dt*(1.0/bgL)*(Q/C)"]
</Updater>
```

The following Updater *kind* attributes can be specified to perform more advanced operations based on predefined USim capabilities:

8.6 equation (1d, 2d, 3d)

The equation updater is used to update a *Algebraic Equations* and evaluates a set of input *nodalArray*s and stores the ouput in one or more user-specified *nodalArray*s. The number of inputs and outputs are defined by the kind of *Algebraic Equations* being used for the **Equation**.

The equation updater accepts the following parameters, in addition to those required by *Updater*:

8.6.1 Data

`in` (string vector, required) Inputs 1 to N are *nodalArray*s which will be supplied to the source through the **Equation** block.

`out` (string vector, required) Outputs 1 to N are *nodalArray*s which will contain the output of the Source.

8.6.2 Parameters

There are no additional parameters for this kind of Updater.

8.6.3 Sub-Blocks

`Equation` Defines the kind of source being solved. Equation in this case is actually a *kind* of *Algebraic Equations*. If multiple <Equation> blocks are defined then the results are added together to produce the output.

8.6.4 Example

The following code block demonstrates the usage of the equation updater combined with the *bremsPowerSrc* source

```
<Updater radiationSourceUpdater>
   kind = equation1d
   onGrid = domain
   in =  [elecNumDensity, temperature, zeffective]
   out = [radiationPower]

  <Equation Bremsstrahlung>
    kind = bremsPowerSrc
  </Equation>
</Updater>
```

8.7 computePrimitiveState(1d, 2d, 3d)

The *computePrimitiveState* updater computes a vector of primitive variables, $\mathbf{w} = \mathbf{w}(\mathbf{q})$, (e.g. density, velocity, pressure), given a vector of conserved variables $\mathbf{q}$ (e.g. density, momentum, total energy) according to relationship specified by a *Hyperbolic Equations*.

The *computePrimitiveState* updater accepts the parameters below, in addition to those required by *Updater*.

8.7.1 Data

in (string vector, required) Input 1 to N are input *nodalArray*s which will be supplied to the equation. Defined by the choice of *Hyperbolic Equations*.

out (string vector, required) Output is a *nodalArray* which will contain $\mathbf{w}(\mathbf{q})$. The number of components is defined by the choice of *Hyperbolic Equations*.

8.7.2 Sub-Blocks

Equation (block, required) The *Hyperbolic Equations* that defines $\mathbf{q}$, $\mathcal{F}(\mathbf{w})$, $\mathbf{w} = \mathbf{w}(\mathbf{q})$, along with the eigensystem associated with $\mathcal{F}(\mathbf{w})$.

8.7.3 Example

The following block demonstrates the *twoTemperatureMhdDednerEqn* used in combination with *computePrimitiveState(1d, 2d, 3d)* to compute $\mathbf{w}(\mathbf{q})$

```
 <Updater computePrimitiveState>
   kind = computePrimitiveState1d

   onGrid = domain
# input data-structures
   in = [q,electricField,current,chargeState,resistivity]

# ouput data-structures
   out = [w]

   <Equation mhd>
     kind = twoTemperatureMhdDednerEqn
     gasGamma = GAS_GAMMA
     electronGamma = $ELECTRON_GAMMA$
     basementDensity = $BASEMENT_DENSITY$
     basementPressure = $BASEMENT_PRESSURE$
     externalEfield = "electricField"
     currentVector = "current"
   </Equation>

 </Updater>
```

8.8 vertexJetUpdater (1d, 2d, 3d)

Initializes a fluid jet based on a tip location and the vector from a center point. Can be used to initialize multiple jets based on ideal gas laws or general equation of state using Propaceos tables. This updater was designed for simulating plasma jet merging experiments.

8.8.1 Data

out (string vector, required) The *nodalArray* in which to store the initialized fluid jet variables. If multiple *out* *nodalArray* are specified, then each *nodalArray* must have the same number components; each *out* data structure will be initialized according to this updater.

8.8.2 Parameters

`width` (float) The width of the initialization region. The jet is initialized in a region with width *width* which is perpendicular to the direction from the jet *vertex* to the *origin*.

`length` (float) The length of the initialization region. The jet is initialized in a region from the vertex to a distance *length* from the tip of the vertex away from the origin.

`model` (string) The equation system that should be used to model the jet. Model should be a fluid system such as eulerEqnEqn, idealMhdEqn, twoTemperatureMhdEqn, twoTemperatureMhdEosEqn. USim will request that values that need to be initialized in the individual models be initialized in this updater as well.

`origin` (float vector) The location where the jet points.

`radialVelocity` (float) Required only if the velocityFunction block is not specified. Provides a bulk velocity for the jet.

`numberDensity` (float) Characteristic number density of the fluid. If the normalizedDensityFunction has a peak value of 1 then the peak value of the number density of the jet will be *numberDensity*. Required if speciesMass is specified. This term is also required if a propaceos *filename* is specified.

`speciesMass` (float) Mass of the atomic species (in Kg). Required if *numberDensity* is specified. The parameter is also required if a propaceos *filename* is specified.

`density` (float) If neither *numberDensity* or *speciesMass* is specified then USim expects the characteristic mass density *density* to be specified.

`pressure` (float) The pressure of the jet (constant throughout the jet). Either the *pressure* or *temperature* for the jet must be defined. USim checks if the *pressure* is set first and if it is it uses that for initialization, otherwise it checks for temperature.

`temperature` (float) The temperature of the jet. Either the *pressure* or the *temperature* for the jet must be defined. USim checks if the *pressure* is set first and if it is it uses that for initialization, otherwise it checks for temperature.

`vertex_n` (float vector) vertex_n defines the tip of the nth jet. n must be a number from 0 to the number of jets in the simulation. If you skip a number the subsequent jets will be ignored. Each jet has the exact same properties, but different orientations. The jet points towards the origin.

`xAxis` (float vector) Allows the user to define an alternative xAxis. The coordinates for the jet vertex will be rotated to the new axis. Magnitude of the vector does not matter.

`yAxis` (float vector) If the user defines an *xAxis* they may also define a yAxis otherwise USim picks its own y Axis. The yAxis should be chosen perpendicular to the xAxis. The Magnitude of the vector does not matter.

`filename` (string) Name of the propaceos file to use if a general equation of state is required.

`includeElectronTemperature` (boolean) If *filename* is specified then you may or may not need to have the electron energy initialized by USim, depending on the model being used. A two-temperature model will need to set the electron energy along with the bulk energy.

`electronTemperatureIndex` (int) If the *includeElectronTemperature* is true then the user will need to specify the index in the in vector for the electron energy equation so USim knows where to initialize the electron energy.

8.8.3 Sub-Blocks

`normalizedDensityFunction` (block) The desired plasma jet density is scaled by the normalized density function. As a result the normalized density function should have values that range from 0 to 1. The

density function is a function of x,y,z where x is measured from the tip of the jet along its axis. y and z are perpendicular to this direction.

velocityFunction (block) The velocity function specifies the velocity of the jet as a function of space x,y,z. The density function is a function of x,y,z where x is measured from the tip of the jet along its axis. y and z are perpendicular to this direction.

8.8.4 Example

```
 <Updater jetSet>
  kind = vertexJetUpdater3d

  origin = [0.0, 0.0, 0.0]
  width = 0.05
  length = 0.5

  radialVelocity = $-U$

  numberDensity = $RHO_JET/MI$
  speciesMass = MI
  temperature = TKELVIN

  <normalizedDensityFunction>
       kind = exprFunc
       preExprs = ["R=1.1e-2","zd=4.5e-2","r=sqrt(y*y+z*z)","alpha=3.8",\
               "G=exp(-(r*r/(R*R)))","kappa=(x/(alpha*zd))^alpha"]
       exprs = ["max(1.0*G*kappa*exp(alpha-(x/zd)),1.0e-6)"]
  </normalizedDensityFunction>

  vertex0 = [0.25, 0.0, 0.0]
  vertex1 = [0.0, 0.25, 0.0]
  vertex2 = [0.0, 0.0, 0.25]

  model = idealMhdEosEqn
  mu0 = MU0

  onGrid = domain
  filename = Ar_Ni_1e^10_10group_NLTE_20110427.prp

  out = [q]
</Updater>
```

The following Updater *kind* attributes can be specified to compute finite volume discretizations of a range of vector calculus operators:

8.9 firstOrderMusclUpdater (1d, 2d, 3d)

The firstOrderMusclUpdater computes an first order upwind discretization of the spatial component of a non-linear hyperbolic system, possibly with source terms:

$$\nabla \cdot [\mathcal{F}(\mathbf{w})] - \mathcal{S}(\mathbf{w})$$

where $\mathbf{q}$ is a vector of conserved variables (e.g. density, momentum, total energy), $\mathcal{F}(\mathbf{w})$ is a non-linear flux tensor computed from a vector of primitive variables, (e.g. density, velocity, pressure), $\mathbf{w} = \mathbf{w}(\mathbf{q})$ and $\mathcal{S}(\mathbf{w})$ is some source term.

The *firstOrderMuscl* updater accepts the parameters below, in addition to those required by *Updater*.

8.9.1 Data

in (string vector, required) Input 1 to N are input *nodalArray*s which will be supplied to the equation. Defined by the choice of *Hyperbolic Equations*.

out (string vector, required) Output is a *nodalArray* which will contain $\nabla \cdot [\mathcal{F}(\mathbf{w})] - \mathcal{S}(\mathbf{w})$. The number of components is defined by the choice of *Hyperbolic Equations*.

waveSpeeds (string vector, optional) Defines the *dynVector* containing the fastest wave speeds in the mesh required by some equation systems (e.g. *mhdDednerEqn*).

8.9.2 Parameters

equations (string vector, required) List of equation systems to solve. Accepts at most one equation

numericalFlux (string, required) Defines the numerical flux need to compute an upwind approximation to the non-linear flux $\mathcal{F}(\mathbf{w})$

cfl (float, optional) Defines the CFL condition for the finite volume scheme. The updater returns an error code if this condition is violated during a timestep. Defaults to $(\#\text{ of dimensions})^{-1}$.

checkCfl (bool, optional) Whether to check the CFL condition during an updater, defaults to true. Should be set to false if combined with *implicitMultiUpdater (1d, 2d, 3d)*.

sources (string vector, optional) List of sources to apply. Each source listed here must be associated with a *Source* block (see below).

8.9.3 Sub-Blocks

Equation (block, required) The *Hyperbolic Equations* that defines $\mathbf{q}$, $\mathcal{F}(\mathbf{w})$, $\mathbf{w} = \mathbf{w}(\mathbf{q})$, along with the eigensystem associated with $\mathcal{F}(\mathbf{w})$.

Source (block) Adds a *Algebraic Equations* to the hyperbolic equation system.

8.9.4 Example

The following block demonstrates the *firstOrderMuscl* updater used in combination with the *mhdDednerEqn* to compute $\nabla \cdot \mathcal{F}(\mathbf{w})$ with an externally supplied magnetic field:

```
<Updater hyper>
  kind=firstOrderMuscl1d
  onGrid=domain

  # input nodal component arrays
  in=[q   backgroundB]

  # output nodal component array
  out=[qnew]

  # input dynVector containing fastest wave speed
  waveSpeeds=[waveSpeed]

  # the numerical flux to use
```

```
    numericalFlux= hlldFlux

    # CFL number to use
    cfl=0.3

    # list of equations to solve
    equations=[mhd]

    <Equation mhd>
      kind=mhdDednerEqn
      gasGamma=1.4
      externalBfield="backgroundB"
    </Equation>

  </Updater>
```

8.10 classicMusclUpdater (1d, 2d, 3d)

The classicMusclUpdater computes an second order upwind discretization (that is suitable for use on good quality tetrahedral and hexahedral meshes) of the spatial component of a non-linear hyperbolic system, possibly with source terms:

$$\nabla \cdot [\mathcal{F}(\mathbf{w})] - \mathcal{S}(\mathbf{w})$$

where $\mathbf{q}$ is a vector of conserved variables (e.g. density, momentum, total energy), $\mathcal{F}(\mathbf{w})$ is a non-linear flux tensor computed from a vector of primitive variables, (e.g. density, velocity, pressure), $\mathbf{w} = \mathbf{w}(\mathbf{q})$ and $\mathcal{S}(\mathbf{w})$ is some source term.

The *classicMuscl* updater accepts the parameters below, in addition to those required by *Updater*.

8.10.1 Data

in (string vector, required) Input 1 to N are input *nodalArray*s which will be supplied to the equation. Defined by the choice of *Hyperbolic Equations*.

out (string vector, required) Output is a *nodalArray* which will contain $\nabla \cdot [\mathcal{F}(\mathbf{w})] - \mathcal{S}(\mathbf{w})$. The number of components is defined by the choice of *Hyperbolic Equations*.

waveSpeeds (string vector, optional) Defines the *dynVector* containing the fastest wave speeds in the mesh required by some equation systems (e.g. *mhdDednerEqn*).

8.10.2 Parameters

equations (string vector, required) List of equation systems to solve. Accepts at most one equation

numericalFlux (string, required) Defines the numerical flux need to compute an upwind approximation to the non-linear flux $\mathcal{F}(\mathbf{w})$

limiter (string vector, required) Defines the limiter to be applied to the input variables; one entry required per input variable.

variableForm (string, required) Whether the reconstruction will occur in *primitive* or *conservative* variables. All systems can be reconstructed in *conservative* form. A number of fluid systems can be also be solved in *primitive* form.

`preservePositivity` (boolean, optional) A number of equation systems can produce negative densities or pressures. The preservePositivity option checks whether the reconstructed values produce positive values for pressure and density. If they do not then it drops the order of reconstruction to first order.

`cfl` (float, optional) Defines the CFL condition for the finite volume scheme. The updater returns an error code if this condition is violated during a timestep. Defaults to $(\#\text{ of dimensions})^{-1}$.

`checkCfl` (bool, optional) Whether to check the CFL condition during an updater, defaults to true. Should be set to false if combined with *implicitMultiUpdater (1d, 2d, 3d)*.

`sources` (string vector, optional) List of sources to apply. Each source listed here must be associated with a *Source* block (see below).

8.10.3 Sub-Blocks

`Equation` (block, required) The *Hyperbolic Equations* that defines $\mathbf{q}$, $\mathcal{F}(\mathbf{w})$, $\mathbf{w} = \mathbf{w}(\mathbf{q})$, along with the eigensystem associated with $\mathcal{F}(\mathbf{w})$.

`Source` (block) Adds a *Algebraic Equations* to the hyperbolic equation system.

8.10.4 Example

The following block demonstrates the *classicMuscl* updater used in combination with the *mhdDednerEqn* to compute $\nabla \cdot \mathcal{F}(\mathbf{w})$ with an externally supplied magnetic field:

```
<Updater hyper>
  kind=classicMuscl1d
  onGrid=domain

  # input nodal component arrays
  in=[q   backgroundB]

  # output nodal component array
  out=[qnew]

  # input dynVector containing fastest wave speed
  waveSpeeds=[waveSpeed]

  # the numerical flux to use
  numericalFlux= hlldFlux

  # CFL number to use
  cfl=0.3
  # Form of variables to limit
  variableForm= primitive

  # Limiter; one per input nodal component array
  limiter=[minmod   minmod]

  # list of equations to solve
  equations=[mhd]

  <Equation mhd>
    kind=mhdDednerEqn
    gasGamma=1.4
    externalBfield="backgroundB"
  </Equation>
```

```
</Updater>
```

8.11 unstructMusclUpdater (1d, 2d, 3d)

The *unstructMusclUpdater* computes an second order upwind discretization (that is suitable for use on general unstructured tetrahedral and hexahedral meshes) of the spatial component of a non-linear hyperbolic system, possibly with source terms:

$$\nabla \cdot [\mathcal{F}(\mathbf{w})] - \mathcal{S}(\mathbf{w})$$

where $\mathbf{q}$ is a vector of conserved variables (e.g. density, momentum, total energy), $\mathcal{F}(\mathbf{w})$ is a non-linear flux tensor computed from a vector of primitive variables, (e.g. density, velocity, pressure), $\mathbf{w} = \mathbf{w}(\mathbf{q})$ and $\mathcal{S}(\mathbf{w})$ is some source term.

The *unstructMuscl* updater accepts the parameters below, in addition to those required by *Updater*.

8.11.1 Data

`in` (string vector, required) Input 1 to N are input *nodalArray*s which will be supplied to the equation. Defined by the choice of *Hyperbolic Equations*.

`out` (string vector, required) Output is a *nodalArray* which will contain $\nabla \cdot [\mathcal{F}(\mathbf{w})] - \mathcal{S}(\mathbf{w})$. The number of components is defined by the choice of *Hyperbolic Equations*.

`waveSpeeds` (string vector, optional) Defines the *dynVector* containing the fastest wave speeds in the mesh required by some equation systems (e.g. *mhdDednerEqn*).

8.11.2 Parameters

`equations` (string vector, required) List of equation systems to solve. Accepts at most one equation

`numericalFlux` (string, required) Defines the numerical flux need to compute an upwind approximation to the non-linear flux $\mathcal{F}(\mathbf{w})$

`limiter` (string vector, required) Defines the limiter to be applied to the input variables; one entry required per input variable.

`variableForm` (string, required) Whether the reconstruction will occur in *primitive* or *conservative* variables. All systems can be reconstructed in *conservative* form. A number of fluid systems can be also be solved in *primitive* form.

`preservePositivity` (boolean, optional) A number of equation systems can produce negative densities or pressures. The preservePositivity option checks whether the reconstructed values produce positive values for pressure and density. If they do not then it drops the order of reconstruction to first order.

`numberOfInterpolationPoints` (integer, required) Number of points to be considerd for the least squares fit. This parameter varies from mesh to mesh and should be determined by computing a known function on the mesh.

The numberOfInterpolationPoints must be greater than (or equal to) the number of coefficients in the polynomial approximation. This means that in 1d the value is 4, in 2D the value is at least 6 and in 3D the value is at least 10.

These choices do not guarantee that a matrix inverse will be found. The following values though appear to be adequate in general: in 1D 4; in 2D 8 and in 3D 20.

orderAccuracy (integer, option) Order of the polynomial that is used to form the operator. Choice of 1, 2 or 3 corresponding, respectively to first, second and third order accuracy. The appropriate choice of order varies on the problem type and the mesh used. Defaults to 2.

formulation (string, optional) Whether to use a reconstruction based on *constant* or *spline* interpolation. Defaults to *constant*.

If *formulation* = *"spline"*, then the following options can be specified:

`leastSquaresBasis` (string, optional) The spline basis to use for the least squares problem. Options are: *wendland*, *wu* and *bumann*. Defaults to *buhmann*.

`leastSquaresBasisOrder` (string, optional) Order of polynomial to use for the least squares basis. Can accept up to 6th order polynomials, dependent on the choice of spline basis.

cfl (float, optional) Defines the CFL condition for the finite volume scheme. The updater returns an error code if this condition is violated during a timestep. Defaults to $(\#\text{ of dimensions})^{-1}$.

checkCfl (bool, optional) Whether to check the CFL condition during an updater, defaults to true. Should be set to false if combined with *implicitMultiUpdater (1d, 2d, 3d)*.

sources (string vector, optional) List of sources to apply. Each source listed here must be associated with a *Source* block (see below).

8.11.3 Sub-Blocks

Equation (block, required) The *Hyperbolic Equations* that defines $\mathbf{q}$, $\mathcal{F}(\mathbf{w})$, $\mathbf{w} = \mathbf{w}(\mathbf{q})$, along with the eigensystem associated with $\mathcal{F}(\mathbf{w})$.

Source (block) Adds a *Algebraic Equations* to the hyperbolic equation system.

8.11.4 Example

The following block demonstrates the *classicMuscl* updater used in combination with the *mhdDednerEqn* to compute $\nabla \cdot \mathcal{F}(\mathbf{w})$ with an externally supplied magnetic field:

```
<Updater hyper>
  kind=classicMuscl1d
  onGrid=domain

  # input nodal component arrays
  in=[q   backgroundB]

  # output nodal component array
  out=[qnew]

  # input dynVector containing fastest wave speed
  waveSpeeds=[waveSpeed]

  # the numerical flux to use
  numericalFlux= hlldFlux

  # CFL number to use
  cfl=0.3
  # Form of variables to limit
  variableForm= primitive
```

```
    # Limiter; one per input nodal component array
    limiter=[minmod   minmod]

    # list of equations to solve
    equations=[mhd]

    <Equation mhd>
      kind=mhdDednerEqn
      gasGamma=1.4
      externalBfield="backgroundB"
    </Equation>

  </Updater>
```

8.12 thirdOrderMusclUpdater (1d, 2d, 3d)

The *thirdOrderMusclUpdater* uses third order accurate spatial reconstruction that is suitable for use on general unstructured tetrahedral and hexahedral meshes to compute an upwind discretization of the spatial component of a non-linear hyperbolic system, possibly with source terms:

$$\nabla \cdot [\mathcal{F}(\mathbf{w})] - \mathcal{S}(\mathbf{w})$$

where $\mathbf{q}$ is a vector of conserved variables (e.g. density, momentum, total energy), $\mathcal{F}(\mathbf{w})$ is a non-linear flux tensor computed from a vector of primitive variables, (e.g. density, velocity, pressure), $\mathbf{w} = \mathbf{w}(\mathbf{q})$ and $\mathcal{S}(\mathbf{w})$ is some source term.

The *thirdOrderMuscl* updater accepts the parameters below, in addition to those required by *Updater*.

8.12.1 Data

`in` (string vector, required) Input 1 to N are input *nodalArray*s which will be supplied to the equation. Defined by the choice of *Hyperbolic Equations*.

`out` (string vector, required) Output is a *nodalArray* which will contain $\nabla \cdot [\mathcal{F}(\mathbf{w})] - \mathcal{S}(\mathbf{w})$. The number of components is defined by the choice of *Hyperbolic Equations*.

`waveSpeeds` (string vector, optional) Defines the *dynVector* containing the fastest wave speeds in the mesh required by some equation systems (e.g. *mhdDednerEqn*).

8.12.2 Parameters

`equations` (string vector, required) List of equation systems to solve. Accepts at most one equation

`numericalFlux` (string, required) Defines the numerical flux need to compute an upwind approximation to the non-linear flux $\mathcal{F}(\mathbf{w})$

`limiter` (string vector, required) Defines the limiter to be applied to the input variables; one entry required per input variable.

`variableForm` (string, required) Whether the reconstruction will occur in *primitive* or *conservative* variables. All systems can be reconstructed in *conservative* form. A number of fluid systems can be also be solved in *primitive* form.

`preservePositivity` (boolean, optional) A number of equation systems can produce negative densities or pressures. The preservePositivity option checks whether the reconstructed values produce positive values for pressure and density. If they do not then it drops the order of reconstruction to first order.

`numberOfInterpolationPoints` (integer, required) Number of points to be considerd for the least squares fit. This parameter varies from mesh to mesh and should be determined by computing a known function on the mesh.

The numberOfInterpolationPoints must be greater than (or equal to) the number of coefficients in the polynomial approximation. This means that in 1d the value is 4, in 2D the value is at least 6 and in 3D the value is at least 10.

These choices do not guarantee that a matrix inverse will be found. The following values though appear to be adequate in general: in 1D 4; in 2D 8 and in 3D 20.

`orderAccuracy` (integer, option) Order of the polynomial that is used to form the operator. Choice of 1, 2 or 3 corresponding, respectively to first, second and third order accuracy. The appropriate choice of order varies on the problem type and the mesh used. Defaults to 2.

`formulation` (string, optional) Whether to use a reconstruction based on *constant* or *spline* interpolation. Defaults to *constant*.

If *formulation = "spline"*, then the following options can be specified:

`leastSquaresBasis` (string, optional) The spline basis to use for the least squares problem. Options are: *wendland*, *wu* and *bumann*. Defaults to *buhmann*.

`leastSquaresBasisOrder` (string, optional) Order of polynomial to use for the least squares basis. Can accept up to 6th order polynomials, dependent on the choice of spline basis.

`cfl` (float, optional) Defines the CFL condition for the finite volume scheme. The updater returns an error code if this condition is violated during a timestep. Defaults to $(\#\text{ of dimensions})^{-1}$.

`checkCfl` (bool, optional) Whether to check the CFL condition during an updater, defaults to true. Should be set to false if combined with *implicitMultiUpdater (1d, 2d, 3d)*.

`sources` (string vector, optional) List of sources to apply. Each source listed here must be associated with a *Source* block (see below).

8.12.3 Sub-Blocks

`Equation` (block, required) The *Hyperbolic Equations* that defines $\mathbf{q}$, $\mathcal{F}(\mathbf{w})$, $\mathbf{w} = \mathbf{w}(\mathbf{q})$, along with the eigensystem associated with $\mathcal{F}(\mathbf{w})$.

`Source` (block) Adds a *Algebraic Equations* to the hyperbolic equation system.

8.12.4 Example

The following block demonstrates the *classicMuscl* updater used in combination with the *mhdDednerEqn* to compute $\nabla \cdot \mathcal{F}(\mathbf{w})$ with an externally supplied magnetic field:

```
<Updater hyper>
  kind=classicMuscl1d
  onGrid=domain

  # input nodal component arrays
  in=[q   backgroundB]
```

```
    # output nodal component array
    out=[qnew]

    # input dynVector containing fastest wave speed
    waveSpeeds=[waveSpeed]

    # the numerical flux to use
    numericalFlux= hlldFlux

    # CFL number to use
    cfl=0.3
    # Form of variables to limit
    variableForm= primitive

    # Limiter; one per input nodal component array
    limiter=[minmod   minmod]

    # list of equations to solve
    equations=[mhd]

    <Equation mhd>
      kind=mhdDednerEqn
      gasGamma=1.4
      externalBfield="backgroundB"
    </Equation>

  </Updater>
```

8.13 vector (1d, 2d, 3d)

The *vector* updater computes a range of first derivatives of quanties defined on the USim computational mesh using a least squares gradient method. This updater differs from the *firstOrderMusclUpdater (1d, 2d, 3d)*, *classicMusclUpdater (1d, 2d, 3d)*, *unstructMusclUpdater (1d, 2d, 3d)* and *thirdOrderMusclUpdater (1d, 2d, 3d)* updaters in that no upwinding is performed here. As such, the *vector* updater is only suitable for problems that do not require upwind stabilization.

The *vector* updater accepts the parameters below, in addition to those required by *Updater*.

8.13.1 Data

`in` (string vector, required) Defined by the choice of the derivative attribute, as detailed below.

`out` (string vector, required) Defined by the choice of the derivative attribute, as detailed below.

8.13.2 Parameters

`orderAccuracy` (integer, required) Order of the polynomial that is used to form the operator. Choice of 1, 2 or 3 corresponding, respectively to first, second and third order accuracy. The appropriate choice of order varies on the problem type and the mesh used.

`numberOfInterpolationPoints` (integer, required) Number of points to be considerd for the least squares fit. This parameter varies from mesh to mesh and should be determined by computing a known function on the mesh.

The numberOfInterpolationPoints must be greater than (or equal to) the number of coefficients in the polynomial approximation. This means that in 1d the value is 4, in 2D the value is at least 6 and in 3D the value is at least 10.

These choices do not guarantee that a matrix inverse will be found. The following values though appear to be adequate in general: in 1D 4; in 2D 8 and in 3D 20.

`coefficient` (float, required) Constant floating point value, c that multiplies the output of the diffusion updater.

`derivative` (string, required) The type of derivative that will be performed. Available derivatives are:

`gradient` (*derivative = gradient*) Computes $c\nabla\phi$ where ϕ is a scalar quantity defined on the grid and c is a constant coefficient. When *derivative = gradient* is specified, the following input and output variables should be specified:

`in` ϕ (*nodalArray*, 1-component, required): scalar quantity to compute the gradient of.

`out` $c\nabla\phi$ (*nodalArray*, 3-components, required): gradient of ϕ

`curl` (*derivative = curl*) Computes $c\nabla \times \mathbf{v}$ where $\mathbf{v}$ is a vector quantity defined on the grid and c is a constant coefficient. When *derivative = curl* is specified, the following input and output variables should be specified:

`in` $\mathbf{v}$ (*nodalArray*, 3-components, required): vector quantity to compute the curl of.

`out` $c\nabla \times \mathbf{v}$ (*nodalArray*, 3-components, required): curl of $\mathbf{v}$

`divergence` (*derivative = divergence*) Computes $c\nabla \cdot \mathbf{v}$ where $\mathbf{v}$ is a vector quantity defined on the grid and c is a constant coefficient. When *derivative = divergence* is specified, the following input and output variables should be specified:

`in` $\mathbf{v}$ (*nodalArray*, 3-components, required); vector quantity to compute the divergence of.

`out` $c\nabla \cdot \mathbf{v}$ (*nodalArray*, 3-components, required); divergence of $\mathbf{v}$

8.13.3 Example

The following code block demonstrates the least squares gradient operator for computing the gradient of a scalar quantity:

```
<Updater derivative>
   kind = vector2d
   derivative = gradient
   coefficient = 1.0
   numberOfInterpolationPoints = 8
   orderAccuracy = 2
   onGrid = domain
   in = [phi]
   out = [gradPhi]
 </Updater>
```

The following code block demonstrates the least squares curl operator for computing the curl of a vector quantity:

```
<Updater copier>
  kind = vector2d
  onGrid = domain
  derivative = curl
  coefficient = 1.0
  orderAccuracy = 1
```

```
  numberOfInterpolationPoints = 5
  in = [q]
  out = [qnew]
</Updater>
```

The following code block demonstrates the least squares divergence operator for computing the divergence of a vector quantity:

```
<Updater copier>
  kind = vector2d
  onGrid = domain
  derivative = divergence
  orderAccuracy = 2
  coefficient = 1.0
  numberOfInterpolationPoints = 8  # 7 for 1st degree polynomial
  in = [q]
  out = [qnew]
</Updater>
```

8.14 diffusion (1d, 2d, 3d)

The *diffusion* updater computes a range of second derivatives of quanties defined on the USim computational mesh using a least squares gradient method. The specific form of the operator can be chosen by *derivative* to the desired type, as detailed below.

The *diffusion* updater accepts the parameters below, in addition to those required by *Updater*.

8.14.1 Data

`in` (string vector, required) Defined by the choice of the derivative attribute, as detailed below.

`out` (string vector, required) Defined by the choice of the derivative attribute, as detailed below.

8.14.2 Parameters

`orderAccuracy` (integer, required) Order of the polynomial that is used to form the operator. Choice of 1, 2 or 3 corresponding, respectively to first, second and third order accuracy. The appropriate choice of order varies on the problem type and the mesh used.

`numberOfInterpolationPoints` (integer, required) Number of points to be considerd for the least squares fit. This parameter varies from mesh to mesh and should be determined by computing a known function on the mesh.

The numberOfInterpolationPoints must be greater than (or equal to) the number of coefficients in the polynomial approximation. This means that in 1d the value is 4, in 2D the value is at least 6 and in 3D the value is at least 10.

These choices do not guarantee that a matrix inverse will be found. The following values though appear to be adequate in general: in 1D 4; in 2D 8 and in 3D 20.

`coefficient` (float, required) Constant floating point value, c that multiplies the output of the diffusion updater.

`derivative` (string, required) The type of derivative that will be performed. Available derivatives are:

diffusion (*derivative = diffusion*) Computes $c\nabla \cdot (\kappa \nabla \phi)$ where ϕ and κ are scalar quantities defined on the grid and c is a constant coefficient. When *derivative = diffusion* is specified, the following input and output variables should be specified:

in

1. ϕ (*nodalArray*, n-components, required); scalar quantity to compute the Laplacian. If $n > 1$ then the Laplacian of each component is computed independently.
2. κ (*nodalArray*, n-components, required); scalar diffusion coefficient. *n* must be the same as for ϕ

out

1. $c\nabla \cdot (\kappa \nabla \phi)$ (*nodalArray*, n-components, required); Laplacian of ϕ. *n* must be the same as for ϕ

anisotropicDiffusion (*derivative = anisotropicDiffusion*) Computes $c\nabla \cdot (\mathcal{K} \nabla \phi)$ where ϕ is a scalar quantity and $\mathcal{K}$ is a tensor quantity defined on the grid and c is a constant coefficient. When *derivative = anisotropicDiffusion* is specified, the following input and output variables should be specified:

in

1. ϕ (*nodalArray*, 1 component, required); scalar quantity to compute the Laplacian.
2. $\mathcal{K}$ (*nodalArray*, 9-components, required); 9-component diffusion tesnore. Note that in one- and two-dimensions, the ignorable coordinates will be multiplied by zero.

out

1. $c\nabla \cdot (\mathcal{K} \nabla \phi)$ (*nodalArray*, 1-component, required); Laplacian of ϕ.

gradientOfDivergence (*derivative = gradientOfDivergence*) Computes $c\nabla (\kappa \nabla \cdot \mathbf{v})$ where $\mathbf{v}$ is a vector quantity and κ is a scalar quantity defined on the grid and c is a constant coefficient. When *derivative = gradientOfDivergence* is specified, the following input and output variables should be specified:

in

1. $\mathbf{v}$ (*nodalArray*, 3-components, required); vector quantity to compute the Laplacian.
2. κ (*nodalArray*, 1-component, required); scalar diffusion coefficient.

out

1. $c\nabla (\kappa \nabla \cdot \mathbf{v})$ (*nodalArray*, 1-components, required); Laplacian of $\mathbf{v}$.

8.14.3 Example

The following code block demonstrates the least squares diffusion operator for computing the laplacian of a scalar quantity with a scalar diffusion coefficient:

```
<Updater leastSquaresDiffusion>
  kind = diffusion2d
  onGrid = domain
  derivative = diffusion
  numScalars = 2
  coefficient = 1.0
  numberOfInterpolationPoints = 8
  in = [q,D]
  out = [qnew]
</Updater>
```

The following code block demonstrates the least squares diffusion operator for computing the laplacian of a scalar quantity with a tensor diffusion coefficient:

```
<Updater computeDiffusion>
  kind = diffusion2d
  derivative = anisotropicDiffusion
  onGrid = domain
  numScalars = 3
  coefficient = 1.0

  orderAccuracy = 1
  numberOfInterpolationPoints = 8

  in = [temperature, conductivityTensor]
  out = [temperatureNew]
</Updater>
```

The following code block demonstrates the least squares diffusion operator for computing the laplacian of a vector quantity with a scalar diffusion coefficient:

```
<Updater derivative>
  kind = diffusion2d
  derivative = gradientOfDivergence
  coefficient = 1.0
  numberOfInterpolationPoints = 8
  orderAccuracy = 2
  onGrid = domain
  in = [q, diffusionCoefficient]
  out = [qnew]
</Updater>
```

The following Updater *kind* attributes can be specified can be used to compute finite volume discretizations of Navier-Stokes and RANS viscous operators:

8.15 navierStokesViscousOperator (1d, 2d, 3d)

The *navierStokesViscousOperator* computes the viscous stress and thermal conduction terms, with contributions from laminar (and optionally, turbulent) viscosity in the Navier Stokes equations using a conservative least squares interpolation scheme.

USim implements the viscous stress terms in the momentum and total energy equations in the Navier-Stokes operator using the form:

$$\mathcal{S}(\mathbf{w}) = \begin{bmatrix} \mathcal{S}_{\rho\mathbf{u}}(\mathbf{w}) \\ S_E(\mathbf{w}) \end{bmatrix} = \begin{bmatrix} c\nabla\cdot\mathbb{S}(\mathbf{w}) \\ c\nabla\cdot\{\mathbb{S}(\mathbf{w})\cdot\mathbf{u}\} \end{bmatrix}$$

$$\mathbb{S}(\mathbf{w}) = 2\left[\mu+\mu_{\text{turb}}\right]\left[\frac{\nabla\mathbf{u}+(\nabla\mathbf{u})^{\text{T}}}{2}-\frac{\mathbb{I}}{3}\nabla\cdot\mathbf{u}\right]-\frac{2\mathbb{I}}{3}\mu_{\text{turb}}\rho k$$

Here, $\mathbf{w}$ is the primitive state vector, $\mathbf{u}$ is the fluid velocity, μ is the laminar viscosity, μ_{turb} is the turbulent viscosity, ρ is the fluid density and k is the local local turbulent kinetic energy. The laminar viscosity can be computed through (e.g.) Sutherland's law:

$$\mu = \mu_0\left(\frac{T}{T_0}\right)^{\frac{3}{2}}\left(\frac{T_0+S}{T+S}\right)$$

where $\mu_0 = 1.716 \times 10^{-5}\text{kg (ms)}^{-1}$, $T_0 = 491.6R$, $S = 198.6R$.

USim implements the thermal conduction terms in the total energy equation in the Navier-Stokes operator using the form:

$$S_E(\mathbf{w}) = c\nabla \cdot [-(\kappa + \kappa_{\text{turb}})\nabla T]$$

where κ and κ_{turb} are the laminar and turbulent heat conductivity, which can be modelled through:

$$\kappa = \frac{c_p \mu}{\text{Pr}}; \quad \kappa_{\text{turb}} = \frac{c_p \mu_{\text{turb}}}{\text{Pr}_{\text{turb}}}$$

where c_p is the heat capacity at constant pressure and Pr and Pr_{turb} are the laminar and turbulent Prandtl numbers.

8.15.1 Data

`in` (string vector, required) Defined by the choice of the `enableThermal`, `enableViscous`, and `enableTurbulence` flags defined below in the Parameters section. Input variables must be in the order listed below. The rules for the which arrays should be included in the vector are as follows:

- `fluid velocity`: required if *enableViscous* = *true* or *enableTurbulence* = *true*
- `total viscosity`: required if *enableViscous* = *true* or *enableTurbulence* = *true*
- `fluid temperature`: required if *enableThermal* = *true* or *enableTurbulence* = *true*
- `total thermal conductivity`: required if *enableThermal* = *true* or *enableTurbulence* = *true*
- `turbulence model`: required if *enableTurbulence* = *true*
- `turbulence viscosity`: required if *enableTurbulence* = *true*

Note: If *enableTurbulence* is true and *enableViscous* is false, then *enableTurbulence* has no effect.

If *enableTurbulence* and *enableThermal* don't impact each other algorithmically.

If *enableTurbulence* is true, then the input variables are required to be present.

`Fluid velocity` (*nodalArray*, 3-components, optional) Vector of fluid velocities, required if *enableViscous* = *true*: 0. $u_{\hat{\mathbf{i}}} = \mathbf{u} \cdot \hat{\mathbf{i}}$: velocity in the $\hat{\mathbf{i}}$ direction 1. $u_{\hat{\mathbf{j}}} = \mathbf{u} \cdot \hat{\mathbf{j}}$: velocity in the $\hat{\mathbf{j}}$ direction 2. $u_{\hat{\mathbf{k}}} = \mathbf{u} \cdot \hat{\mathbf{k}}$: velocity in the $\hat{\mathbf{k}}$ direction

`Total viscosity` (*nodalArray*, 1-components, optional) Sum of the laminar and turbulent viscosities, $\mu + \mu_{\text{turb}}$ (if turbulence is not modelled, the latter can be ommitted). Required if *enableViscous* = *true*:

`Fluid Temperature` (*nodalArray*, 1-components, optional) Required if *enableThermal* = *true*.

`Total thermal conductivity` (*nodalArray*, 1-components, optional) Sum of the laminar and turbulent thermal conductivities, $\mu + \mu_{\text{turb}}$ (if turbulence is not modelled, the latter can be ommitted). Required if *enableThermal* = *true*.

`Turbulence model` (*nodalArray*, 1-components, optional) Required if *enableTurbulence* = *true*. One of:

`Turbulent kinetic energy density` $\rho k - \rho\epsilon$

`Turbulent kinetic energy density dissipation rate` $\rho k - \rho\omega$

`Turbulent viscosity` (*nodalArray*, 1-components, optional) Turbulent viscosities, μ_{turb}. Required if *enableTurbulence* = *true*:

out (string vector, required)

Vector of Fluxes (*nodalArray*, 4-components)

0. $\mathcal{S}\left(\rho u_{\hat{\mathbf{i}}}\right)$: $\hat{\mathbf{i}}$ momentum flux
1. $\mathcal{S}\left(\rho u_{\hat{\mathbf{j}}}\right)$: $\hat{\mathbf{j}}$ momentum flux
2. $\mathcal{S}\left(\rho u_{\hat{\mathbf{k}}}\right)$: $\hat{\mathbf{k}}$ momentum flux
3. $\mathcal{S}\left(E\right)$: total energy flux

8.15.2 Parameters

The *navierStokesViscousOperator* updater accepts the parameters below, in addition to those required by *Updater*:

orderAccuracy (integer, required) Order of the polynomial that is used to form the operator. Choice of 1, 2 or 3 corresponding, respectively to first, second and third order accuracy. The appropriate choice of order varies on the problem type and the mesh used.

numberOfInterpolationPoints (integer, required) Number of points to be considerd for the least squares fit. This parameter varies from mesh to mesh and should be determined by computing a known function on the mesh.

The numberOfInterpolationPoints must be greater than (or equal to) the number of coefficients in the polynomial approximation. This means that in 1d the value is 4, in 2D the value is at least 6 and in 3D the value is at least 10.

These choices do not guarantee that a matrix inverse will be found. The following values though appear to be adequate in general: in 1D 4; in 2D 8 and in 3D 20.

coefficient (float, required) Constant floating point value, c that multiplies the output of the diffusion updater.

enableThermal (boolean, optional) Tell USim whether to include the contribution from heat conduction. Default true.

enableViscous (boolean, optional) Tell USim whether to inclue the contribution from viscous stress. Default true.

enableTurbulence (boolean, optional) Tell USim whether to include the turbulence. Default false.

8.15.3 Examples

```
<Updater computeViscousSource>
  kind = navierStokesViscousOperator2d
  onGrid = domain

  coefficient = 1.0

  numberOfInterpolationPoints = 8
  orderAccuracy = 2

  enableThermal = false
  enableViscous = true

  in = [velocity, dynamicViscosity, temperature, thermalCoefficient]
```

```
  out = [viscousSource]
</Updater>
```

```
<Updater computeViscousSource>
  kind = navierStokesViscousOperator2d
  onGrid = domain
  isRadial = true
  coefficient = 1.0

  numberOfInterpolationPoints = 8

  enableThermal = false
  enableViscous = true
  enableTurbulence = true
  temperatureIndex = 0

  in = [velocity, totalVisc, avgTemp, totalCond, kEpsilon, turbulentViscosity]
  out = [viscousSource]
</Updater>
```

8.16 kOmegaOperator (1d, 2d, 3d)

The *kOmegaOperator* implements the right-hand side of the "Standard" Menter SST Two-Equation Model:

$$\frac{\partial(\rho k)}{\partial t} + \nabla\cdot[\rho\mathbf{u}k] = P - \beta^\star\rho\omega k + \nabla\cdot[(\mu+\sigma_k\mu_{\text{turb}})\nabla k]$$

$$\frac{\partial(\rho\omega)}{\partial t} + \nabla\cdot[\rho\mathbf{u}\omega] = \frac{\gamma}{\nu_{\text{turb}}}P - \beta^\star\rho\omega^2 + \nabla\cdot[(\mu+\sigma_k\mu_{\text{turb}})\nabla\omega] + 2(1-F_1)\frac{\rho\sigma_\omega}{\omega 2}\nabla k\nabla\omega$$

The full details of this model, including the definition of the various constants, etc. can be found at http://turbmodels.larc.nasa.gov/sst.html

The *kOmegaOperator* operator computes the right-hand side of this model:

$$\mathcal{S}_{\rho k} = P - \beta^\star\rho\omega k + \nabla\cdot[(\mu+\sigma_k\mu_{\text{turb}})\nabla k]$$

$$\mathcal{S}_{\rho\omega} = \frac{\gamma}{\nu_{\text{turb}}}P - \beta^\star\rho\omega^2 + \nabla\cdot[(\mu+\sigma_k\mu_{\text{turb}})\nabla\omega] + 2(1-F_1)\frac{\rho\sigma_\omega}{\omega 2}\nabla k\nabla\omega$$

The advective terms, $\nabla\cdot[\rho\mathbf{u}k]$ and $\nabla\cdot[\rho\mathbf{u}\omega]$ can be computed using *classicMusclUpdater (1d, 2d, 3d)* combined with *multiSpeciesSingleVelocityEqn*.

8.16.1 Data

`in` (string vector of 7, required)

`Fluid Model` (*nodalArray*, 5-components, required) The vector of conserved quantities for the fluid model, $\mathbf{q}$ has 5 entries:

0. ρ: mass density
1. $\rho\, u_{\hat{\mathbf{i}}} = \rho\mathbf{u}\cdot\hat{\mathbf{i}}$: momentum density in the $\hat{\mathbf{i}}$ direction
2. $\rho\, u_{\hat{\mathbf{j}}} = \rho\mathbf{u}\cdot\hat{\mathbf{j}}$: momentum density in the $\hat{\mathbf{j}}$ direction
3. $\rho\, u_{\hat{\mathbf{k}}} = \rho\mathbf{u}\cdot\hat{\mathbf{k}}$: momentum density in the $\hat{\mathbf{k}}$ direction
4. $E = \frac{P}{\gamma-1} + \frac{1}{2}\rho|\mathbf{u}|^2$: total energy density

`Turbulence model` (*nodalArray*, 2-components, required) The vector of conserved quantities for the turbulence model:

0. ρk
1. $\rho \omega$

`Fluid velocity` (*nodalArray*, 3-components, required) Vector of fluid velocities, required if *enableViscous* = *true*:

0. $u_{\hat{\mathbf{i}}} = \mathbf{u} \cdot \hat{\mathbf{i}}$: velocity in the $\hat{\mathbf{i}}$ direction
1. $u_{\hat{\mathbf{j}}} = \mathbf{u} \cdot \hat{\mathbf{j}}$: velocity in the $\hat{\mathbf{j}}$ direction
2. $u_{\hat{\mathbf{k}}} = \mathbf{u} \cdot \hat{\mathbf{k}}$: velocity in the $\hat{\mathbf{k}}$ direction

`Fluid Temperature` (*nodalArray*, 1-components, required)

`Dynamic Viscosity` (*nodalArray*, 1-components, required)

`Thermal Conductivity` (*nodalArray*, 1-components, required)

`Distance from Wall` (*nodalArray*, 1-components, required)

`out` (string vector of 4, required) `Vector of Fluid Model Source terms` (*nodalArray*, 5-components, required)

0. $\mathcal{S}(\rho)$: mass source
1. $\mathcal{S}\left(\rho\, u_{\hat{\mathbf{i}}}\right)$: $\hat{\mathbf{i}}$ momentum source
2. $\mathcal{S}\left(\rho\, u_{\hat{\mathbf{j}}}\right)$: $\hat{\mathbf{j}}$ momentum source
3. $\mathcal{S}\left(\rho\, u_{\hat{\mathbf{k}}}\right)$: $\hat{\mathbf{k}}$ momentum source
4. $\mathcal{S}(E)$: total energy source

`Vector of Turbulence Model Source terms` (*nodalArray*, 2-components, required)

0. $\mathcal{S}(\rho k)$
1. $\mathcal{S}(\rho \omega)$

`Turbulent viscosity` (*nodalArray*, 1-component, required)

`Maximum turbulent diffusion` (*nodalArray*, 1-component, required)

8.16.2 Parameters

The *kOmegaOperator* updater accepts the parameters below, in addition to those required by *Updater*:

`numberOfInterpolationPoints` (integer, required) Number of points to be considerd for the least squares fit. This parameter varies from mesh to mesh and should be determined by computing a known function on the mesh.

The numberOfInterpolationPoints must be greater than (or equal to) the number of coefficients in the polynomial approximation. This means that in 1d the value is 4, in 2D the value is at least 6 and in 3D the value is at least 10.

These choices do not guarantee that a matrix inverse will be found. The following values though appear to be adequate in general: in 1D 4; in 2D 8 and in 3D 20.

`orderAccuracy` (integer, option) Order of the polynomial that is used to form the operator. Choice of 1, 2 or 3 corresponding, respectively to first, second and third order accuracy. The appropriate choice of order varies on the problem type and the mesh used. Defaults to 2.

`turbulentPrandtlNumber` (float, required) Prandtl number for turbulent flows, which is the ratio of eddy diffusivities of momentum and heat transfer

`Cp` (float, required) Specific heat at constant pressure

8.16.3 Example

```
<Updater computeRansSource>
   kind = kOmegaOperator2d
   onGrid = domain
   coefficient = 1.0
   numberOfInterpolationPoints = 16
   turbulentPrandtlNumber = 0.85
   Cp = CP
   in = [q, kOmega, velocity, temperature, visc, cond, distance]
   out = [dummySource, kOmegaSource, turbulentViscosity, maxTurbulentDiffusion]
 </Updater>
```

8.17 kEpsilonOperator (1d, 2d, 3d)

Estimates the turbulent viscosity using k-epsilon model (http://turbmodels.larc.nasa.gov/ke-chien.html).

The *kEpsilonOperator* implements the right-hand side of the "Standard" Menter SST Two-Equation Model:

$$\frac{\partial(\rho k)}{\partial t} + \nabla\cdot[\rho\mathbf{u}k] = P - \rho\epsilon + \nabla\cdot\left[\left(\mu + \frac{\mu_{\text{turb}}}{\sigma_k}\right)\nabla k\right] + \rho L_k$$

$$\frac{\partial(\rho\epsilon)}{\partial t} + \nabla\cdot[\rho\mathbf{u}\epsilon] = \frac{C_{\epsilon_1} f_1 \epsilon}{k} P - \frac{C_{\epsilon_2} f_2 \rho\epsilon^2}{k} + \nabla\cdot\left[\left(\mu + \frac{\mu_{\text{turb}}}{\sigma_\epsilon}\right)\nabla\epsilon\right] + \rho L_\epsilon$$

The full details of this model, including the definition of the various constants, etc. can be found at (http://turbmodels.larc.nasa.gov/ke-chien.html)

The *kEpsilonOperator* operator computes the right-hand side of this model:

$$\mathcal{S}_{\rho k} = P - \rho\epsilon + \nabla\cdot\left[\left(\mu + \frac{\mu_{\text{turb}}}{\sigma_k}\right)\nabla k\right] + \rho L_k$$

$$\mathcal{S}_{\rho\omega} = \frac{C_{\epsilon_1} f_1 \epsilon}{k} P - \frac{C_{\epsilon_2} f_2 \rho\epsilon^2}{k} + \nabla\cdot\left[\left(\mu + \frac{\mu_{\text{turb}}}{\sigma_\epsilon}\right)\nabla\epsilon\right] + \rho L_\epsilon$$

The advective terms, $\nabla\cdot[\rho\mathbf{u}k]$ and $\nabla\cdot[\rho\mathbf{u}\omega]$ can be computed using *classicMusclUpdater (1d, 2d, 3d)* combined with *multiSpeciesSingleVelocityEqn*.

8.17.1 Data

`in` (string vector of 7, required)

`Fluid Model` (*nodalArray*, 5-components, required) The vector of conserved quantities for the fluid model, **q** has 5 entries:

0. ρ: mass density

1. $\rho\, u_{\hat{\mathbf{i}}} = \rho \mathbf{u} \cdot \hat{\mathbf{i}}$: momentum density in the $\hat{\mathbf{i}}$ direction
2. $\rho\, u_{\hat{\mathbf{j}}} = \rho \mathbf{u} \cdot \hat{\mathbf{j}}$: momentum density in the $\hat{\mathbf{j}}$ direction
3. $\rho\, u_{\hat{\mathbf{k}}} = \rho \mathbf{u} \cdot \hat{\mathbf{k}}$: momentum density in the $\hat{\mathbf{k}}$ direction
4. $E = \frac{P}{\gamma - 1} + \frac{1}{2}\rho |\mathbf{u}|^2$: total energy density

`Turbulence model` (*nodalArray*, 2-components, required) The vector of conserved quantities for the turbulence model:

0. ρk
1. $\rho \epsilon$

`Fluid velocity` (*nodalArray*, 3-components, required) Vector of fluid velocities, required if *enableViscous* = *true*:

0. $u_{\hat{\mathbf{i}}} = \mathbf{u} \cdot \hat{\mathbf{i}}$: velocity in the $\hat{\mathbf{i}}$ direction
1. $u_{\hat{\mathbf{j}}} = \mathbf{u} \cdot \hat{\mathbf{j}}$: velocity in the $\hat{\mathbf{j}}$ direction
2. $u_{\hat{\mathbf{k}}} = \mathbf{u} \cdot \hat{\mathbf{k}}$: velocity in the $\hat{\mathbf{k}}$ direction

`Fluid Temperature` (*nodalArray*, 1-components, required)

`Dynamic Viscosity` (*nodalArray*, 1-components, required)

`Thermal Conductivity` (*nodalArray*, 1-components, required)

`Distance from Wall` (*nodalArray*, 1-components, required)

`out` (string vector of 4, required) `Vector of Fluid Model Source terms` (*nodalArray*, 5-components, required)

0. $\mathcal{S}\left(\rho\right)$: mass source
1. $\mathcal{S}\left(\rho\, u_{\hat{\mathbf{i}}}\right)$: $\hat{\mathbf{i}}$ momentum source
2. $\mathcal{S}\left(\rho\, u_{\hat{\mathbf{j}}}\right)$: $\hat{\mathbf{j}}$ momentum source
3. $\mathcal{S}\left(\rho\, u_{\hat{\mathbf{k}}}\right)$: $\hat{\mathbf{k}}$ momentum source
4. $\mathcal{S}\left(E\right)$: total energy source

`Vector of Turbulence Model Source terms` (*nodalArray*, 2-components, required)

0. $\mathcal{S}\left(\rho k\right)$
1. $\mathcal{S}\left(\rho \epsilon\right)$

`Turbulent viscosity` (*nodalArray*, 1-component, required)

`Maximum turbulent diffusion` (*nodalArray*, 1-component, required)

8.17.2 Parameters

The *kEpsilonOperator* updater accepts the parameters below, in addition to those required by *Updater*:

`numberOfInterpolationPoints` (integer, required) Number of points to be considerd for the least squares fit. This parameter varies from mesh to mesh and should be determined by computing a known function on the mesh.

The numberOfInterpolationPoints must be greater than (or equal to) the number of coefficients in the polynomial approximation. This means that in 1d the value is 4, in 2D the value is at least 6 and in 3D the value is at least 10.

These choices do not guarantee that a matrix inverse will be found. The following values though appear to be adequate in general: in 1D 4; in 2D 8 and in 3D 20.

`orderAccuracy` (integer, option) Order of the polynomial that is used to form the operator. Choice of 1, 2 or 3 corresponding, respectively to first, second and third order accuracy. The appropriate choice of order varies on the problem type and the mesh used. Defaults to 2.

`turbulentPrandtlNumber` (float, required) Prandtl number for turbulent flows, which is the ratio of eddy diffusivities of momentum and heat transfer

`Cp` (float, required) Specific heat at constant pressure

`minDt` (float, required) The dissipation time step is proportion to k/ϵ, which can go to infinity. We limit this ratio to the value specified by *minDt*.

`computeViscosity` (bool, required) Whether to compute the turbulent viscosity

`computeSource` (bool, required) Whether to compute source terms for the turbulence model.

8.17.3 Example

```
<Updater computeRansSource>
  kind = kEpsilonOperator2d
  onGrid = domain
  coefficient = 1.0
  numberOfInterpolationPoints = 16
  turbulentPrandtlNumber = 0.85
  minDt = MINDT
  computeViscosity = false
  computeSource = true
  Cp = CP
  in = [q, kEpsilon, velocity, temperature, visc, cond, distance]
  out = [dummySource, kEpsilonSource, turbulentViscosity, maxTurbulentDiffusion]
</Updater>
```

The following Updater *kind* attributes can be specified can be used to compute the generalized Ohm's law for an ionized plasma:

8.18 generalizedOhmsLaw (1d, 2d, 3d)

The *generalizedOhmsLaw* updater computes the electric field determined by parameters in the generalized Ohm's law.

$$\mathbf{E} = \eta \mathbf{J} + \left[- \left(\mathbf{J} - \mathbf{J}_{\text{ion}} \right) \times \mathbf{B} + \nabla P_e \right] \left(n_e q_e \right)^{-1} \tag{8.-12}$$

8.18.1 Data

`in` (string vector of 3, required)

`Vector of Conserved Quantities` (*nodalArray*, 9-components, required) The vector of conserved quantities, $\mathbf{q}$ has 9 entries:

0. ρ: mass density

1. $\rho\, u_{\hat{\mathbf{i}}} = \rho\mathbf{u}\cdot\hat{\mathbf{i}}$: momentum density in the $\hat{\mathbf{i}}$ direction
2. $\rho\, u_{\hat{\mathbf{j}}} = \rho\mathbf{u}\cdot\hat{\mathbf{j}}$: momentum density in the $\hat{\mathbf{j}}$ direction
3. $\rho\, u_{\hat{\mathbf{k}}} = \rho\mathbf{u}\cdot\hat{\mathbf{k}}$: momentum density in the $\hat{\mathbf{k}}$ direction
4. $E = \frac{P}{\gamma-1} + \frac{1}{2}\rho|\mathbf{u}|^2 + \frac{1}{2}|\mathbf{b}|^2$: total energy density
5. $b_{\hat{\mathbf{i}}} = \mathbf{b}\cdot\hat{\mathbf{i}} = \mu_0^{-1/2}\mathbf{B}\cdot\hat{\mathbf{i}}$: magnetic field normalized by permeability of free-space in the $\hat{\mathbf{i}}$ direction
6. $b_{\hat{\mathbf{j}}} = \mathbf{b}\cdot\hat{\mathbf{j}} = \mu_0^{-1/2}\mathbf{B}\cdot\hat{\mathbf{j}}$: magnetic field normalized by permeability of free-space in the $\hat{\mathbf{j}}$ direction
7. $b_{\hat{\mathbf{k}}} = \mathbf{b}\cdot\hat{\mathbf{k}} = \mu_0^{-1/2}\mathbf{B}\cdot\hat{\mathbf{k}}$: magnetic field normalized by permeability of free-space in the $\hat{\mathbf{k}}$ direction

`Current Density` (*nodalArray*, 3-components, required) Vector of plasma currents:

0. $J_{\hat{\mathbf{i}}} = \mathbf{J}\cdot\hat{\mathbf{i}}$: current in the $\hat{\mathbf{i}}$ direction
1. $J_{\hat{\mathbf{j}}} = \mathbf{J}\cdot\hat{\mathbf{j}}$: current in the $\hat{\mathbf{j}}$ direction
2. $J_{\hat{\mathbf{k}}} = \mathbf{J}\cdot\hat{\mathbf{k}}$: current in the $\hat{\mathbf{k}}$ direction

`Charge State` (*nodalArray*, 1-component, required) Z, the ionization state.

`out` (string vector, required)

`Electric field` (*nodalArray*, 3-components, required) Vector of electric fields:

0. $E_{\hat{\mathbf{i}}} = \mathbf{E}\cdot\hat{\mathbf{i}}$: electric field in the $\hat{\mathbf{i}}$ direction
1. $E_{\hat{\mathbf{j}}} = \mathbf{E}\cdot\hat{\mathbf{j}}$: electric field in the $\hat{\mathbf{j}}$ direction
2. $E_{\hat{\mathbf{k}}} = \mathbf{E}\cdot\hat{\mathbf{k}}$: electric field in the $\hat{\mathbf{k}}$ direction

`resistivity` (*nodalArray*, 1-component, optional) Scalar resistivity: if this term exists, then the resistive term is included in the evaluation of E.

`electronPressureDerivative` (*nodalArray*, 3-components, optional) ∇P_e If this term is set then the diamagnetic drift term is used in determining the electric field.

8.18.2 Parameters

The *generalizedOhmsLaw* updater accepts the parameters below, in addition to those required by *Updater*:

`idealTerm` (boolean) Set to false if the ideal term $\mathbf{u}\times\mathbf{B}$ should be ignored, otherwise set to true. Defaults to true.

`hallTerm` (boolean) Set to false if the Hall term should be ignored, otherwise set to true. Defaults to false.

`fundamentalCharge` (float) The charge of a proton

`ionMass` (float) mass of the ion. Currently assumes only one ion species

`electronMass` (float) mass of the electron.

`boltzmannConstant` (float) boltzmann's constant

8.18.3 Example

```
<Updater computeE>
  kind = generalizedOhmsLaw1d
  onGrid = domain

  in = [q, J, Zbar]
  out = [E]
  electronPressureDerivative = gradPe
  resistivity = etaJ

  hallTerm = true

  fundamentalCharge = CHARGE
  ionMass = MI
  electronMass = ME
  boltzmannConstant = KB
</Updater>
```

8.19 resitiveOperator (1d, 2d, 3d)

The *resistiveOperator* computes sources terms for the MHD equations using a conservative least squares gradient method:

$$\mathcal{S}(\mathbf{w}) = \begin{bmatrix} S_E(\mathbf{w}) \\ S_{\mathbf{B}}(\mathbf{w}) \end{bmatrix} = \begin{bmatrix} \nabla \cdot \left(\mu_0^{-1}\mathbf{E}_{\text{Extended}} \times \mathbf{B}\right) \\ \nabla \times \mathbf{E}_{\text{Extended}} \end{bmatrix}$$

$$\mathbf{E}_{\text{Extended}} = \mu_0^{-1}\left[\eta_{\text{Ohmic}}\nabla \times \mathbf{B} - \eta_{\text{Hall}}\left(\nabla \times \mathbf{B}\right) \times \mathbf{B}\right]$$

$$\eta_{\text{Hall}} = -\left(\frac{\rho Q Z}{m_{\text{ion}}}\right)^{-1}$$

Here, η_{Ohmic} is the Ohmic resistvity, η_{Hall} is the Hall coefficient, rho is the fluid density, Q is the charge on a proton, Z is the ion charge state and m_{ion} is the ion mass.

8.19.1 Data

`in` (string vector of 4, required)

`Magnetic field` (***nodalArray***, **3-components, required**)

0. $B_{\hat{\mathbf{i}}} = \mathbf{B} \cdot \hat{\mathbf{i}}$: magnetic field in the $\hat{\mathbf{i}}$ direction
1. $B_{\hat{\mathbf{j}}} = \mathbf{B} \cdot \hat{\mathbf{j}}$: magnetic field in the $\hat{\mathbf{j}}$ direction
2. $B_{\hat{\mathbf{k}}} = \mathbf{B} \cdot \hat{\mathbf{k}}$: magnetic field in the $\hat{\mathbf{k}}$ direction

`Ohmic Resistivity` (***nodalArray***, **1-components, optional**) Scalar ohmic resistivity, η_{Ohmic}; required if *enableOhmicTerm* = *true* (see below).

`Mass density` (***nodalArray***, **1-components, optional**) Fluid mass density, ρ; required if *enableHallTerm* = *true* (see below).

`Charge State` (***nodalArray***, **1-components, optional**) Fluid charge state, Z; required if *enablePartiallyIonized* = *true* (see below).

8.19.2 Parameters

The *resistiveOperator* updater accepts the parameters below, in addition to those required by *Updater*:

`orderAccuracy` (integer, required) Order of the polynomial that is used to form the operator. Choice of 1, 2 or 3 corresponding, respectively to first, second and third order accuracy. The appropriate choice of order varies on the problem type and the mesh used.

`numberOfInterpolationPoints` (integer, required) Number of points to be considerd for the least squares fit. This parameter varies from mesh to mesh and should be determined by computing a known function on the mesh.

The numberOfInterpolationPoints must be greater than (or equal to) the number of coefficients in the polynomial approximation. This means that in 1d the value is 4, in 2D the value is at least 6 and in 3D the value is at least 10.

These choices do not guarantee that a matrix inverse will be found. The following values though appear to be adequate in general: in 1D 4; in 2D 8 and in 3D 20.

`coefficient` (float, required) Constant floating point value, c that multiplies the output of the diffusion updater.

`permeability` (float, required) The permeability of free space, μ_0.

`enableOhmicTerm` (bool, optional) Include Ohmic resistivity, $\eta_{\text{Ohmic}} \nabla \times \mathbf{B}$ in the extended MHD electric field. Default: *true*.

`enableHallTerm` (bool, optional) Include the Hall effect, $\eta_{\text{Hall}} \left(\nabla \times \mathbf{B}\right) \times \mathbf{B}$ in the extended MHD electric field. Default: *false*.

If *enableHallTerm* = *true*, then the following parameters are available:

`ionMass` (float, optional) The mass of an ion. Default value is 1.0.

`fundamentalCharge` (float, optional) The charge of a proton. Default value is 1.0.

`enablePartiallyIonized` (bool, optional) Whether to include the ion ionization state in the Hall effect. Default is false, in which case the ion is assumed to be singly ionized.

8.19.3 Example

```
<Updater computeResistiveSources>
  kind = resistiveOperator2d
  onGrid = domain

  coefficient = 1.0
  permeability = 1.0

  numberOfInterpolationPoints = 8

  in = [magneticField, resistivity]
  out = [source]
</Updater>
```

The following Updater *kind* attributes can be specified to perform operations related to time advance (time integration, time step restrictions):

8.20 multiUpdater (1d, 2d, 3d)

The *multiUpdater* references several updaters and uses them to perform an explicit time-integration. The *multiUpdater* typically used to advance systems of the form:

$$\frac{\partial \mathbf{q}}{\partial t} + \nabla \cdot [\mathcal{F}(\mathbf{w})] = \mathcal{S}(\mathbf{w})$$

Time integration schemes currently supported by *multiUpdater* include 1st, 2nd and 3rd order Runge-Kutta, subcycling and 1st, 2nd order accurate Super Time Step methods.

The *multiUpdater* updater accepts the parameters required by *Updater*.

The operations performed by the *multiUpdater* are specified using the *UpdateStep* and *UpdateSequence* pattern used for the main USim input file. Note that only the *loop* attribute for the *UpdateSequence* is used by the *multiUpdater* and that the attribute "operation = integrate" must be specified in the final *UpdateStep* in the loop in order for USim to perform the time integration.

The time integration scheme used by the *multiUpdater* is specified by the use of a *Time Integrator* block.

8.20.1 Data

in (string vector, required) Input 1 to N are input *nodalArray*s to be used in the updaters specified in the UpdateStep.

out (string vector, required) Output 1 to N are output *nodalArray*s resulting from the integration UpdateStep.

8.20.2 Sub-Blocks

TimeIntegrator *Time Integrator* Currently only one time integration scheme can be specified.

Time integration schemes currently supported by *multiUpdater* include 1st, 2nd and 3rd order Runge-Kutta, subcycling and 1st, 2nd order accurate Super Time Step methods.

UpdateSequence *UpdateSequence* This block is used to set the sequence of update steps

UpdateStep *UpdateStep* The steps used in the update

8.20.3 Example

The code block below demonstrates the use of a multiUpdater to solve a multi-species fluid problem with collision operators and boundary conditions:

```
<Updater rkUpdater>
  kind = multiUpdater1d
  onGrid = domain

  in = [q, q1, q2, q3]
  out = [qnew, qnew1, qnew2, qnew3]

  <TimeIntegrator rkIntegrator>
    kind = rungeKutta1d
    ongrid = domain
    scheme = third
  </TimeIntegrator>
```

```
    <UpdateSequence sequence>
      loop = [boundaries,hyper]
    </UpdateSequence>

    <UpdateStep boundaries>
       updaters = [openBoundaries, openBoundaries1, openBoundaries2, openBoundaries3]
       syncVars = [q, q1, q2, q3]
    </UpdateStep>

    <UpdateStep hyper>
       operation = "integrate"
       updaters = [\
                computeN1, computeN2, computeN3, \
                computeT1, computeT2, computeT3, \
                computeV1, computeV2, computeV3, \
                collisionFrequency, \
                momentumSource, energySource,\
                hyper, hyper1, hyper2, hyper3, \
                addThermalRelaxation1, addThermalRelaxation2, addThermalRelaxation3 \
                  ]
       syncVars = [qnew, qnew1, qnew2, qnew3]
    </UpdateStep>

  </Updater>
```

8.21 implicitMultiUpdater (1d, 2d, 3d)

The *implicitMultiUpdater* provides Jacobian Free Newton-Krylov methods with a range of non-linear and linear solvers and preconditioning strategies for solving elliptic or implicit hyperbolic problems. The *implicitMultiUpdater* casts these problems in residual form:

$$\mathcal{R}(\mathbf{q}) = 0$$

This formulation can then be used to solve linear problems, such as Poisson's equation:

$$\mathcal{R}(\mathbf{q}) = \nabla^2\phi - \sum_{\text{species}} Q_{\text{species}}$$

or, non linear problems such as a backward Euler discretization of a non-linear hyperbolic equation:

$$\mathcal{R}(\mathbf{q}) = q^{n+1} - q^n + \Delta t\left\{\nabla\cdot\left[\mathcal{F}\left(\mathbf{w}^{n+1}\right)\right] - \mathcal{S}\left(\mathbf{w}^{n+1}\right)\right\}$$

The operations performed by the *implictMultiUpdater* are specified using the *UpdateStep* and *UpdateSequence* pattern used for the main USim input file. Note that only the *loop* attribute for the *UpdateSequence* is used by the *implicitMultiUpdater*. If the *implictMultiUpdater* is being used to solve a system that includes a time discretization (e.g. the backward Euler example above), then the attribute "operation = integrate" must be specified in the final *UpdateStep* in the loop in order for USim to perform the time integration. The *UpdateStep* attribute "operation = operate" is not compatible with the *implicitMultiUpdater*.

The time integration scheme used by the *multiUpdater* is specified by the use of a *Time Integrator* block.

Preconditioning of the *implictMultiUpdater* can be specified by the addition of a *Preconditioner* block. one preconditioner can be specified.

8.21.1 Data

`in` (string vector, required) The *implicitMultiUpdater* accepts *exactly* one input variable:

`Vector of Unknowns` (*nodalArray*, n-components, required) The vector of unknowns, $\mathbf{q}$ at time t^n. Must have the attribute *useEpetraVector = true* defined in the associated dataStruct block.

`out` (string vector, required) The *implicitMultiUpdater* accepts *exactly* one output variable:

`Vector of Unknowns` (*nodalArray*, n-components, required) The vector of unknowns, $\mathbf{q}$ at time t^{n+1}. Must have the attribute *useEpetraVector = true* defined in the associated dataStruct block and have the *same* number of components as the input vector of unknows.

`residual` (string vector, optional) A string vector that specifies exactly one *nodalArray* to store the residual, $\mathcal{R}(\mathbf{q})$ at the end of the solve. The specified *nodalArray* must have the attribute *useEpetraVector = true* defined in the associated dataStruct block.

`solverPerf` (string vector, optional) A string vector that specifies exactly one *dynVector* with 6 componets to store the solver performance. Data is appended to this data structure at each Newton iteration. The components correspond to:

0. Number of non-linear iterations.
1. The residual norm.
2. The number of linear iterations.
3. The achieved tolerance.
4. Time to solve the linear system.
5. Time to construct the preconditioner.

8.21.2 Parameters

The *implicitMultiUpdater* updater accepts the parameters below, in addition to those required by *Updater*:

`maxNonlinearIterations` (integer, required) Maximum number of outer Newton steps the solver will take before returning an error code.

`maxLinearIterations` (integer, required) The maximum number of inner linear (Krylov) iterations to take at each Newton iteration.

`numItersToStagnation` (integer, optional) The number of iterations that the stagnation condition (see *stagnationThreshold*) is allowed to be violated for before the outer Newton solve exits. Defaults to $\mathrm{maxNonlinearIterations}/4$

`linearTolerance` (float, required) The tolerance required to achieve convergence in the inner linear (Krylov) iterations for each Newton step.

`relativeResidual` (float, optional) The outer Newton problem converges when $\mathrm{currentResidual} \leq \mathrm{relativeResidual} \times \mathrm{initialResidual}$. Required if *convergenceCriteria = [relativeResidual]* is specified.

`absoluteResidual` (float, optional) The outer Newton problem converges when $\mathrm{currentResidual} \leq \mathrm{absoluteResidual}$. Required if *convergenceCriteria = [absoluteResidual]* is specified.

`stagnationThreshold` (float, optional) Causes the outer Newton solve to exit if $\mathrm{currentResidual} > \mathrm{stagnationThreshold} \times \mathrm{previousResidual}$ for *numItersToStagnation* iterations. Defaults to 1.0.

`convergenceTest` (string vector, optional) List of convergence tests to determine when the outer Newton solve is converged. Options include:

`relativeResidual` The outer Newton problem converges when $\text{currentResidual} \leq \text{relativeResidual} \times \text{initialResidual}$.

`absoluteResidual` The outer Newton problem converges when $\text{currentResidual} \leq \text{absoluteResidual}$

`normSolutionUpdate` The outer Newton problem converges when the norm of the change in the solution vector falls below 1.0e-3.

`rmsSolutionUpdate` The outer Newton problem converges when the weighted root mean square norm fo the solution update satisfies

$$\sqrt{\frac{1}{N}\sum_{i=1}^{N}\left(\frac{(q_i^k - q_i^{k-1})}{10^{-2}|q_i^{k-1}| + 10^{-8}}\right)^2} \leq 1.0$$

`finiteResidual` Causes the Newton solve to exit if NaN is encountered.

`residualStagnation` Causes the Newton solve to exit if $\text{currentResidual} > \text{stagnationThreshold} \times \text{previousResidual}$ for *numItersToStagnation* iterations.

8.21.3 Sub-Blocks

`TimeIntegrator` *Time Integrator* Currently only one time integration scheme can be specified.

`Preconditioner` *Preconditioner* Currently, only one preconditioner can be specified.

`UpdateSequence` *UpdateSequence* This block is used to set the sequence of update steps

`UpdateStep` *UpdateStep* The steps used in the update

8.21.4 Examples

The code block below demonstrates the *implicitMultiUpdater* for solving a three-dimensional Poisson problem using an algebraic multigrid preconditioner:

```
<Updater computeValues>
   kind = implicitMultiUpdater3d
   onGrid = domain
   in = [phi]
   out = [phiNew]
   residual = [f]
   solverPerf = [solverPerformance]
   maxNonlinearIterations = 4
   nonlinearTolerance = 1.e-12
   maxLinearIterations = 128
   linearTolerance = 1.e-14
   computePreconditioningMatrix = 1
   preconditioner = ML
   newtonForceMethod = Constant
   stencilUpdater = [computeNablaPhi]
   writePreconditioningMatrixToFile = 0

   <TimeIntegrator implicitUpdater>
     kind = implicit3d
     ongrid = domain
     scheme = none
   </TimeIntegrator>
```

```
    <Preconditioner myPreconditioner>
      # None/ML/AztecOO/Ifpack/New Ifpack
      preconditioner=ML
      # if 0, use a FD preconditioner
      computePreconditioningMatrix = 1
      # write out the preconditioning matrix at startup
      writePreconditioningMatrixToFile = 0
      # maximum age of preconditioner in outer Newton steps (needs a better name)
      linearMaxPrecAge = 10
      # rebuild, reuse or recompute preconditioner (needs a better name)
      linearReusePolicy = Reuse
      mlStrategy=classicSA # SA/DD
      stencilUpdater = [computeNablaPhi]
      testPreconditioner = false
      #mlDampingFactor=0.0675
      #mlSmoother = BSGS-A

      # Block to allow arbitray parameters to be passed to the preconditioner XML list
      #<ParameterList ml>
      #  increasing or decreasing = "increasing"
      #</ParameterList>
    </Preconditioner>

    <UpdateSequence sequence>
      loop = [boundaries,hyper]
    </UpdateSequence>

    <UpdateStep hyper>
       updaters = [computeNablaPhi,poisson,copyBc]
       syncVars = [phiNew]
    </UpdateStep>

    <UpdateStep boundaries>
       updaters = [dirchletBc]
       syncVars = [phi]
    </UpdateStep>

  </Updater>
```

The code block demonstrates the *implicitMultiUpdater* for solving a 2 or 3 dimensional compressible flow problem:

```
    <Updater computeValues>
       kind = implicitMultiUpdater$NDIM$d
       onGrid = domain
       inpIndices = [0]
       in = [q]
       out = [qNew]
       residual = [f]
       solverPerf = [solverPerformance]
       maxNonlinearIterations = 100
       nonlinearTolerance = 1.e-6
       maxLinearIterations = 128
       linearTolerance = 1.e-4
       preconditioner = ML
       #aztecPreconditioner = ilut
       stencilUpdater = [hyper]
       newtonForceMethod = Constant
```

```
    computePreconditioningMatrix = 0

    <TimeIntegrator implicitUpdater>
      kind = implicit$NDIM$d
      ongrid = domain
      scheme = theta
      theta = 0.5 #Crank-Nicholson
      noInitialGuess = true
    </TimeIntegrator>

    <Preconditioner myPreconditioner>
      kind=autoPreconditioner$NDIM$d
      # None/ML/AztecOO/Ifpack/New Ifpack
      preconditioner=ML
      # if 0, use a FD preconditioner
      computePreconditioningMatrix = 1
      # write out the preconditioning matrix at startup
      writePreconditioningMatrixToFile = 0
      testPreconditioner = false
      # maximum age of preconditioner in outer Newton steps (needs a better name)
      linearMaxPrecAge = 10
      # rebuild, reuse or recompute preconditioner (needs a better name)
      linearReusePolicy = Reuse
      mlStrategy=DD # SA/DD
      #mlSmoother="symmetric Gauss Seidel"
      stencilUpdater = [hyper]
      #mlDampingFactor=0.125
      mlSmoother = BSGS-E

      # Block to allow arbitray parameters to be passed into the preconditioner XML list
      #<ParameterList ml>
      #  increasing or decreasing = "increasing"
      #</ParameterList>
    </Preconditioner>

    <UpdateSequence sequence>
      loop = [boundaries,hyper]
    </UpdateSequence>

    <UpdateStep hyper>
       operation = "integrate"
       updaters = [hyper,periodicFlux,copyQnewToFlux]
       syncVars = [qNew]
    </UpdateStep>

    <UpdateStep boundaries>
       updaters = [periodicQ]
       syncVars = [q]
    </UpdateStep>

  </Updater>
```

8.22 localOdeIntegrator (1d, 2d, 3d)

The *localOdeIntegrator* is used to integrate a *Algebraic Equations* and evaluates a set of input *nodalArray*s and stores the ouput in one or more user-specified *nodalArray*s. The number of inputs and outputs are defined by

the kind of *Algebraic Equations* being used for the **Equation**.

8.22.1 Data

in (string vector, required) Inputs 1 to N are *nodalArray*s which will be supplied to the source through the **Equation** block.

out (string vector, required) Outputs 1 to N are *nodalArray*s which will contain the output of the Source.

8.22.2 Parameters

relativeErrorTolerance (float) The allowable error.

integrationScheme Integration scheme to use. Allowable types are *bulirschStoer*, *rk5* and *rosenbrock*. The integrations schemes are described in the boost, odeint library.

8.22.3 Sub-Blocks

Equation Defines the kind of source being solved. Equation in this case is actually a *kind* of *Algebraic Equations*. If multiple <Equation> blocks are defined then the results are added together to produce the output.

8.22.4 Example

The following code block demonstrates the usage of the‘localOdeIntegrator combined with the *exprHyperSrc* source:

```
<Updater integrator>
  kind = localOdeIntegrator1d
  integrationScheme = bulirschStoer
  onGrid = domain

  relativeErrorTolerance = 0.1

  in = [q]
  out = [qnew]

  <Equation gravity>
   kind = exprHyperSrc
   indVars = ["a","b"]
   exprs = ["-x*b","x*a"]
  </Equation>

</Updater>
```

8.23 timeStepRestrictionUpdater (1d, 2d, 3d)

The *timeStepRestrictionUpdater* computes the minimum time step and fastest wave speed based on specified restrictions. This data can both be used to determine the time step that the simulation will be advanced over and (optionally) store this data in a *dynVector* that can be passed to, e.g *Time Integrator* or *classicMusclUpdater (1d, 2d, 3d)*.

8.23.1 Data

in (string vector, required) Inputs 1 to N are *nodalArray*s which will be supplied to the time step restriction. Defined by the choice of *Time Step Restriction*.

timeSteps (string, optional) At most **one** *dynVector* that will contain the time step associated with each of the *Time Step Restriction* blocks. The *dynVector* must have the same number of components as the number of *Time Step Restriction* blocks.

waveSpeeds (string, optional) At most **one** *dynVector* that will contain the fastest wave speed associated with each of the *Time Step Restriction* blocks. The *dynVector* must have the same number of components as the number of *Time Step Restriction* blocks.

8.23.2 Parameters

courantCondition (float, required) The CFL condition to apply to the time-step restrictions computed by this updater.

restrictions (string vector, required) Names of 1 to N time step restrictions to compute using this updater. The names must correspond to the names of *Time Step Restriction* subblocks specified in this updater.

8.23.3 Sub-Blocks

TimeStepRestriction (block) The *Time Step Restriction* that defines the time-step and fastest wave speed to be computed by this updater. At least one *Time Step Restriction* must be specifed and This updater requires at least one TimeStepRestriction block. Each of the block names used should be put into the *restrictions* list.

8.23.4 Example

The following block demonstrates the twoTemperatureMhdDednerEqn used in combination with *timeStepRestrictionUpdater (1d, 2d, 3d)*, *hyperbolic (1d, 2d, 3d)* and *quadratic (1d, 2d, 3d)* to compute $dt_{\min}$, dt_{diff} and c_{fast} for resistive two-temperature MHD:

```
 <Updater getHypDT>
  kind = timeStepRestrictionUpdater1d
   in = [q,electricField,current,chargeState,resistivity]
  onGrid = domain
  waveSpeeds = [waveSpeed]
  timeSteps = [diffDT]
  restrictions = [idealMhd,quadratic]
  courantCondition = CFL

  <TimeStepRestriction idealMhd>
    kind = hyperbolic1d
    cfl = CFL
    model = twoTemperatureMhdDednerEqn
    gasGamma = GAS_GAMMA
    electronGamma = $ELECTRON_GAMMA$
    correctNans = true
    correct = true
    correctNans = true
    basementDensity = $BASEMENT_DENSITY$
    basementPressure = $BASEMENT_PRESSURE$
    externalEfield = "electricField"
```

```
    currentVector = "current"
    storeTimeStep = False
  </TimeStepRestriction>

  <TimeStepRestriction quadratic>
    kind = quadratic1d
    in = [resistivity]
    cfl = CFL
  </TimeStepRestriction>
</Updater>
```

The following Updater *kind* attributes can be specified to perform operations on the grid:

8.24 boundaryEntityGenerator (1d, 2d, 3d)

Generates ghost cells for a particular entity so that an updater requiring ghost cells can be used on a subset of the full domain.

8.24.1 Parameters

`onEntity` (string) The entity which will be used to construct ghost layers around. This can be considered the interior region

`boundaryName` (string) The name of the new entity that contains the boundary ghost cells of the interior region

`layers` (integer) The number of ghost layers to define

8.24.2 Example

```
<Updater generateVacuumBoundary>
  kind = boundaryEntityGenerator2d
  onGrid = domain
  layers = 2
  boundaryName = vacuumBoundary
  onEntity = vacuum
</Updater>
```

8.25 characteristicCellLength (1d, 2d, 3d)

Computes a characteristic cell length scale for every cell in the domain. This length scale is similar to the shortest side side length of a cell and can be used to determine maximum stable diffusion terms.

8.25.1 Data

`out` (string vector, required) The output is a *nodalArray* containing the characteristic length for each cell in the domain. This value is computed from geometry alone so no Inputs are required.

8.25.2 Parameters

coefficient (float) A constant factor that is multiplied by every component of the output vector

8.25.3 Example

```
<Updater computeCellDx>
  kind = characteristicCellLength2d
  onGrid = domain
  out = [cellDx]
  coefficient = 1.0
</Updater>
```

8.26 entityGenerator (1d, 2d, 3d)

Generates entities which can be used in defining regions for boundary conditions.

8.26.1 Parameters

onEntity (string) The entity generator will be a subset of the entity reference by onEntity. For boundary conditions this *onEntity* = *ghost* or a subset of *ghost*

newEntityName (string) The name of the newEntity that can then be used in updaters that are applied to entities.

8.26.2 Sub-Blocks

Function (block) Function which defines the entity. Wherever the function is positive and belongs the entity *onEntity* is defined the entity newEntityName will be defined.

8.26.3 Example

```
<Updater generateOpen>
  kind = entityGenerator2d
  onGrid = domain
  newEntityName = openBoundary
  onEntity = ghost
  <Function mask>
    kind = exprFunc
    exprs = ["if( (x>0.001) and (y>0.34),1.0,-1.0)"]
  </Function>
</Updater>
```

8.27 minDistanceToWall (1d, 2d, 3d)

Calculates the shortest distance from the grid boundary to each cell in the domain.

8.27.1 Parameters

entity (string) The name of the grid boundary

computeUt (boolean) Specify whether to compute the shear velocity. If true, *out* vector must be of size 2.

8.27.2 Data

in (string vector of 3, required)

- **input 1** is fluid vector
- **input 2** is gradient of the velocity-magnitude
- **input 3** is laminar viscosity

out (string vector, required) *out* is a single vector consisting of the shortest distance in component-1. If *computeUt=true*, shear velocity will be stored in component-2.

8.27.3 Example

```
<Updater computeDistancesToPlate>
  kind = minDistanceToWall2d
  onGrid = domain
  entity = plate
  computeUt = true
  in = [q, gradU, visc]
  out = [distance]
</Updater>
```

8.28 operatorEntityGenerator (1d, 2d, 3d)

Performs logical operations (not, and, or) on a list of entities to produce a new entity.

8.28.1 Data

entities (string vector) A list of entities that the logical operation will be performed on to produce the new entity.

8.28.2 Parameters

entityName (string) The name of the new entity resulting from the logical operations.

operation (string) The operation to be performed on the list of entities. The operation can have values (not, and, or).

8.28.3 Example

```
<Updater generatePlasma>
  kind = operatorEntityGenerator2d
  onGrid = domain
  operation = not
  entityName = plasma
  entities = [vacuum]
</Updater>
```

8.29 paintEntity (1d, 2d, 3d)

Sets the output variable to 0 everywhere the entity is not defined and 1 everywhere it is defined. This is useful for debugging boundary conditions.

8.29.1 Data

out (string vector) A *nodalArray* where the domain of the entity is stored. If you are looking at boundary conditions make sure to set writeHalos=true in the grid.

8.29.2 Example

```
<Updater init>
  kind = paintEntity2d
  onGrid = domain
  out = [q]
  entity = ghost
</Updater>
```

The following Updater *kind* attributes can be specified to compute output diagnostics from a simulation:

8.30 binCells (1d, 2d, 3d)

Initializes a *bin* dataStruct by storing cell data inside the *bin* structure. Every cell partially or completely inside the bin is stored in the bin.

8.30.1 Data

out (string vector) output *bin* where the cell binning data is stored

8.30.2 Example

```
<Updater fillBin>
  kind = binCells2d
  onGrid = domain
  out = [cellBin]
</Updater>
```

8.31 fieldAtPoint (1d, 2d, 3d)

Record a field at a particular point in space

8.31.1 Data

in (string vector, required)

Sampled Data The first variable is the data we will be sampling

bin The second variable is the bin used to determine what cell the point is located in.

out (string vector, required) Output *dynVector* where the result of the operation is stored

8.31.2 Parameters

point (vector float) This is the point where the data will be computed

inpIndices (vector integer) These are the indexes of the input array that will be stored in the output vector

8.31.3 Example

```
<Updater computeValues>
  kind = fieldAtPoint1d
  onGrid = domain
  point = [-0.25,0.0,0.0]

  inpIndices = [0, 1, 2]
  in = [em, cellBin]

  out = [E]
</Updater>
```

8.32 intCombinedFields (1d, 2d, 3d)

Integrates a quantity over the volume of the domain and writes to a *dynVector*

8.32.1 Data

in (string vector) Input 1 to N are input nodalArrays on which operations will be performed. Example in = [E, B]

out (string vector) output *dynVector* where the result of the operation is stored

8.32.2 Parameters

indVars_name (string vector) For each input variable an "indVars" array must be defined. So if in = [E, B] then indVars_E and indVars_B must be defined. If indVars_E = ["Ex","Ey","Ez"] then operations are performed on "Ex","Ey" and "Ez" in the expression evaluator.

preExprs (string vector) Strings must be put in quotes. The preExprs is used to compute quantities based on indVars that can later be used in the *exprs* to evaluate the output. Available commands are defined by the muParser (http://muparser.sourceforge.net)

exprs (string vector) Strings must be put in quotes. The strings are evaluated and placed in the output array. Available command are defined by the muParser (http://muparser.sourceforge.net/)

other (variable definition) In addition, an arbitrary number of constants can be defined that can then be used in evaluating expression in both *preExprs* and *exprs*

Also, the updater has predefined variables including x,y,z representing the spatial location of the cell and t the time.

8.32.3 Example

```
<Updater computeTotalE>
  kind = intCombinedFields2d
  onGrid = domain

  in = [qnew]
  out = [totalE]
  mi = MI
  mu0 = MU0
  gamma = GAMMA
  k=KB
  indVars_qnew = ["rho","mx","my","mz","en","bx","by","bz","phi"]
  exprs = ["en"]
</Updater>
```

8.33 lineIntegral (1d, 2d, 3d)

Performs operations on a set of input *nodalArray* to produce a *dynVector* by integrating along a specified trajectory. Uses an expression updater to evaluate the expression. The expression evaluator recognizes positions "x","y","z" and time "t" and these can be used to evaluate functions of time and space.

8.33.1 Data

in (string vector) Input 1 to N are input *nodalArray* on which operations will be performed. Example in = [E, B]

out (string) output *dynVector* where the result of the operation is stored

layout (string) The name of the *bin* to use when constructing the line integral. The line integral requires that a *bin* be constructed. The updater *binCells (1d, 2d, 3d)* can be used to initialize a *bin* dataStruct.

8.33.2 Parameters

startPosition The starting position of the line integral

endPosition The end position of the line integral

numberOfSamples The number of sample points along the line to be used in computing the line integral

indVars_inName For each input variable an "indVars" array must be defined. So if in = [E, B] then indVars_E and indVars_B must be defined. If indVars_E = ["Ex","Ey","Ez"] then operations are performed on "Ex","Ey" and "Ez" in the expression evaluator.

preExprs (string vector) Strings must be put in quotes. The preExprs is used to compute quantities based on indVars that can later be used in the *exprs* to evaluate the output. Available commands are defined by the muParser (http://muparser.sourceforge.net)

exprs (string vector) Strings must be put in quotes. The strings are evaluated and placed in the output array. Available command are defined by the muParser (http://muparser.sourceforge.net/)

other (variable definition) In addition, an arbitrary number of constants can be defined that can then be used in evaluating expression in both *preExprs* and *exprs*

Also, the combiner has predefined variables including x,y,z representing the spatial location of the cell, t and dt, representing time and time step and dVolume representing the volume of a cell.

8.33.3 Example

```
<Updater computeLineIntegral>
  kind = lineIntegral2d
  onGrid = domain
  startPosition = [0.0, 0.0]
  endPosition = [0.5, 0.5]
  numberOfSamples = 100

  layout = [cellBin]
  in = [potential]
  indVars_potential = ["phi"]
  exprs = ["phi"]
  out = [lineIntegralPhi]
</Updater>
```

8.34 maxCombinedFields (1d, 2d, 3d)

Computes the maximum value (pressure or energy for example) over the domain. The value is stored in a *dynVector*.

8.34.1 Data

in (string vector) Input 1 to N are input nodalArrays on which operations will be performed. Example in = [E, B]

out (string vector) output *dynVector* where the result of the operation is stored

8.34.2 Parameters

indVars_name (string vector) For each input variable an "indVars" array must be defined. So if in = [E, B] then indVars_E and indVars_B must be defined. If indVars_E = ["Ex","Ey","Ez"] then operations are performed on "Ex","Ey" and "Ez" in the expression evaluator.

preExprs (string vector) Strings must be put in quotes. The preExprs is used to compute quantities based on indVars that can later be used in the *exprs* to evaluate the output. Available commands are defined by the muParser (http://muparser.sourceforge.net)

exprs (string vector) Strings must be put in quotes. The strings are evaluated and placed in the output array. Available command are defined by the muParser (http://muparser.sourceforge.net/)

other (variable definition) In addition, an arbitrary number of constants can be defined that can then be used in evaluating expression in both *preExprs* and *exprs*

Also, the updater has predefined variables including x,y,z representing the spatial location of the cell and t the time.

8.34.3 Example

```
<Updater computeMaxP>
 kind = maxCombinedFields2d
 onGrid = domain

 in = [qnew]
 out = [maxP]
 mi = MI
 mu0 = MU0
 gamma = GAMMA
 k=KB
 indVars_qnew = ["rho","mx","my","mz","en","bx","by","bz","phi"]
 exprs = ["(gamma-1)*(en-(0.5/mu0)*(bx*bx+by*by+bz*bz)-0.5*(mx*mx+my*my+mz*mz)/rho)"]
</Updater>
```

8.35 surfaceIntegral (1d, 2d, 3d)

Computes a surface integral as a function of *nodalArray* values and dumps the results in a *dynVector*.

8.35.1 Data

in (string vector) Input 1 to N are input *nodalArray* on which operations will be performed. Example *in = [E, B]*

out (string vector) The *dynVector* where the result of the surface integral is stored.

8.35.2 Parameters

onEntity (string) Name of the entity on which the surface integral will be applied.

indVars_name (string vector) For each input variable an "indVars" array must be defined. So if in = [E, B] then indVars_E and indVars_B must be defined. If indVars_E = ["Ex","Ey","Ez"] then operations are performed on "Ex","Ey" and "Ez" in the expression evaluator. This expression evaluator takes, "x", "y", "z", "NormalX", "NormalY" and "NormalZ" as parameters, where "Normal" are the component normals to the surface location.

preExprs (string vector) Strings must be put in quotes. The preExprs is used to compute quantities based on indVars that can later be used in the *exprs* to evaluate the output. Available commands are defined by the muParser (http://muparser.sourceforge.net)

exprs (string vector) Strings must be put in quotes. The strings are evaluated and placed in the output array. Available command are defined by the muParser (http://muparser.sourceforge.net/)

other (variable definition) In addition, an arbitrary number of constants can be defined that can then be used in evaluating expression in both *preExprs* and *exprs*

Also, the updater has predefined variables including x,y,z representing the spatial location of the surface, t representing time and NormalX, NormalY and NormalZ representing the surface normal to the boundary.

8.35.3 Example

```
<Updater computeSurfaceCurrent>
  kind = surfaceIntegral2D
  onGrid = domain
  entity = ghost

  length = LENGTH
  in = [J]
  out = [surfaceCurrent]

  indVars_J = ["Jx","Jy","Jz"]
  exprs = ["2*3.14159*x*Jx"]
</Updater>
```

8.36 surfaceVariables (1d, 2d, 3d)

Obtains the variables such as temperature, heat flux, evaporation flux etc at the solid fluid interface. The interface can be specified using domain boundary entities of interest. The standard updater block is given below:

```
<Updater SurfaceVariable>
 kind = surfaceVariables2d
 onGrid = domain
 variablesType = ablation
 storeSurfaceProperty = 1

 in = [surfTemp]
 out = [abSurfProp]

 entity = left
</Updater>
```

The parameters required by this updater block are listed below:

`storeSurfaceProperty` (boolean)

Store the surfaceVariables in the first row of cells along the boundary entity

`entity` (string)

Name of the boundary on which *surfaceVariables* have to be evaluated.

`variablesType` (string)

The type of surface variable to compute. Currently implemented surfaceVariables are

8.36.1 ablation (1d, 2d, 3d)

Computes the surface evaporation parameters for a given material. The description of the parameters specific to ablation is given blelow:

Parameters

`variablesType` = ablation

`storeSurfaceProperty` = 1. This should be *true* to vizulaize and utilize the evaluated variables in boundary conditions

`in` = vector containing the surface temperature

`ablationModel` option to chose the type of ablation model. currently implemented model is *sonic*, which assumes the vapor expands to sonic speed at the fluid interface.

`numConstituents` (integer) is the number of material elements present inside the compound material.

`satPressure` (real) The material specific variables of Claussius-Clapeyron equation to obtain the saturation pressure at a given temperature. Each element requires 3 constants reference pressure (Pa), enthalpy of evaporation (J/mol) and reference temperature (K). If there are two elements in the cmpound material, the constants of the second element should be entered right after the first element.

`moleFraction` (real) Molefractions of the *constiTuents*

`averageMolecularWeight` (real) average molecular weight of the compound material

`out` the result vector consisting of Density (kg/m^3), velocity (m/s), temperature (K), pressure (Pa), saturation pressure (Pa), and number density $(1/m^3)$ of the compound material.

Example

Code block

```
 <Updater computeAbSurfProp>
  kind = surfaceVariables2d
  onGrid = domain
  variablesType = ablation
  storeSurfaceProperty = 1
  dynVectors = []

  in = [surfTemp]

  ablationModel = sonic
  numConstituents = 2
  satPressure = [p01 dh1 T01 p02 dh2 T02]
  moleFraction = [MolF1 MolF2]
  averageMolecularWeight = MWAvg

  out = [abSurfProp]

  entity = left
</Updater>
```

8.36.2 temperatureAndHeatFlux (1d, 2d, 3d)

Evaluates the temperature and heatflux on the boundary of interest. The description of specific parameters is given blelow:

Parameters

`variablesType` = temperatureAndHeatFlux

`storeSurfaceProperty` = 1. This should be *true* to vizulaize and utilize the evaluated variables in boundary conditions

`in` = vectors containing the thermal conductivity and temperature gradient. Note that temperature gradient vector will have three components.

`heatFluxBalanceModel` option to chose the type of heat flux balance. currently implemented model is *radiationEquilibrium*.

`emissivity` (real) is the surface emissivity.

`baseTemperature` (real) minimum possible surface temperature.

`averageMolecularWeight` (float) average molecular weight of the compound material

`out` the result vector consisting of surface temperature (K) and surface heat flux (W/m^2)

Example

Code block

```
 <Updater computeSurfTemp>
  kind = surfaceVariables2d
  onGrid = domain
  variablesType = temperatureAndHeatFlux
  storeSurfaceProperty = 1

  dynVectors = []
  in = [D,gradTemp]

  heatFluxBalanceModel = radiationEquilibrium
  emissivity = 0.9
  baseTemperature = BASETEMP

  out = [surfTemp]

  entity = ghost
</Updater>
```

The following Updater *kind* attributes can be specified to *fix* unphysical behaviour (e.g. NaN, negative density, pressures) in a simulation:

8.37 nanChecker (1d, 2d, 3d)

Checks a nodalArray to see if any of the numbers are undefined nan, inf etc... Throws an exception of a nan is found and provides the index.

8.37.1 Data

in The *nodalArray* that will be searched for nans

8.37.2 Example

```
<Updater init>
  kind = nanChecker2d
  onGrid = domain
  out = [q]
</Updater>
```

8.38 pressureDensityCorrector (1d, 2d, 3d)

Computes the pressure and density in a nodalArray and modifies the pressure and density if they are below basement values. This is a simple way to prevent pressures and densities from becoming too small but is also non-conservative.

8.38.1 Data

out (string vector) Output 1 stores the *nodalArray* that will have its pressure and density corrected. The nodalArray must have the same number of components as is required by the chosen model.

8.38.2 Parameters

model (string) The model equation used for determining how to compute pressure and density. The model must be a fluid model such as eulerEqn. Will not work with maxwellEqn since no pressure or density is defined. When the model is initialized it will request additional variables required by that model, for example *gasGamma* and *mu0* for MHD type equations.

basementDensity (float) basementDensity used in determining when to switch between accurate and positive solutions. Default is 0.0.

basementPressure (float) basementPressure used in determining when to switch between accurate and positive solutions. Default is 0.0.

8.38.3 Example

```
<Updater correct>
  kind = pressureDensityCorrector2d
  model = eulerEqn
  basementDensity = BASEMENT_DENSITY
  basementPressure = BASEMENT_PRESSURE
  gasGamma = GAS_GAMMA
  onGrid = domain
  out = [q]
</Updater>
```

8.39 valueCorrector (1d, 2d, 3d)

Cleans a nodalArray of "bad" values. If a value in the array is below a basment value or is undefined (nan,inf etc), the value corrector sets the value to the basement value. Cleans every value in the output vector so if your output is length 6 it cleans all 6 components.

8.39.1 Data

`out` (string vector) The variable that will be modified

8.39.2 Parameters

`basementValue` (float) Minimum value the nodalArray is allowed to have. If the value drops below this then it will be reset to basementValue

8.39.3 Example

```
<Updater init>
  kind = valueCorrector1d
  onGrid = domain
  out = [q]
  basementValue = 1.0
</Updater>
```

CHAPTER

NINE

TIME INTEGRATOR

Time integrators in USim allow updaters such as *multiUpdater (1d, 2d, 3d)* and *implicitMultiUpdater (1d, 2d, 3d)* to discretize partial differential equations in time. USim provides support for total variation diminsighing explicit Runge-Kutta schemes at up to fourth order; super-time-step schemes at first and second order, subcycling methods and implicit discretizations at up to second order.

An example demonstrating an explicit third order Runge-Kutta scheme is below:

```
<TimeIntegrator rkIntegrator>
  kind = rungeKutta1d
  ongrid = domain
  scheme = third
</TimeIntegrator>
```

The following parameters are common to all *TimeIntegrator* blocks:

`kind` (string, required) Specifies the time-integration scheme to use: Available options are:

`rungeKutta(1d,2d,3d)` Specifices explicit Runge Kutta integration methods in 1, 2 or 3 dimensions. Appropriate for hyperbolic problems.

`superTimeStep(1d,2d,3d)` Specifices explicit super time step integration methods in 1, 2 or 3 dimensions. Appropriate for diffusion problems.

`implicit(1d,2d,3d)` Specifices implicit integration methods in 1, 2 or 3 dimensions.

`onGrid` (string, required) The *Grid* the time integration is performed on.

`scheme` (string, required) The order of the time integration method to use. Available options are:

`None` Do not integrate in time. Only available for *kind = implicit(1d,2d,3d)*. Used for solving problems that are not discretized in time, e.g. Poisson's equation.

`theta` Only available for *kind = implicit(1d,2d,3d)*. Provides an implicit discretization of the form

$$\mathbf{q}^{n+1} - \mathbf{q}^n - \Delta t \theta \left[\nabla \cdot \mathcal{F}\left(\mathbf{w}^{n+1}\right) - \mathcal{S}\left(\mathbf{w}^{n+1}\right)\right] - \Delta t \left[1-\theta\right]\left[\nabla \cdot \mathcal{F}\left(\mathbf{w}^{n}\right) - \mathcal{S}\left(\mathbf{w}^{n}\right)\right] \text{`}$$

Here, $\theta = 1$ corresponds to backwards Euler, $\theta = 1/2$ corresponds to Crank-Nicholson and $\theta = 0$ corresponds to forward Euler. I

`zeroth` First order subcycling scheme. Only available for *kind = superTimeStep(1d,2d,3d)*.

`first` First order accurate schemes. Only available for *kind = rungeKutta(1d,2d,3d)*, *kind = superTimeStep(1d,2d,3d)*.

`second` Second order accurate schemes. Only available for *kind = rungeKutta(1d,2d,3d)*, *kind = superTimeStep(1d,2d,3d)*.

`third` Third order accurate schemes. Only available for *kind = rungeKutta(1d,2d,3d)*.

`fourth` Fourth order accurate schemes. Only available for *kind = rungeKutta(1d,2d,3d)*.

`timeStepRestrictions` (string vector, optional) List of *dynVector* that holds the timestep associated with the diffusion operator that forms the right-hand side of the equation. Required if *kind* = *superTimeStep(1d,2d,3d)*.

`theta` (float, optional) Specifies θ for implicit discretizations. Required if *kind* = *implicit(1d,2d,3d)* and *scheme* = *theta*.

CHAPTER

TEN

PRECONDITIONER

Preconditioner blocks are used in combination with *implicitMultiUpdater (1d, 2d, 3d)*. They allow USim to solve linear systems in an efficient, scalable fashion. An example *Preconditioner* block is given below

```
<Preconditioner myPreconditioner>
 preconditioner = ML # None/ML/AztecOO/Ifpack/New Ifpack
 computePreconditioningMatrix = 1 # if 0, use a FD preconditioner
 writePreconditioningMatrixToFile = 0 # write out the preconditioning matrix at startup
 linearMaxPrecAge = 10 # maximum age of preconditioner in outer Newton steps
 linearReusePolicy = Reuse # rebuild, reuse or recompute preconditioner
 stencilUpdater = [computeNablaPhi]
 mlStrategy=classicSA # SA/DD
</Preconditioner>
```

The following parameters are common to all *Preconditioner* blocks.

`kind` (string, required) Specify the method for computing the matrix for preconditioning the linear system. Available options are:

> `preconditioner(1,2,3)d` With this choice, USim computes the matrix through the stencil supplied by a single updater specied by the *stencilUpdater* parameter. This option is useful when the operator to be solved has a simple signature (e.g. the Laplacian ∇^2)
>
> `autoPreconditioner(1,2,3)d` With this choice, USim uses an efficient finite difference method to compute the matrix for the system of equations specified by the *UpdateSequence* block in the *implicitMultiUpdater (1d, 2d, 3d)*. This option is useful for systems that solve multiphysics problems.

`preconditioner` (string, required) Options are None, ML, AztecOO, Ifpack and New Ifpack. ML (Multi-Level) preconditioners are the preferred option for USim due to the highly anisotropic nature of the matrix produced by USim operators. These preconditioner are based on the ML package (https://trilinos.org/packages/ml/). Other options include preconditioners based on the AztecOO package (https://trilinos.org/packages/aztecoo/) and Ifpack (https://trilinos.org/packages/ifpack/)

`computePreconditioningMatrix` (int, required) If computePreconditioningMatrix = 1, then compute a matrix based on a user-specified updater (if *kind = preconditioner(1,2,3)d*) or using an efficient finite difference method (if *kind = autoPreconditioner(1,2,3)d*. If computePreconditioningMatrix = 0, then the matrix is determined using a (slow) finite-difference computation. This latter option, allows for debugging the system of equations.

`writePreconditioningMatrixToFile` (int, required) If writePreconditioningMatrixToFile = 1, then the matrix used to precondition the non-linear problem is written out each time it is filled in Matrix Market format. This option is expensive, both in terms of simulation time and storage space and so it is recommended that writePreconditioningMatrixToFile = 0 except if needing to debug the simulation.

`stencilUpdater` (string vector, optional) Tell the solver which updater to compute the matrix to use as a preconditioner. Required if *kind = preconditioner(1,2,3)d* and *computePreconditioningMatrix = 1*. Currently, only

accepts one entry.

linearMaxPrecAge (int, required) The number of outer Newton steps to take between each update of the preconditioner. Determining this value is problem dependent and requires careful experimentation by the user.

The following options are available if *preconditioner* = *ML*:

testPreconditioner (bool) Test the ability of the preconditioner to invert the matrix

testSmoother (bool) Test the ability of the range of smoothers available in ML to invert the matrix.

mlStrategy (string) Determines whether or not to use smoothed aggregation or domain decomposition for the multi-level solver. Available options are:

`SA` Specify smoothed aggregation methods.

`DD` Specify domain decomposition methods.

`classicSA` Specify smoothed aggregation methods appropriate for diffusion-type problems.

`classicDD` Specify domain decomposition methods appropriate for diffusion-type problems.

mlSmoother (string) Specify the smoother strategy used to compute the ML hierarcy when Options include:

```
Jacobi
block Gauss-Seidel
symmetric Gauss-Seidel
BSGS-A
BSGS-E
```

The choice of smoother is best determined for a given problem by first running the problem with *testPreconditioner* = *true* and *testSmoother* = *true*. This combination of options will provide information about the ability of the different *mlSmoother* options to solve the matrix. The *mlSmoother* option can then be set appropriately and the *testPreconditioner*, *testSmoother* options can be set to *false* in order to improve efficiency.

mlNumPDE (int, optional) Specify the number of PDE's represented in the matrix. Typically, this option should match the number of components for the input *nodalArray* input to the *implicitMultiUpdater (1d, 2d, 3d)*.

The following subblocks can be supplied to the preconditioner:

ParameterList A *ParameterList* block can be supplied to any preconditioner block. The *ParameterList* block should contain options accepted by the *Trilinos* preconditioner specified by the *preconditioner* string parameter, documented at https://trilinos.org/packages/ml/, https://trilinos.org/packages/aztecoo/) and https://trilinos.org/packages/ifpack/

CHAPTER

ELEVEN

HYPERBOLIC EQUATIONS

An **Equation** block that describes a hyperbolic conservation law of the form:

$$\frac{\partial \mathbf{q}}{\partial t} + \nabla \cdot \left[\mathcal{F}\left(\mathbf{w}\right)\right] = 0$$

where $\mathbf{q}$ is a vector of conserved variables (e.g. density, momentum, total energy), $\mathcal{F}\left(\mathbf{w}\right)$ is a non-linear flux tensor computed from a vector of primitive variables, (e.g. density, velocity, pressure), $\mathbf{w} = \mathbf{w}(\mathbf{q})$. The choice of hyperbolic equation defines $\mathbf{q}$, $\mathcal{F}\left(\mathbf{w}\right)$, $\mathbf{w} = \mathbf{w}(\mathbf{q})$, along with the eigensystem associated with $\mathcal{F}\left(\mathbf{w}\right)$.

An *Equation* block is owned by an updater (e.g. *classicMusclUpdater (1d, 2d, 3d)*). The updater that owns the *Equation* sets the input, output and any additional data structures that are required by the Equation system.

The following parameters are common to all *Equation* blocks:

`kind` (string) All *Equation* blocks take a string *kind* that species the type of hyperbolic equation. The following equations can be used to simulate neutral plasmas:

11.1 eulerEqn

Defines the equations of inviscid compressible hydrodynamics:

$$\begin{aligned}
\frac{\partial \rho}{\partial t} + \nabla \cdot \left[\rho\, \mathbf{u}\right] &= 0 \\
\frac{\partial \rho\, \mathbf{u}}{\partial t} + \nabla \cdot \left[\rho\, \mathbf{u}\, \mathbf{u}^T + \mathbb{I} P\right] &= 0 \\
\frac{\partial E}{\partial t} + \nabla \cdot \left[\left(E + P\right) \mathbf{u}\right] &= 0
\end{aligned}$$

Here, $\mathbb{I}$ is the identity matrix, $P = \rho\epsilon(\gamma - 1)$ is the pressure of an ideal gas, ϵ is the specific internal energy and γ is the adiabatic index (ratio of specific heats).

11.1.1 Parameters

`gasGamma` (float) Specifies the adiabatic index (ratio of specific heats), γ. Defaults to 5/3.

`basementPressure` (float, optional) The minimum pressure allowed. Pressures below this value will be replaced with this value for primitive state, eigensystem and flux computations. Defaults to zero.

`basementDensity` (float, optional) The minimum density allowed. Densities below this value will be replaced with this value for primitive state, eigensystem and flux computations. Defaults to zero.

11.1.2 Parent Updater Data

`in` (string vector, required)

`Vector of Conserved Quantities` (*nodalArray*, 5-components, required) The vector of conserved quantities, $\mathbf{q}$ has 5 entries:

0. ρ: mass density
1. $\rho\, u_{\hat{\mathbf{i}}} = \rho\mathbf{u}\cdot\hat{\mathbf{i}}$: momentum density in the $\hat{\mathbf{i}}$ direction
2. $\rho\, u_{\hat{\mathbf{j}}} = \rho\mathbf{u}\cdot\hat{\mathbf{j}}$: momentum density in the $\hat{\mathbf{j}}$ direction
3. $\rho\, u_{\hat{\mathbf{k}}} = \rho\mathbf{u}\cdot\hat{\mathbf{k}}$: momentum density in the $\hat{\mathbf{k}}$ direction
4. $E = \frac{P}{\gamma-1} + \frac{1}{2}\rho|\mathbf{u}|^2$: total energy density

`out` (string vector, required) For the eulerEqn, one of four output variables are computed, depending on whether the equation is combined with an updater capable of computing fluxes (*classicMusclUpdater (1d, 2d, 3d)*), primitive variables (*computePrimitiveState(1d, 2d, 3d)*), the time step associated with the CFL condition (*timeStepRestrictionUpdater (1d, 2d, 3d)*) or the fastest wave speed in the grid (*timeStepRestrictionUpdater (1d, 2d, 3d)*).

`Vector of Fluxes` (*nodalArray*, 5-components) When combined with an updater that computes $\nabla\cdot\mathcal{F}(\mathbf{w})$ (e.g. *classicMusclUpdater (1d, 2d, 3d)*), the equation system returns:

0. $\nabla\cdot\mathcal{F}(\rho)$: mass flux
1. $\nabla\cdot\mathcal{F}\left(\rho\, u_{\hat{\mathbf{i}}}\right)$: $\hat{\mathbf{i}}$ momentum flux
2. $\nabla\cdot\mathcal{F}\left(\rho\, u_{\hat{\mathbf{j}}}\right)$: $\hat{\mathbf{j}}$ momentum flux
3. $\nabla\cdot\mathcal{F}\left(\rho\, u_{\hat{\mathbf{k}}}\right)$: $\hat{\mathbf{k}}$ momentum flux
4. $\nabla\cdot\mathcal{F}(E)$: total energy flux

`Vector of Primitive States` (*nodalArray*, 5-components) When combined with an updater that computes $\mathbf{w} = \mathbf{w}(\mathbf{q})$ (e.g. *computePrimitiveState(1d, 2d, 3d)*), the equation systen returns:

0. ρ: mass density
1. $u_{\hat{\mathbf{i}}} = \mathbf{u}\cdot\hat{\mathbf{i}}$: velocity in the $\hat{\mathbf{i}}$ direction
2. $u_{\hat{\mathbf{j}}} = \mathbf{u}\cdot\hat{\mathbf{j}}$: velocity in the $\hat{\mathbf{j}}$ direction
3. $u_{\hat{\mathbf{k}}} = \mathbf{u}\cdot\hat{\mathbf{k}}$: velocity in the $\hat{\mathbf{k}}$ direction
4. $P = \rho\epsilon(\gamma-1)$: ideal gas pressure

`Time Step` (*dynVector*, 1-component) When combined with the kind=hyperblic, model=eulerEqn *timeStepRestrictionUpdater (1d, 2d, 3d)*, and storeTimeStep is true, the equation system returns the time step consistent with the CFL condition across the entire simulation domain.

`Fastest Wave Speed` (*dynVector*, 1-component) When combined with the kind=hyperbolic, model=eulerEqn *timeStepRestrictionUpdater (1d, 2d, 3d)*, and storeWaveSpeed is true, the equation system returns the fastest wave speed across the entire simulation domain, c_{fast}.

11.1.3 Example

The following block demonstrates the eulerEqn used in combination with *classicMusclUpdater (1d, 2d, 3d)* to compute $\nabla\cdot\mathcal{F}(\mathbf{w})$:

```
<Updater hyper>
  kind=classicMuscl1d
  onGrid=domain
  timeIntegrationScheme=none
  numericalFlux=roeFlux
  limiter=[muscl]
  variableForm=primitive
  in=[q]
  out=[qnew]
  cfl=0.3
  equations=[euler]

  <Equation euler>
    kind=eulerEqn
    gasGamma=1.4
    basementDensity = 1.0e-5
    basementPressure = 1.0e-6
  </Equation>

</Updater>
```

11.2 realGasEqn

Real gas using a real gas equation of state. Requires the computation of specific heat and temperature and assignment of zero point energy outside of the equation. Assumes single temperature. The equations are solved in conservative form.

$$\frac{\partial}{\partial t}\begin{pmatrix}\rho\\ \rho\,u_x\\ \rho\,u_y\\ \rho\,u_z\\ e\end{pmatrix}+\nabla\cdot\begin{pmatrix}\rho\,u_x & \rho\,u_y & \rho\,u_z\\ \rho\,u_x^2+P & \rho\,u_x\,u_y & \rho\,u_x\,u_z\\ \rho\,u_y\,u_x & \rho\,u_y\,u_y+P & \rho\,u_y\,u_z\\ \rho\,u_z\,u_x & \rho\,u_z\,u_y & \rho\,u_z\,u_z+P\\ u_x\,(e+P) & u_y\,(e+P) & u_z\,(e+P)\end{pmatrix}=0$$

The energy is given by

$$e=\frac{1}{2}\rho\left(u_x^2+u_y^2+u_z^2\right)+\sum_i n_i\left(Cv_iT+e_{0\,i}\right) \tag{11.-2}$$

11.2.1 Parameters

`numSpecies` (float) The number of species modeled in the real gas system.

`basementPressure` (float) The minimum pressure allowed. Defaults to 0.

`basementDensity` (float) The minimum density allowed. Defaults to 0.

> **Note:** basementPressure and basementDensity are only used if correct=true

`correct` (boolean) Tells whether or not densities or pressures should be corrected when the fall below basement pressures or basement densities. When set to true pressure=max(basementPressure, pressure) and density = max(basementDensity, density). Defaults to false.

> **Note:** Setting correctNans or correct to true can lead to energy conservation errors

11.2.2 Parent Updater Data

in (string vector, required)

Vector of conserved quantities

0. ρ mass density
1. $\rho\, u_x$ x momentum density
2. $\rho\, u_y$ y momentum density
3. $\rho\, u_z$ z momentum density
4. e energy density

2nd variable (3n+1) 3n+1 auxiliary variables with n the number of species

0. variables 0-(n-1). n_i species number density
1. variables n-(2n-1). Cv_i species specific heat at constant volume
2. variables n-(3n-1). $e_{0\,i}$ species zero point energy density
3. variables 3n. T Temperature in Kelvin

11.2.3 Example

An example *realGas* equation block is given below

```
<Equation realGas>
  kind = realGasEqn
  numSpecies = 7
</Equation>
```

11.3 realGasEosEqn

Gas dynamics with a general equation of state. The equations are solved in conservative form.

$$\frac{\partial}{\partial t}\begin{pmatrix}\rho\\ \rho\,u_x\\ \rho\,u_y\\ \rho\,u_z\\ e\end{pmatrix}+\nabla\cdot\begin{pmatrix}\rho\,u_x & \rho\,u_y & \rho\,u_z\\ \rho\,u_x^2+P & \rho\,u_x\,u_y & \rho\,u_x\,u_z\\ \rho\,u_y\,u_x & \rho\,u_y\,u_y+P & \rho\,u_y\,u_z\\ \rho\,u_z\,u_x & \rho\,u_z\,u_y & \rho\,u_z\,u_z+P\\ u_x\,(e+P) & u_y\,(e+P) & u_z\,(e+P)\end{pmatrix}=0$$

11.3.1 Parameters

basementPressure (float) The minimum pressure allowed. Default is 0.

basementDensity (float) The minimum density allowed. Default is 0.

> **Note:** basementPressure and basementDensity are only used if correct=true

correct (boolean) Tells whether or not densities or pressures should be corrected when the fall below basement pressures or basement densities. When set to true pressure=max(basementPressure, pressure) and density = max(basementDensity, density)

11.3.2 Parent Updater Data

`in` (string vector, required)

`Vector of conserved quantities` (5 components)

0. ρ mass density
1. $\rho\, u_x$ x momentum density
2. $\rho\, u_y$ y momentum density
3. $\rho\, u_z$ z momentum density
4. e energy density

`fluid pressure` (1 component)

0. P total fluid pressure (not magnetic pressure included)

`gas dynamic sound speed` (1 component)

0. a estimate of the fluid sound speed

11.3.3 Example

An example *realGasEos* equation block is given below:

```
<Equation realGasEos>
  kind = realGasEosEqn
</Equation>
```

11.4 tenMomentEqn

Ideal compressible 10 moment fluid equations. The equations are solved in conservative form.

$$\frac{\partial}{\partial t}\begin{pmatrix} \rho \\ \rho\, u_x \\ \rho\, u_y \\ \rho\, u_z \\ \rho\, u_x^2 + P_{x\,x} \\ \rho\, u_x\, u_y + P_{x\,y} \\ \rho\, u_x\, u_z + P_{x\,z} \\ \rho\, u_y^2 + P_{y\,y} \\ \rho\, u_y\, u_z + P_{y\,z} \\ \rho\, u_z^2 + P_{z\,z} \end{pmatrix} + \nabla \cdot\ P = 0$$

where P is defined as

$$\begin{pmatrix} \rho\, u_x & \rho\, u_y & \rho\, u_z \\ \rho\, u_x^2 + P_{x\,x} & \rho\, u_x\, u_y + P_{x\,y} & \rho\, u_x\, u_z + P_{x\,z} \\ \rho\, u_y\, u_x + P_{x\,y} & \rho\, u_y\, u_y + P_{y\,y} & \rho\, u_y\, u_z + P_{y\,z} \\ \rho\, u_z\, u_x + P_{x\,z} & \rho\, u_z\, u_y + P_{y\,z} & \rho\, u_z\, u_z + P_{z\,z} \\ \rho\, u_x^3 + 3u_x P_{x\,x} & \rho\, u_y u_x^2 + u_x P_{y\,y} + 2u_x P_{x\,y} & \rho\, u_z u_x^2 + u_z P_{x\,x} + 2u_x P_{x\,z} \\ \rho\, u_x^2 u_y + 2\, u_x\, P_{x\,y} + u_y P_{x\,x} & 0 & 0 \\ \rho\, u_x^2 u_z + 2\, u_x\, P_{x\,z} + u_z P_{x\,x} & 0 & 0 \\ \rho\, u_x u_y^2 + u_x P_{y\,y} + 2u_y P_{x\,y} & \rho\, u_y^3 + 3u_y P_{y\,y} & 0 \\ \rho\, u_x u_y u_z + u_x P_{y\,z} + u_y P_{x\,z} + u_z P_{x\,y} & 0 & 0 \\ \rho\, u_x u_z^2 + u_x P_{z\,z} + 2u_z P_{x\,z} & 0 & \rho\, u_z^3 + 3u_z P_{z\,z} \end{pmatrix}$$

11.4.1 Parameters

`basementPressure` (float) The minimum pressure allowed. Defaults to 0.

`basementDensity` (float) The minimum density allowed. Defaults to 0.

11.4.2 Parent Updater Data

`in` (string vector, required)

1st variable

0. ρ mass density
1. $\rho\, u_x$ x momentum density
2. $\rho\, u_y$ y momentum density
3. $\rho\, u_z$ z momentum density
4. $\rho\, u_x^2 + P_{x\,x}$ xx energy density
5. $\rho\, u_x\, u_y + P_{x\,y}$ xy energy density
6. $\rho\, u_x\, u_z + P_{x\,z}$ xz energy density
7. $\rho\, u_y^2 + P_{y\,y}$ yy energy density
8. $\rho\, u_y\, u_z + P_{y\,z}$ yz energy density
9. $\rho\, u_z^2 + P_{z\,z}$ zz energy density

11.4.3 Example

An example *tenMoment* equation block is given below:

```
<Equation tenMoment>
  kind = tenMomentEqn
</Equation>
```

11.5 multiSpeciesSingleVelocityEqn

This equation represents continuity equations for n species. The species continuity equation is given by

$$\frac{\partial n_i}{\partial t} + \nabla_j \left(n_i \, u_j \right) = 0 \tag{11.-4}$$

11.5.1 Parameters

`basementNumberDensity` (float) The minimum species number density allowed

`basementDensity` (float) The minimum auxiliary variable mass density allowed. Defaults to 0.

`numberOfSpecies` (integer) The number of species that have continuity equations.

`useParentEigenvalues` (boolean) When set to true the eigenvalues of the parent system are used in computing dissipation in fluxes such as the localLaxFlux as well as time step restrictions. When set to false, the eigenvalue is simply u normal to the direction of interest.

11.5.2 Sub-Blocks

`Equation` (block) Defines the parent equation type of the system. The parent equation could be eulerEqn or idealMhdEqn for example. The first 4 components must be density, followed by the 3 components of momentum. This equation is used to compute the advection velocity and if useParentEigenvalues=true then the eigenvalues of this system are used to compute the level of dissipation in the flux functions.

11.5.3 Parent Updater Data

`in` (string vector, required)

`Species densities` Entries 1-N where N is the number of species

0. variables 0-(N-1) n_i number density of species i

`Vector of conserved quantities` Entries are determined by the Equation sub-block and only the first 4 entries are used in this equation. Entries 1-N where N the number variables in the parent equation

0. ρ species density
1. $\rho \, u_x$ species x momentum
2. $\rho \, u_y$ species y momentum
3. $\rho \, u_z$ species z momentum
4. all components beyond 3 are ignored.

11.5.4 Example

An example *multiSpeciesSingleVelocity* equation block is given below

```
<Equation speciesContinuity>
  kind = multiSpeciesSingleVelocityEqn
  useParentEigenvalues = true
  inputVariables = [qSpecies, q]
```

```
    numberOfSpecies = NSPECIES

    <Equation realGas>
      kind = realGasEqn
      inputVariables = [q, realGasVariables]
      numSpecies = NSPECIES
    </Equation>

  </Equation>
```

The following equations can be used to simulate ionized, quasi-neutral plasmas in the magnetohydrodynamic limit:

11.6 mhdDednerEqn

Defines the equations of ideal compressible magnetohydrodynamics with divergence cleaning:

$$\begin{aligned}
\frac{\partial \rho}{\partial t} + \nabla \cdot [\rho \, \mathbf{u}] &= 0 \\
\frac{\partial \rho \, \mathbf{u}}{\partial t} + \nabla \cdot \left[\rho \, \mathbf{u} \, \mathbf{u}^T - \mathbf{b} \, \mathbf{b}^T + \mathbb{I} \left(P + \tfrac{1}{2} \left|\mathbf{b}\right|^2\right)\right] &= 0 \\
\frac{\partial E}{\partial t} + \nabla \cdot [(E + P) \, \mathbf{u} + \mathbf{e} \times \mathbf{b}] &= 0 \\
\frac{\partial \mathbf{b}^{\text{plasma}}}{\partial t} + \nabla \times \mathbf{e} + \nabla \psi &= 0 \\
\frac{\partial \psi}{\partial t} + \nabla \cdot \left[c_{\text{fast}}^2 \mathbf{b}\right] &= 0
\end{aligned}$$

Here, $\mathbb{I}$ is the identity matrix, $P = \rho\epsilon(\gamma - 1)$ is the pressure of an ideal gas, ϵ is the specific internal energy and γ is the adiabatic index (ratio of specific heats). The quantity c_{fast} corresponds to the fastest wave speed over the entire simulation domain; divergence errors are advected out of the domain with this speed.

The electromagnetic fields are defined as:

$$\begin{aligned}
\mathbf{b} &= \mathbf{b}^{\text{plasma}} + \mathbf{b}^{\text{external}} = \mu_0^{-1/2} \left(\mathbf{B}^{\text{plasma}} + \mathbf{B}^{\text{external}}\right) \\
\mathbf{e} &= -\mathbf{u} \times \mathbf{b} + \mathbf{e}^{\text{external}} = \mu_0^{-1/2} \left(-\mathbf{u} \times \mathbf{B} + \mathbf{E}^{\text{external}}\right)
\end{aligned}$$

Here, $\mathbf{b}^{\text{plasma}}$ is the magnetic field induced in the plasma by the inductive electric field, $\mathbf{e}$, while $\mathbf{e}^{\text{external}}$ and $\mathbf{b}^{\text{external}}$ are electromagnetic fields computed "externally" to the ideal magnetohydrodynamic equations.

11.6.1 Parameters

`basementPressure` (float, optional) The minimum pressure allowed. Pressures below this value will be replaced with this value for primitive state, eigensystem and flux computations. Defaults to zero.

`basementDensity` (float, optional) The minimum density allowed. Densities below this value will be replaced with this value for primitive state, eigensystem and flux computations. Defaults to zero.

`gasGamma` (float, optional) Specifies the adiabatic index (ratio of specific heats), γ. Defaults to 5/3.

`mu0` (float, optional) Optional value for the constant μ_0. Defaults to $4\pi \times 10^{-7}$.

`externalEfield` (string, optional) Specifies the name of the data structure containing the externally computed electric field, $\mathbf{e}^{\text{external}}$.

`externalBfield` (string, optional) Specifies the name of the data structure containing the externally computed magnetic field, $\mathbf{b}^{\text{external}}$.

11.6.2 Parent Updater Data

`in` (string vector, required)

`Vector of Conserved Quantities` (*nodalArray*, 9-components, required) The vector of conserved quantities, $\mathbf{q}$ has 9 entries:

0. ρ: mass density
1. $\rho\, u_{\hat{\mathbf{i}}} = \rho\mathbf{u}\cdot\hat{\mathbf{i}}$: momentum density in the $\hat{\mathbf{i}}$ direction
2. $\rho\, u_{\hat{\mathbf{j}}} = \rho\mathbf{u}\cdot\hat{\mathbf{j}}$: momentum density in the $\hat{\mathbf{j}}$ direction
3. $\rho\, u_{\hat{\mathbf{k}}} = \rho\mathbf{u}\cdot\hat{\mathbf{k}}$: momentum density in the $\hat{\mathbf{k}}$ direction
4. $E = \frac{P}{\gamma-1} + \frac{1}{2}\rho|\mathbf{u}|^2 + \frac{1}{2}|\mathbf{b}|^2$: total energy density
5. $b_{\hat{\mathbf{i}}} = \mathbf{b}\cdot\hat{\mathbf{i}} = \mu_0^{-1/2}\mathbf{B}\cdot\hat{\mathbf{i}}$: magnetic field normalized by permeability of free-space in the $\hat{\mathbf{i}}$ direction
6. $b_{\hat{\mathbf{j}}} = \mathbf{b}\cdot\hat{\mathbf{j}} = \mu_0^{-1/2}\mathbf{B}\cdot\hat{\mathbf{j}}$: magnetic field normalized by permeability of free-space in the $\hat{\mathbf{j}}$ direction
7. $b_{\hat{\mathbf{k}}} = \mathbf{b}\cdot\hat{\mathbf{k}} = \mu_0^{-1/2}\mathbf{B}\cdot\hat{\mathbf{k}}$: magnetic field normalized by permeability of free-space in the $\hat{\mathbf{k}}$ direction
8. ψ: correction potential

`Fastest Wave Speed` (*dynVector*, 1-component, required) The fastest wave speed across the entire simulation domain, c_{fast}. Can be computed using *hyperbolic (1d, 2d, 3d)* (see below).

`Externally Computed Electric Field` (*nodalArray*, 3-components, optional) Additional terms in the generalized Ohm's law, $\mathbf{E}^{\text{external}}$, computed "externally" to the ideal magnetohydrodynamic system. The data structure containing $\mathbf{e}^{\text{external}}$ is specified by the "externalEField" option described below.

0. $e^{\text{external}}_{\hat{\mathbf{i}}} = \mathbf{e}^{\text{external}}\cdot\hat{\mathbf{i}} = \mu_0^{-1/2}\mathbf{E}^{\text{external}}\cdot\hat{\mathbf{i}}$: "externally" computed electric field normalized by permeability of free-space in the $\hat{\mathbf{i}}$ direction.
1. $e^{\text{external}}_{\hat{\mathbf{j}}} = \mathbf{e}^{\text{external}}\cdot\hat{\mathbf{j}} = \mu_0^{-1/2}\mathbf{E}^{\text{external}}\cdot\hat{\mathbf{j}}$:"externally" computed electric field normalized by permeability of free-space in the $\hat{\mathbf{j}}$ direction
2. $e^{\text{external}}_{\hat{\mathbf{k}}} = \mathbf{e}^{\text{external}}\cdot\hat{\mathbf{k}} = \mu_0^{-1/2}\mathbf{E}^{\text{external}}\cdot\hat{\mathbf{k}}$: "externally" computed electric field normalized by permeability of free-space in the $\hat{\mathbf{k}}$ direction

`Externally Computed Magnetic Field` (*nodalArray*, 3-components, optional) Additional contribution to the magnetic field, $\mathbf{b}^{\text{external}}$, which is not evolved by the induction equation, but does contribute to the Lorentz force and the work done on the plasma. The data structure containing $\mathbf{b}^{\text{external}}$ is specified by the "externalBField" option described below.

0. $b^{\text{external}}_{\hat{\mathbf{i}}} = \mathbf{b}^{\text{external}}\cdot\hat{\mathbf{i}} = \mu_0^{-1/2}\mathbf{B}^{\text{external}}\cdot\hat{\mathbf{i}}$: magnetic field normalized by permeability of free-space in the $\hat{\mathbf{i}}$ direction
1. $b^{\text{external}}_{\hat{\mathbf{j}}} = \mathbf{b}^{\text{external}}\cdot\hat{\mathbf{j}} = \mu_0^{-1/2}\mathbf{B}^{\text{external}}\cdot\hat{\mathbf{j}}$: magnetic field normalized by permeability of free-space in the $\hat{\mathbf{j}}$ direction
2. $b^{\text{external}}_{\hat{\mathbf{k}}} = \mathbf{b}^{\text{external}}\cdot\hat{\mathbf{k}} = \mu_0^{-1/2}\mathbf{B}^{\text{external}}\cdot\hat{\mathbf{k}}$: magnetic field normalized by permeability of free-space in the $\hat{\mathbf{k}}$ direction

`out` (string vector, required) For the mhdDednerEqn, one of four output variables are computed, depending on whether the equation is combined with an updater capable of computing fluxes (*classicMusclUpdater (1d, 2d, 3d)*), primitive variables (*computePrimitiveState(1d, 2d, 3d)*), the time step associated with the CFL condition (*timeStepRestrictionUpdater (1d, 2d, 3d)*) or the fastest wave speed in the grid (*hyperbolic (1d, 2d, 3d)*).

`Vector of Fluxes` (*nodalArray*, 9-components) When combined with an updater that computes $\nabla \cdot \mathcal{F}(\mathbf{w})$ (e.g. *classicMusclUpdater (1d, 2d, 3d)*), the equation system returns:

0. $\nabla \cdot \mathcal{F}(\rho)$: mass flux
1. $\nabla \cdot \mathcal{F}\left(\rho\, u_{\hat{\mathbf{i}}}\right)$: $\hat{\mathbf{i}}$ momentum flux
2. $\nabla \cdot \mathcal{F}\left(\rho\, u_{\hat{\mathbf{j}}}\right)$: $\hat{\mathbf{j}}$ momentum flux
3. $\nabla \cdot \mathcal{F}\left(\rho\, u_{\hat{\mathbf{k}}}\right)$: $\hat{\mathbf{k}}$ momentum flux
4. $\nabla \cdot \mathcal{F}(E)$: total energy flux
5. $\nabla \cdot \mathcal{F}\left(b_{\hat{\mathbf{i}}}\right)$: $\hat{\mathbf{i}}$ magnetic field flux
6. $\nabla \cdot \mathcal{F}\left(b_{\hat{\mathbf{j}}}\right)$: $\hat{\mathbf{j}}$ magnetic field flux
7. $\nabla \cdot \mathcal{F}\left(b_{\hat{\mathbf{k}}}\right)$: $\hat{\mathbf{k}}$ magnetic field flux
8. $\nabla \cdot \mathcal{F}(\psi)$: correction potential flux

`Vector of Primitive States` (*nodalArray*, 9-components) When combined with an updater that computes $\mathbf{w} = \mathbf{w}(\mathbf{q})$ (e.g. *computePrimitiveState(1d, 2d, 3d)*), the equation systen returns:

0. ρ: mass density
1. $u_{\hat{\mathbf{i}}} = \mathbf{u} \cdot \hat{\mathbf{i}}$: velocity in the $\hat{\mathbf{i}}$ direction
2. $u_{\hat{\mathbf{j}}} = \mathbf{u} \cdot \hat{\mathbf{j}}$: velocity in the $\hat{\mathbf{j}}$ direction
3. $u_{\hat{\mathbf{k}}} = \mathbf{u} \cdot \hat{\mathbf{k}}$: velocity in the $\hat{\mathbf{k}}$ direction
4. $P = \rho\epsilon(\gamma - 1)$: ideal gas pressure
5. $b_{\hat{\mathbf{i}}} = \mathbf{b} \cdot \hat{\mathbf{i}} = \mu_0^{-1/2}\mathbf{B} \cdot \hat{\mathbf{i}}$: magnetic field normalized by permeability of free-space in the $\hat{\mathbf{i}}$ direction
6. $b_{\hat{\mathbf{j}}} = \mathbf{b} \cdot \hat{\mathbf{j}} = \mu_0^{-1/2}\mathbf{B} \cdot \hat{\mathbf{j}}$: magnetic field normalized by permeability of free-space in the $\hat{\mathbf{j}}$ direction
7. $b_{\hat{\mathbf{k}}} = \mathbf{b} \cdot \hat{\mathbf{k}} = \mu_0^{-1/2}\mathbf{B} \cdot \hat{\mathbf{k}}$: magnetic field normalized by permeability of free-space in the $\hat{\mathbf{k}}$ direction
8. ψ: correction potential

`Time Step` (*dynVector*, 1-component) When combined with *timeStepRestrictionUpdater (1d, 2d, 3d)*, the equation system returns the time step consisten with the CFL condition across the entire simulation domain.

`Fastest Wave Speed` (*dynVector*, 1-component) When combined with *hyperbolic (1d, 2d, 3d)*, the equation system returns the fastest wave speed across the entire simulation domain, c_{fast}.

11.6.3 Examples

The following block demonstrates the mhdDednerEqn used in combination with *classicMusclUpdater (1d, 2d, 3d)* to compute $\nabla \cdot \mathcal{F}(\mathbf{w})$ with an externally supplied magnetic field:

```
<Updater hyper>
  kind=classicMuscl1d
  onGrid=domain

  # input nodal component arrays
  in=[q   backgroundB]

  # output nodal component array
  out=[qnew]

  # input dynVector containing fastest wave speed
  waveSpeeds=[waveSpeed]

  # the numerical flux to use
  numericalFlux= hlldFlux

  # CFL number to use
  cfl=0.3
  # Form of variables to limit
  variableForm= primitive

  # Limiter; one per input nodal component array
  limiter=[minmod   minmod]

  # list of equations to solve
  equations=[mhd]

  <Equation mhd>
    kind=mhdDednerEqn
    gasGamma=1.4
    externalBfield="backgroundB"
  </Equation>

</Updater>
```

The following block demonstrates the mhdDednerEqn used in combination with *timeStepRestrictionUpdater (1d, 2d, 3d)* and *hyperbolic (1d, 2d, 3d)* to compute c_{fast} with an externally supplied magnetic field:

```
<Updater getWaveSpeed>
  kind=timeStepRestrictionUpdater1d
  onGrid=domain

  # input nodal component arrays
  in=[q   backgroundB]

  # output dynVector containing fastest wave speed
  waveSpeeds=[waveSpeed]

  # list of equations to compute fastest wave speed for
  restrictions=[idealMhd]

  # courant condition to apply to the timestep
  courantCondition=1.0

  <TimeStepRestriction idealMhd>
    kind=hyperbolic1d
    model=mhdDednerEqn
    gasGamma= 1.4
```

```
      externalBfield=True
      includeInTimeStep=False
    </TimeStepRestriction>
  </Updater>
```

11.7 mhdDednerEosEqn

Defines the equations of ideal compressible magnetohydrodynamics with and arbitrary equation of state (EOS) and divergence cleaning:

$$\begin{aligned}
\frac{\partial \rho}{\partial t} + \nabla \cdot [\rho \, \mathbf{u}] &= 0 \\
\frac{\partial \rho \, \mathbf{u}}{\partial t} + \nabla \cdot \left[\rho \, \mathbf{u} \, \mathbf{u}^T - \mathbf{b} \, \mathbf{b}^T + \mathbb{I} \left(P + \tfrac{1}{2} |\mathbf{b}|^2 \right) \right] &= 0 \\
\frac{\partial E}{\partial t} + \nabla \cdot [(E + P) \, \mathbf{u} + \mathbf{e} \times \mathbf{b}] &= 0 \\
\frac{\partial \mathbf{b}^{\text{plasma}}}{\partial t} + \nabla \times \mathbf{e} + \nabla \psi &= 0 \\
\frac{\partial \psi}{\partial t} + \nabla \cdot \left[c_{\text{fast}}^2 \mathbf{b} \right] &= 0
\end{aligned}$$

Here, $\mathbb{I}$ is the identity matrix and P is the pressure as specified by an external EOS. Updaters that compute all the data required from an EOS are found in *vanDerWaalsComputeVariables*, *sesameComputeVariables* and *propaceosComputeVariables*. The quantity c_{fast} corresponds to the fastest wave speed over the entire simulation domain; divergence errors are advected out of the domain with this speed.

The electromagnetic fields are defined as:

$$\begin{aligned}
\mathbf{b} &= \mathbf{b}^{\text{plasma}} + \mathbf{b}^{\text{external}} = \mu_0^{-1/2} \left(\mathbf{B}^{\text{plasma}} + \mathbf{B}^{\text{external}} \right) \\
\mathbf{e} &= -\mathbf{u} \times \mathbf{b} + \mathbf{e}^{\text{external}} = \mu_0^{-1/2} \left(-\mathbf{u} \times \mathbf{B} + \mathbf{E}^{\text{external}} \right)
\end{aligned}$$

Here, $\mathbf{b}^{\text{plasma}}$ is the magnetic field induced in the plasma by the inductive electric field, $\mathbf{e}$, while $\mathbf{e}^{\text{external}}$ and $\mathbf{b}^{\text{external}}$ are electromagnetic fields computed "externally" to the ideal magnetohydrodynamic equations.

11.7.1 Parameters

`basementPressure` (float, optional) The minimum pressure allowed. Pressures below this value will be replaced with this value for primitive state, eigensystem and flux computations. Defaults to zero.

`basementDensity` (float, optional) The minimum density allowed. Densities below this value will be replaced with this value for primitive state, eigensystem and flux computations. Defaults to zero.

`mu0` (float, optional) Optional value for the constant μ_0. Defaults to $4\pi \times 10^{-7}$.

`externalEfield` (string, optional) Specifies the name of the data structure containing the externally computed electric field, $\mathbf{e}^{\text{external}}$.

`externalBfield` (string, optional) Specifies the name of the data structure containing the externally computed magnetic field, $\mathbf{b}^{\text{external}}$.

11.7.2 Parent Updater Data

`in` (string vector, required)

`Vector of Conserved Quantities` (*nodalArray*, **9-components, required**) The vector of conserved quantities, $\mathbf{q}$ has 9 entries:

0. ρ: mass density
1. $\rho\, u_{\hat{\mathbf{i}}} = \rho\mathbf{u}\cdot\hat{\mathbf{i}}$: momentum density in the $\hat{\mathbf{i}}$ direction
2. $\rho\, u_{\hat{\mathbf{j}}} = \rho\mathbf{u}\cdot\hat{\mathbf{j}}$: momentum density in the $\hat{\mathbf{j}}$ direction
3. $\rho\, u_{\hat{\mathbf{k}}} = \rho\mathbf{u}\cdot\hat{\mathbf{k}}$: momentum density in the $\hat{\mathbf{k}}$ direction
4. $E = \rho\epsilon + \frac{1}{2}\rho|\mathbf{u}|^2 + \frac{1}{2}|\mathbf{b}|^2$: total energy density where ϵ is the specific internal energy
5. $b_{\hat{\mathbf{i}}} = \mathbf{b}\cdot\hat{\mathbf{i}} = \mu_0^{-1/2}\mathbf{B}\cdot\hat{\mathbf{i}}$: magnetic field normalized by permeability of free-space in the $\hat{\mathbf{i}}$ direction
6. $b_{\hat{\mathbf{j}}} = \mathbf{b}\cdot\hat{\mathbf{j}} = \mu_0^{-1/2}\mathbf{B}\cdot\hat{\mathbf{j}}$: magnetic field normalized by permeability of free-space in the $\hat{\mathbf{j}}$ direction
7. $b_{\hat{\mathbf{k}}} = \mathbf{b}\cdot\hat{\mathbf{k}} = \mu_0^{-1/2}\mathbf{B}\cdot\hat{\mathbf{k}}$: magnetic field normalized by permeability of free-space in the $\hat{\mathbf{k}}$ direction
8. ψ: correction potential

`Pressure` (*nodalArray*, **1-component, required**) Value of the pressure as computed by the external EOS.

`Sound speed squared` (*nodalArray*, **1-component, required**) Value of the sound speed squared as computed by the external EOS.

`internal energy` (*nodalArray*, **1-component, required**) Value of the internal energy ($\rho\epsilon$) as computed by the external EOS.

`Fastest Wave Speed` (*dynVector*, **1-component, required**) The fastest wave speed across the entire simulation domain, c_{fast}. Can be computed using *hyperbolic (1d, 2d, 3d)* (see below).

`Externally Computed Electric Field` (*nodalArray*, **3-components, optional**) Additional terms in the generalized Ohm's law, $\mathbf{E}^{\text{external}}$, computed "externally" to the ideal magnetohydrodynamic system. The data structure containing $\mathbf{e}^{\text{external}}$ is specified by the "externalEField" option described below.

0. $e^{\text{external}}_{\hat{\mathbf{i}}} = \mathbf{e}^{\text{external}}\cdot\hat{\mathbf{i}} = \mu_0^{-1/2}\mathbf{E}^{\text{external}}\cdot\hat{\mathbf{i}}$: "externally" computed electric field normalized by permeability of free-space in the $\hat{\mathbf{i}}$ direction.
1. $e^{\text{external}}_{\hat{\mathbf{j}}} = \mathbf{e}^{\text{external}}\cdot\hat{\mathbf{j}} = \mu_0^{-1/2}\mathbf{E}^{\text{external}}\cdot\hat{\mathbf{j}}$:"externally" computed electric field normalized by permeability of free-space in the $\hat{\mathbf{j}}$ direction
2. $e^{\text{external}}_{\hat{\mathbf{k}}} = \mathbf{e}^{\text{external}}\cdot\hat{\mathbf{k}} = \mu_0^{-1/2}\mathbf{E}^{\text{external}}\cdot\hat{\mathbf{k}}$: "externally" computed electric field normalized by permeability of free-space in the $\hat{\mathbf{k}}$ direction

`Externally Computed Magnetic Field` (*nodalArray*, **3-components, optional**) Additional contribution to the magnetic field, $\mathbf{b}^{\text{external}}$, which is not evolved by the induction equation, but does contribute to the Lorentz force and the work done on the plasma. The data structure containing $\mathbf{b}^{\text{external}}$ is specified by the "externalBField" option described below.

0. $b^{\text{external}}_{\hat{\mathbf{i}}} = \mathbf{b}^{\text{external}}\cdot\hat{\mathbf{i}} = \mu_0^{-1/2}\mathbf{B}^{\text{external}}\cdot\hat{\mathbf{i}}$: magnetic field normalized by permeability of free-space in the $\hat{\mathbf{i}}$ direction
1. $b^{\text{external}}_{\hat{\mathbf{j}}} = \mathbf{b}^{\text{external}}\cdot\hat{\mathbf{j}} = \mu_0^{-1/2}\mathbf{B}^{\text{external}}\cdot\hat{\mathbf{j}}$: magnetic field normalized by permeability of free-space in the $\hat{\mathbf{j}}$ direction
2. $b^{\text{external}}_{\hat{\mathbf{k}}} = \mathbf{b}^{\text{external}}\cdot\hat{\mathbf{k}} = \mu_0^{-1/2}\mathbf{B}^{\text{external}}\cdot\hat{\mathbf{k}}$: magnetic field normalized by permeability of free-space in the $\hat{\mathbf{k}}$ direction

`out` (string vector, required) For the mhdDednerEosEqn, one of four output variables are computed, depending on whether the equation is combined with an updater capable of computing fluxes (*classicMusclUpdater (1d, 2d, 3d)*), primitive variables (*computePrimitiveState(1d, 2d, 3d)*), the time step associated with the CFL condition (*timeStepRestrictionUpdater (1d, 2d, 3d)*) or the fastest wave speed in the grid (*hyperbolic (1d, 2d, 3d)*).

`Vector of Fluxes` (*nodalArray*, 9-components) When combined with an updater that computes $\nabla \cdot \mathcal{F}(\mathbf{w})$ (e.g. *classicMusclUpdater (1d, 2d, 3d)*), the equation system returns:

0. $\nabla \cdot \mathcal{F}(\rho)$: mass flux
1. $\nabla \cdot \mathcal{F}\left(\rho\, u_{\hat{\mathbf{i}}}\right)$: $\hat{\mathbf{i}}$ momentum flux
2. $\nabla \cdot \mathcal{F}\left(\rho\, u_{\hat{\mathbf{j}}}\right)$: $\hat{\mathbf{j}}$ momentum flux
3. $\nabla \cdot \mathcal{F}\left(\rho\, u_{\hat{\mathbf{k}}}\right)$: $\hat{\mathbf{k}}$ momentum flux
4. $\nabla \cdot \mathcal{F}(E)$: total energy flux
5. $\nabla \cdot \mathcal{F}\left(b_{\hat{\mathbf{i}}}\right)$: $\hat{\mathbf{i}}$ magnetic field flux
6. $\nabla \cdot \mathcal{F}\left(b_{\hat{\mathbf{j}}}\right)$: $\hat{\mathbf{j}}$ magnetic field flux
7. $\nabla \cdot \mathcal{F}\left(b_{\hat{\mathbf{k}}}\right)$: $\hat{\mathbf{k}}$ magnetic field flux
8. $\nabla \cdot \mathcal{F}(\psi)$: correction potential flux

`Vector of Primitive States` (*nodalArray*, 9-components) When combined with an updater that computes $\mathbf{w} = \mathbf{w}(\mathbf{q})$ (e.g. *computePrimitiveState(1d, 2d, 3d)*), the equation systen returns:

0. ρ: mass density
1. $u_{\hat{\mathbf{i}}} = \mathbf{u} \cdot \hat{\mathbf{i}}$: velocity in the $\hat{\mathbf{i}}$ direction
2. $u_{\hat{\mathbf{j}}} = \mathbf{u} \cdot \hat{\mathbf{j}}$: velocity in the $\hat{\mathbf{j}}$ direction
3. $u_{\hat{\mathbf{k}}} = \mathbf{u} \cdot \hat{\mathbf{k}}$: velocity in the $\hat{\mathbf{k}}$ direction
4. $P = \rho\epsilon(\gamma - 1)$: ideal gas pressure
5. $b_{\hat{\mathbf{i}}} = \mathbf{b} \cdot \hat{\mathbf{i}} = \mu_0^{-1/2}\mathbf{B} \cdot \hat{\mathbf{i}}$: magnetic field normalized by permeability of free-space in the $\hat{\mathbf{i}}$ direction
6. $b_{\hat{\mathbf{j}}} = \mathbf{b} \cdot \hat{\mathbf{j}} = \mu_0^{-1/2}\mathbf{B} \cdot \hat{\mathbf{j}}$: magnetic field normalized by permeability of free-space in the $\hat{\mathbf{j}}$ direction
7. $b_{\hat{\mathbf{k}}} = \mathbf{b} \cdot \hat{\mathbf{k}} = \mu_0^{-1/2}\mathbf{B} \cdot \hat{\mathbf{k}}$: magnetic field normalized by permeability of free-space in the $\hat{\mathbf{k}}$ direction
8. ψ: correction potential

`Time Step` (*dynVector*, 1-component) When combined with *timeStepRestrictionUpdater (1d, 2d, 3d)*, the equation system returns the time step consisten with the CFL condition across the entire simulation domain.

`Fastest Wave Speed` (*dynVector*, 1-component) When combined with *hyperbolic (1d, 2d, 3d)*, the equation system returns the fastest wave speed across the entire simulation domain, c_{fast}.

11.7.3 Examples

The following block demonstrates the mhdDednerEosEqn used in combination with *classicMusclUpdater (1d, 2d, 3d)* to compute $\nabla \cdot \mathcal{F}(\mathbf{w})$:

```
<Updater hyper>
  kind = classicMuscl2d
  onGrid = domain

  # input nodal component arrays
  in=[q, pressure, soundSqr, intEnergy]

  # output nodal component arrays
  out = [qNew]

  # input dynVector containing fastest wave speed
  waveSpeeds = [waveSpeed]

  # the numerical flux to use
  numericalFlux = hlldFlux

  # CFL number to use
  cfl = 0.5

  # determines solve is conservative or primitive
  variableForm = conservative

  # Limiter; one per input nodal component array
  limiter=[muscl, muscl, muscl, muscl]

  # list of equations to solve
  equations = [mhd]

  <Equation mhd>
    kind=mhdDednerEosEqn
    mu0=1.0
  </Equation>

</Updater>
```

The following block demonstrates the mhdDednerEosEqn used in combination with *timeStepRestrictionUpdater (1d, 2d, 3d)* and *hyperbolic (1d, 2d, 3d)* to compute c_{fast} with an externally supplied magnetic field:

```
<Updater getWaveSpeed>
  kind=timeStepRestrictionUpdater2d
  onGrid=domain

  # input nodal component arrays
  in=[q, pressure, soundSqr, intEnergy]

  # output dynVector containing fastest wave speed
  waveSpeeds=[waveSpeed]

  # list of equations to compute fastest wave speed for
  restrictions=[idealMhd]

  # courant condition to apply to the timestep
  courantCondition=0.5

  <TimeStepRestriction idealMhd>
    kind=hyperbolic1d
    model=mhdDednerEosEqn
    mu0=1.0
```

```
    </TimeStepRestriction>
  </Updater>
```

11.8 gasDynamicMhdDednerEqn

Defines the equations of inviscid fluid dynamics coupled to pre-Maxwell's equations in source term form with divergence cleaning:

$$\frac{\partial \rho}{\partial t} + \nabla \cdot [\rho \, \mathbf{u}] = 0$$

$$\frac{\partial (\rho \mathbf{u})}{\partial t} + \nabla \cdot \left[\rho \, \mathbf{u} \, \mathbf{u}^T + \mathbb{I} P\right] = \sum_{\text{species}} \left(q^{\text{species}} \mathbf{E} + \mathbf{J}^{\text{species}} \times \mathbf{B}\right)$$

$$\frac{\partial E}{\partial t} + \nabla \cdot [(E + P) \, \mathbf{u}] = \sum_{\text{species}} \mathbf{J}^{\text{species}} \cdot \mathbf{E}^{\text{species}}$$

$$\frac{\partial \mathbf{B}^{\text{plasma}}}{\partial t} + \nabla \times \mathbf{E} + \nabla \psi = 0$$

$$\frac{\partial \psi}{\partial t} + \nabla \cdot \left[c_{\text{fast}}^2 \mathbf{b}\right] = 0$$

Here, q^{species} is the species charge density, $\mathbf{J}^{\text{species}}$ is the species current density, $\mathbb{I}$ is the identity matrix, $P = \rho\epsilon(\gamma - 1)$ is the pressure of an ideal gas, ϵ is the specific internal energy and γ is the adiabatic index (ratio of specific heats). The quantity c_{fast} corresponds to the fastest wave speed over the entire simulation domain; divergence errors are advected out of the domain with this speed.

In order to integrate these equations, USim casts them into flux-conservative form using the following standard identities (note that the use of these identities does not require an assumption of quasi-neutrality):

$$\sum_{\text{species}} \left(q^{\text{species}} \mathbf{E} + \mathbf{J}^{\text{species}} \times \mathbf{B}\right) = -\frac{\partial c^{-2} \mathbf{S}^{\text{EM}}}{\partial t} + \nabla \cdot \mathcal{T}^{\text{EM}}$$

$$\sum_{\text{species}} \mathbf{J}^{\text{species}} \cdot \mathbf{E} = -\frac{\partial E^{\text{EM}}}{\partial t} - \nabla \cdot \mathbf{S}^{\text{EM}}$$

Here, $\mathcal{T}^{\text{EM}}$ is the electromagnetic stress tensor and $\mathbf{S}^{\text{EM}}$ is the electromagnetic energy (Poynting) flux vector, which are defined as:

$$\mathcal{T}^{\text{EM}} = \frac{1}{\mu_0}\left(\frac{\mathbf{E}\mathbf{E}^T}{c^2} + \mathbf{B}\mathbf{B}^T\right) + \mathbb{I} E_{\text{EM}} = \frac{\mathbf{e}\mathbf{e}^T}{c^2} + \mathbf{b}\mathbf{b}^T + \mathbb{I} E_{\text{EM}}$$

$$\mathbf{S}^{\text{EM}} = \mu_0^{-1} \mathbf{E} \times \mathbf{B} = \mathbf{e} \times \mathbf{b}$$

$$E^{\text{EM}} = \frac{1}{2\mu_0}\left(\frac{|\mathbf{E}|^2}{c^2} + |\mathbf{B}|^2\right) = \frac{1}{2}\left(\frac{|\mathbf{e}|^2}{c^2} + |\mathbf{b}|^2\right)$$

Here, E^{EM} is the electromagnetic energy density and the electromagnetic fields are defined as:

$$\mathbf{b} = \mathbf{b}^{\text{plasma}} + \mathbf{b}^{\text{external}} = \mu_0^{-1/2}\left(\mathbf{B}^{\text{plasma}} + \mathbf{B}^{\text{external}}\right)$$

$$\mathbf{e} = -\mathbf{u} \times \mathbf{b} + \mathbf{e}^{\text{external}} = \mu_0^{-1/2}\left(-\mathbf{u} \times \mathbf{B} + \mathbf{E}^{\text{external}}\right)$$

Here, $\mathbf{b}^{\text{plasma}}$ is the magnetic field induced in the plasma by the inductive electric field, $\mathbf{e}$, while $\mathbf{e}^{\text{external}}$ and $\mathbf{b}^{\text{external}}$ are electromagnetic fields computed "externally" to the pre-Maxwell equations.

With these identitifications, the gasDynamicMhdDednerEqn takes the form:

$$\begin{aligned}
\frac{\partial \rho}{\partial t} + \nabla \cdot [\rho\, \mathbf{u}] &= 0 \\
\frac{\partial \left(\rho \mathbf{u} + c^{-2}\mathbf{S}^{\mathrm{EM}}\right)}{\partial t} + \nabla \cdot \left[\rho\, \mathbf{u}\, \mathbf{u}^T + \mathbb{I} P - \mathcal{T}^{\mathrm{EM}}\right] &= 0 \\
\frac{\partial \left(E + E^{\mathrm{EM}}\right)}{\partial t} + \nabla \cdot \left[(E+P)\, \mathbf{u} + \mathbf{S}^{\mathrm{EM}}\right] &= 0 \\
\frac{\partial \mathbf{b}^{\mathrm{plasma}}}{\partial t} + \nabla \times \mathbf{e} + \nabla \psi &= 0 \\
\frac{\partial \psi}{\partial t} + \nabla \cdot \left[c_{\mathrm{fast}}^2 \mathbf{b}\right] &= 0
\end{aligned}$$

This flux-conservative formulation is implemented in USim.

11.8.1 Parameters

`lightSpeed` (float, optional) The speed of light in m/s. Defaults to 2.99792458e8.

`basementPressure` (float, optional) The minimum pressure allowed. Pressures below this value will be replaced with this value for primitive state, eigensystem and flux computations. Defaults to zero.

`basementDensity` (float, optional) The minimum density allowed. Densities below this value will be replaced with this value for primitive state, eigensystem and flux computations. Defaults to zero.

`gasGamma` (float, optional) Specifies the adiabatic index (ratio of specific heats), γ. Defaults to 5/3.

`externalEfield` (string, optional) Specifies the name of the data structure containing the externally computed electric field, $\mathbf{e}^{\mathrm{external}}$.

`externalBfield` (string, optional) Specifies the name of the data structure containing the externally computed magnetic field, $\mathbf{b}^{\mathrm{external}}$.

11.8.2 Parent Updater Data

`in` (string vector, required)

`Vector of Conserved Quantities` (*nodalArray*, 9-components, required) The vector of conserved quantities, $\mathbf{q}$ has 9 entries:

0. ρ: mass density
1. $\rho\, u_{\hat{\mathbf{i}}} + c^{-2} S_{\hat{\mathbf{i}}}^{\mathrm{EM}} = \left(\rho \mathbf{u} + c^{-2}\mathbf{S}^{\mathrm{EM}}\right) \cdot \hat{\mathbf{i}}$: total momentum density in the $\hat{\mathbf{i}}$ direction
2. $\rho\, u_{\hat{\mathbf{j}}} + c^{-2} S_{\hat{\mathbf{j}}}^{\mathrm{EM}} = \left(\rho \mathbf{u} + c^{-2}\mathbf{S}^{\mathrm{EM}}\right) \cdot \hat{\mathbf{j}}$: total momentum density in the $\hat{\mathbf{j}}$ direction
3. $\rho\, u_{\hat{\mathbf{k}}} + c^{-2} S_{\hat{\mathbf{k}}}^{\mathrm{EM}} = \left(\rho \mathbf{u} + c^{-2}\mathbf{S}^{\mathrm{EM}}\right) \cdot \hat{\mathbf{k}}$: total momentum density in the $\hat{\mathbf{k}}$ direction
4. $E + E^{\mathrm{EM}} = \frac{P}{\gamma - 1} + \frac{1}{2}\rho |\mathbf{u}|^2 + E^{\mathrm{EM}}$: total energy density
5. $b_{\hat{\mathbf{i}}} = \mathbf{b} \cdot \hat{\mathbf{i}} = \mu_0^{-1/2} \mathbf{B} \cdot \hat{\mathbf{i}}$: magnetic field normalized by permeability of free-space in the $\hat{\mathbf{i}}$ direction
6. $b_{\hat{\mathbf{j}}} = \mathbf{b} \cdot \hat{\mathbf{j}} = \mu_0^{-1/2} \mathbf{B} \cdot \hat{\mathbf{j}}$: magnetic field normalized by permeability of free-space in the $\hat{\mathbf{j}}$ direction
7. $b_{\hat{\mathbf{k}}} = \mathbf{b} \cdot \hat{\mathbf{k}} = \mu_0^{-1/2} \mathbf{B} \cdot \hat{\mathbf{k}}$: magnetic field normalized by permeability of free-space in the $\hat{\mathbf{k}}$ direction
8. ψ: correction potential

`Fastest Wave Speed` (*dynVector*, 1-component, required) The fastest wave speed across the entire simulation domain, c_{fast}. Can be computed using *hyperbolic (1d, 2d, 3d)* (see below).

`Externally Computed Electric Field` (*nodalArray*, 3-components, optional) Additional terms in the generalized Ohm's law, $\mathbf{E}^{\text{external}}$, computed "externally" to the ideal magnetohydrodynamic system. The data structure containing $\mathbf{e}^{\text{external}}$ is specified by the "externalEField" option described below.

0. $e^{\text{external}}_{\hat{\mathbf{i}}} = \mathbf{e}^{\text{external}} \cdot \hat{\mathbf{i}} = \mu_0^{-1/2}\mathbf{E}^{\text{external}} \cdot \hat{\mathbf{i}}$: "externally" computed electric field normalized by permeability of free-space in the $\hat{\mathbf{i}}$ direction.
1. $e^{\text{external}}_{\hat{\mathbf{j}}} = \mathbf{e}^{\text{external}} \cdot \hat{\mathbf{j}} = \mu_0^{-1/2}\mathbf{E}^{\text{external}} \cdot \hat{\mathbf{j}}$:"externally" computed electric field normalized by permeability of free-space in the $\hat{\mathbf{j}}$ direction
2. $e^{\text{external}}_{\hat{\mathbf{k}}} = \mathbf{e}^{\text{external}} \cdot \hat{\mathbf{k}} = \mu_0^{-1/2}\mathbf{E}^{\text{external}} \cdot \hat{\mathbf{k}}$: "externally" computed electric field normalized by permeability of free-space in the $\hat{\mathbf{k}}$ direction

`Externally Computed Magnetic Field` (*nodalArray*, 3-components, optional) Additional contribution to the magnetic field, $\mathbf{b}^{\text{external}}$, which is not evolved by the induction equation, but does contribute to the Lorentz force and the work done on the plasma. The data structure containing $\mathbf{b}^{\text{external}}$ is specified by the "externalBField" option described below.

0. $b^{\text{external}}_{\hat{\mathbf{i}}} = \mathbf{b}^{\text{external}} \cdot \hat{\mathbf{i}} = \mu_0^{-1/2}\mathbf{B}^{\text{external}} \cdot \hat{\mathbf{i}}$: magnetic field normalized by permeability of free-space in the $\hat{\mathbf{i}}$ direction
1. $b^{\text{external}}_{\hat{\mathbf{j}}} = \mathbf{b}^{\text{external}} \cdot \hat{\mathbf{j}} = \mu_0^{-1/2}\mathbf{B}^{\text{external}} \cdot \hat{\mathbf{j}}$: magnetic field normalized by permeability of free-space in the $\hat{\mathbf{j}}$ direction
2. $b^{\text{external}}_{\hat{\mathbf{k}}} = \mathbf{b}^{\text{external}} \cdot \hat{\mathbf{k}} = \mu_0^{-1/2}\mathbf{B}^{\text{external}} \cdot \hat{\mathbf{k}}$: magnetic field normalized by permeability of free-space in the $\hat{\mathbf{k}}$ direction

`out` (string vector, required) For the gasDynamicMhdDednerEqn, one of four output variables are computed, depending on whether the equation is combined with an updater capable of computing fluxes (*classicMusclUpdater (1d, 2d, 3d)*), primitive variables (*computePrimitiveState(1d, 2d, 3d)*), the time step associated with the CFL condition (*timeStepRestrictionUpdater (1d, 2d, 3d)*) or the fastest wave speed in the grid (*hyperbolic (1d, 2d, 3d)*).

`Vector of Fluxes` (*nodalArray*, 9-components) When combined with an updater that computes $\nabla \cdot \mathcal{F}(\mathbf{w})$ (e.g. *classicMusclUpdater (1d, 2d, 3d)*), the equation system returns:

0. $\nabla \cdot \mathcal{F}(\rho)$: mass flux
1. $\nabla \cdot \mathcal{F}\left(\rho\, u_{\hat{\mathbf{i}}} + c^{-2} S^{\text{EM}}_{\hat{\mathbf{i}}}\right)$: $\hat{\mathbf{i}}$ momentum flux
2. $\nabla \cdot \mathcal{F}\left(\rho\, u_{\hat{\mathbf{j}}} + c^{-2} S^{\text{EM}}_{\hat{\mathbf{j}}}\right)$: $\hat{\mathbf{j}}$ momentum flux
3. $\nabla \cdot \mathcal{F}\left(\rho\, u_{\hat{\mathbf{k}}} + c^{-2} S^{\text{EM}}_{\hat{\mathbf{k}}}\right)$: $\hat{\mathbf{k}}$ momentum flux
4. $\nabla \cdot \mathcal{F}(E)$: total energy flux
5. $\nabla \cdot \mathcal{F}\left(b_{\hat{\mathbf{i}}}\right)$: $\hat{\mathbf{i}}$ magnetic field flux
6. $\nabla \cdot \mathcal{F}\left(b_{\hat{\mathbf{j}}}\right)$: $\hat{\mathbf{j}}$ magnetic field flux
7. $\nabla \cdot \mathcal{F}\left(b_{\hat{\mathbf{k}}}\right)$: $\hat{\mathbf{k}}$ magnetic field flux
8. $\nabla \cdot \mathcal{F}(\psi)$: correction potential flux

`Vector of Primitive States` (*nodalArray*, 9-components) When combined with an updater that computes $\mathbf{w} = \mathbf{w}(\mathbf{q})$ (e.g. *computePrimitiveState(1d, 2d, 3d)*), the equation systen returns:

0. ρ: mass density
1. $u_{\hat{\mathbf{i}}} = \mathbf{u} \cdot \hat{\mathbf{i}}$: velocity in the $\hat{\mathbf{i}}$ direction
2. $u_{\hat{\mathbf{j}}} = \mathbf{u} \cdot \hat{\mathbf{j}}$: velocity in the $\hat{\mathbf{j}}$ direction
3. $u_{\hat{\mathbf{k}}} = \mathbf{u} \cdot \hat{\mathbf{k}}$: velocity in the $\hat{\mathbf{k}}$ direction
4. $P = \rho\epsilon(\gamma - 1)$: ideal gas pressure
5. $b_{\hat{\mathbf{i}}} = \mathbf{b} \cdot \hat{\mathbf{i}} = \mu_0^{-1/2}\mathbf{B} \cdot \hat{\mathbf{i}}$: magnetic field normalized by permeability of free-space in the $\hat{\mathbf{i}}$ direction
6. $b_{\hat{\mathbf{j}}} = \mathbf{b} \cdot \hat{\mathbf{j}} = \mu_0^{-1/2}\mathbf{B} \cdot \hat{\mathbf{j}}$: magnetic field normalized by permeability of free-space in the $\hat{\mathbf{j}}$ direction
7. $b_{\hat{\mathbf{k}}} = \mathbf{b} \cdot \hat{\mathbf{k}} = \mu_0^{-1/2}\mathbf{B} \cdot \hat{\mathbf{k}}$: magnetic field normalized by permeability of free-space in the $\hat{\mathbf{k}}$ direction
8. ψ: correction potential

`Time Step` (*dynVector*, 1-component) When combined with *timeStepRestrictionUpdater (1d, 2d, 3d)*, the equation system returns the time step consisten with the CFL condition across the entire simulation domain.

`Fastest Wave Speed` (*dynVector*, 1-component) When combined with *hyperbolic (1d, 2d, 3d)*, the equation system returns the fastest wave speed across the entire simulation domain, c_{fast}.

11.8.3 Examples

The following block demonstrates the mhdDednerEqn used in combination with *classicMusclUpdater (1d, 2d, 3d)* to compute $\nabla \cdot \mathcal{F}(\mathbf{w})$ with an externally supplied magnetic field:

```
<Updater hyper>
  kind=classicMuscl1d
  onGrid=domain

  # input nodal component arrays
  in=[q   backgroundB]

  # output nodal component array
  out=[qnew]

  # input dynVector containing fastest wave speed
  waveSpeeds=[waveSpeed]

  # the numerical flux to use
  numericalFlux= hlldFlux

  # CFL number to use
  cfl=0.3
  # Form of variables to limit
  variableForm= primitive

  # Limiter; one per input nodal component array
  limiter=[minmod   minmod]

  # list of equations to solve
  equations=[mhd]
```

```
  <Equation mhd>
    kind=gasDynamicMhdDednerEqn
    gasGamma=1.4
    externalBfield="backgroundB"
  </Equation>

</Updater>
```

The following block demonstrates the gasDynamicMhdDednerEqn used in combination with *timeStepRestrictionUpdater (1d, 2d, 3d)* and *hyperbolic (1d, 2d, 3d)* to compute c_{fast} with an externally supplied magnetic field:

```
<Updater getWaveSpeed>
  kind=timeStepRestrictionUpdater1d
  onGrid=domain

  # input nodal component arrays
  in=[q   backgroundB]

  # output dynVector containing fastest wave speed
  waveSpeeds=[waveSpeed]

  # list of equations to compute fastest wave speed for
  restrictions=[idealMhd]

  # courant condition to apply to the timestep
  courantCondition=1.0

  <TimeStepRestriction idealMhd>
    kind=hyperbolic1d
    model=gasDynamicMhdDednerEqn
    gasGamma= 1.4
    externalBfield=True
    includeInTimeStep=False
  </TimeStepRestriction>
</Updater>
```

11.9 simpleTwoTemperatureMhdDednerEqn

Defines the equations of ideal compressible magnetohydrodynamics with divergence cleaning and an electron entropy equation:

$$\begin{aligned}
\frac{\partial \rho}{\partial t} + \nabla \cdot [\rho \, \mathbf{u}] &= 0 \\
\frac{\partial \rho \, \mathbf{u}}{\partial t} + \nabla \cdot \left[\rho \, \mathbf{u} \, \mathbf{u}^T - \mathbf{b} \, \mathbf{b}^T + \mathbb{I} \left(P_{\text{tot}} + \tfrac{1}{2} |\mathbf{b}|^2 \right) \right] &= 0 \\
\frac{\partial E}{\partial t} + \nabla \cdot [(E + P) \, \mathbf{u} + \mathbf{e} \times \mathbf{b}] &= 0 \\
\frac{\partial \mathbf{b}^{\text{plasma}}}{\partial t} + \nabla \times \mathbf{e} + \nabla \psi &= 0 \\
\frac{\partial \psi}{\partial t} + \nabla \cdot \left[c_{\text{fast}}^2 \mathbf{b} \right] &= 0 \\
\frac{\partial S_{\text{electron}}}{\partial t} + \nabla \cdot [S_{\text{electron}} \, \mathbf{u}] &= 0
\end{aligned}$$

Here, $\mathbb{I}$ is the identity matrix, $P_{\text{tot}} = P_{\text{ion}} + P_{\text{electron}} = \rho_{\text{ion}} \epsilon_{\text{ion}} (\gamma_{\text{ion}} - 1) + \rho_{\text{electron}} \epsilon_{\text{electron}} (\gamma_{\text{electron}} - 1)$ is the total plasma pressure, $\epsilon_{\text{ion,electron}}$ is the specific internal energy of ions and electrons and $\gamma_{\text{ion,electron}}$ is the adiabatic index (ratio of specific heats) for the ions and electrons. The quantity c_{fast} corresponds to the fastest wave speed over the entire simulation domain; divergence errors are advected out of the domain with this speed.

In order to track the electron temperature, USim evolves the electron entropy, defined as:

$$S_{\text{electron}} = P_{\text{electron}} n_{\text{electron}}^{-(\gamma_{\text{electron}}+1)}; \;\; n_{\text{electron}} = \frac{\rho}{m_{\text{electron}} + \frac{m_{\text{ion}}}{Z}}$$

Here, n_{electron} is the electron number density, m_{electron} is the electron mass, m_{ion} is the ion mass and Z is the ion charge state. with the fluid velocity, $\mathbf{u}$. In order to advect the electron entropy with the electron velocity, refer to *twoTemperatureMhdDednerEqn*. The method provided by *simpleTwoTemperatureMhdDednerEqn* is generally more robust and has lower computational cost than that provided by *twoTemperatureMhdDednerEqn*. If, for example, heating of electrons by (for example) magnetic dissipation is required, then this can be accomplished by adding source terms of the electron entropy equation, see, e.g. *mhdSrc*.

The electromagnetic fields are defined as:

$$\begin{aligned}
\mathbf{b} &= \mathbf{b}^{\text{plasma}} + \mathbf{b}^{\text{external}} = \mu_0^{-1/2} \left(\mathbf{B}^{\text{plasma}} + \mathbf{B}^{\text{external}} \right) \\
\mathbf{e} &= -\mathbf{u} \times \mathbf{b} + \mathbf{e}^{\text{external}} = \mu_0^{-1/2} \left(-\mathbf{u} \times \mathbf{B} + \mathbf{E}^{\text{external}} \right)
\end{aligned}$$

Here, $\mathbf{b}^{\text{plasma}}$ is the magnetic field induced in the plasma by the inductive electric field, $\mathbf{e}$, while $\mathbf{e}^{\text{external}}$ and $\mathbf{b}^{\text{external}}$ are electromagnetic fields computed "externally" to the ideal magnetohydrodynamic equations.

11.9.1 Parameters

`basementPressure` (float, optional) The minimum pressure allowed. Pressures below this value will be replaced with this value for primitive state, eigensystem and flux computations. Defaults to zero.

`basementDensity` (float, optional) The minimum density allowed. Densities below this value will be replaced with this value for primitive state, eigensystem and flux computations. Defaults to zero.

`gasGamma` (float, optional) Specifies the adiabatic index (ratio of specific heats) for the total pressure, γ. Defaults to 5/3.

`electronGamma` (float, optional) Specifies the adiabatic index (ratio of specific heats) for the electrons, γ_{electron}. Defaults to 5/3.

`electronMass` (float, optional) Specifies the electron mass, m_{electron}. Defaults to $(1836)^{-1}$.

`ionMass` (float, optional) Specifies the ion mass, m_{ion}. Defaults to 1.

`chargeState` (float, optional) Specifies the charge on an ion, Z. Defaults to 1.

`currentVector` (string, required) Specifies the name of the data structure containing the total (ion + electron) plasma current, $\mathbf{J}^{\text{plasma}}$.

`externalEfield` (string, optional) Specifies the name of the data structure containing the externally computed electric field, $\mathbf{e}^{\text{external}}$.

`externalBfield` (string, optional) Specifies the name of the data structure containing the externally computed magnetic field, $\mathbf{b}^{\text{external}}$.

11.9.2 Parent Updater Data

`in` (string vector, required)

`Vector of Conserved Quantities` (*nodalArray*, 10-components, required) The vector of conserved quantities, $\mathbf{q}$ has 10 entries:

0. ρ: mass density
1. $\rho\, u_{\hat{\mathbf{i}}} = \rho\mathbf{u}\cdot\hat{\mathbf{i}}$: momentum density in the $\hat{\mathbf{i}}$ direction
2. $\rho\, u_{\hat{\mathbf{j}}} = \rho\mathbf{u}\cdot\hat{\mathbf{j}}$: momentum density in the $\hat{\mathbf{j}}$ direction
3. $\rho\, u_{\hat{\mathbf{k}}} = \rho\mathbf{u}\cdot\hat{\mathbf{k}}$: momentum density in the $\hat{\mathbf{k}}$ direction
4. $E = \frac{P}{\gamma-1} + \frac{1}{2}\rho|\mathbf{u}|^2 + \frac{1}{2}|\mathbf{b}|^2$: total energy density
5. $b_{\hat{\mathbf{i}}} = \mathbf{b}\cdot\hat{\mathbf{i}} = \mu_0^{-1/2}\mathbf{B}\cdot\hat{\mathbf{i}}$: magnetic field normalized by permeability of free-space in the $\hat{\mathbf{i}}$ direction
6. $b_{\hat{\mathbf{j}}} = \mathbf{b}\cdot\hat{\mathbf{j}} = \mu_0^{-1/2}\mathbf{B}\cdot\hat{\mathbf{j}}$: magnetic field normalized by permeability of free-space in the $\hat{\mathbf{j}}$ direction
7. $b_{\hat{\mathbf{k}}} = \mathbf{b}\cdot\hat{\mathbf{k}} = \mu_0^{-1/2}\mathbf{B}\cdot\hat{\mathbf{k}}$: magnetic field normalized by permeability of free-space in the $\hat{\mathbf{k}}$ direction
8. ψ: correction potential
9. S_{electron}: electron entropy

`Fastest Wave Speed` (*dynVector*, 1-component, required) The fastest wave speed across the entire simulation domain, c_{fast}. Can be computed using *hyperbolic (1d, 2d, 3d)* (see below).

`Externally Computed Electric Field` (*nodalArray*, 3-components, optional) Additional terms in the generalized Ohm's law, $\mathbf{E}^{\text{external}}$, computed "externally" to the ideal magnetohydrodynamic system. The data structure containing $\mathbf{e}^{\text{external}}$ is specified by the "externalEField" option described below.

0. $e^{\text{external}}_{\hat{\mathbf{i}}} = \mathbf{e}^{\text{external}}\cdot\hat{\mathbf{i}} = \mu_0^{-1/2}\mathbf{E}^{\text{external}}\cdot\hat{\mathbf{i}}$: "externally" computed electric field normalized by permeability of free-space in the $\hat{\mathbf{i}}$ direction.
1. $e^{\text{external}}_{\hat{\mathbf{j}}} = \mathbf{e}^{\text{external}}\cdot\hat{\mathbf{j}} = \mu_0^{-1/2}\mathbf{E}^{\text{external}}\cdot\hat{\mathbf{j}}$:"externally" computed electric field normalized by permeability of free-space in the $\hat{\mathbf{j}}$ direction
2. $e^{\text{external}}_{\hat{\mathbf{k}}} = \mathbf{e}^{\text{external}}\cdot\hat{\mathbf{k}} = \mu_0^{-1/2}\mathbf{E}^{\text{external}}\cdot\hat{\mathbf{k}}$: "externally" computed electric field normalized by permeability of free-space in the $\hat{\mathbf{k}}$ direction

`Externally Computed Magnetic Field` (*nodalArray*, 3-components, optional) Additional contribution to the magnetic field, $\mathbf{b}^{\text{external}}$, which is not evolved by the induction equation, but does contribute to the Lorentz force and the work done on the plasma. The data structure containing $\mathbf{b}^{\text{external}}$ is specified by the "externalBField" option described below.

0. $b_{\hat{\mathbf{i}}}^{\text{external}} = \mathbf{b}^{\text{external}} \cdot \hat{\mathbf{i}} = \mu_0^{-1/2}\mathbf{B}^{\text{external}} \cdot \hat{\mathbf{i}}$: magnetic field normalized by permeability of free-space in the $\hat{\mathbf{i}}$ direction
1. $b_{\hat{\mathbf{j}}}^{\text{external}} = \mathbf{b}^{\text{external}} \cdot \hat{\mathbf{j}} = \mu_0^{-1/2}\mathbf{B}^{\text{external}} \cdot \hat{\mathbf{j}}$: magnetic field normalized by permeability of free-space in the $\hat{\mathbf{j}}$ direction
2. $b_{\hat{\mathbf{k}}}^{\text{external}} = \mathbf{b}^{\text{external}} \cdot \hat{\mathbf{k}} = \mu_0^{-1/2}\mathbf{B}^{\text{external}} \cdot \hat{\mathbf{k}}$: magnetic field normalized by permeability of free-space in the $\hat{\mathbf{k}}$ direction

`out` (string vector, required) For the mhdDednerEqn, one of four output variables are computed, depending on whether the equation is combined with an updater capable of computing fluxes (*classicMusclUpdater (1d, 2d, 3d)*), primitive variables (*computePrimitiveState(1d, 2d, 3d)*), the time step associated with the CFL condition (*timeStepRestrictionUpdater (1d, 2d, 3d)*) or the fastest wave speed in the grid (*hyperbolic (1d, 2d, 3d)*).

`Vector of Fluxes` (*nodalArray*, 9-components) When combined with an updater that computes $\nabla \cdot \mathcal{F}(\mathbf{w})$ (e.g. *classicMusclUpdater (1d, 2d, 3d)*), the equation system returns:

0. $\nabla \cdot \mathcal{F}(\rho)$: mass flux
1. $\nabla \cdot \mathcal{F}\left(\rho\, u_{\hat{\mathbf{i}}}\right)$: $\hat{\mathbf{i}}$ momentum flux
2. $\nabla \cdot \mathcal{F}\left(\rho\, u_{\hat{\mathbf{j}}}\right)$: $\hat{\mathbf{j}}$ momentum flux
3. $\nabla \cdot \mathcal{F}\left(\rho\, u_{\hat{\mathbf{k}}}\right)$: $\hat{\mathbf{k}}$ momentum flux
4. $\nabla \cdot \mathcal{F}(E)$: total energy flux
5. $\nabla \cdot \mathcal{F}\left(b_{\hat{\mathbf{i}}}\right)$: $\hat{\mathbf{i}}$ magnetic field flux
6. $\nabla \cdot \mathcal{F}\left(b_{\hat{\mathbf{j}}}\right)$: $\hat{\mathbf{j}}$ magnetic field flux
7. $\nabla \cdot \mathcal{F}\left(b_{\hat{\mathbf{k}}}\right)$: $\hat{\mathbf{k}}$ magnetic field flux
8. $\nabla \cdot \mathcal{F}(\psi)$: correction potential flux
9. $\nabla \cdot \mathcal{F}(S_{\text{electron}})$: electron entropy flux

`Vector of Primitive States` (*nodalArray*, 9-components) When combined with an updater that computes $\mathbf{w} = \mathbf{w}(\mathbf{q})$ (e.g. *computePrimitiveState(1d, 2d, 3d)*), the equation systen returns:

0. ρ: mass density
1. $u_{\hat{\mathbf{i}}} = \mathbf{u} \cdot \hat{\mathbf{i}}$: velocity in the $\hat{\mathbf{i}}$ direction
2. $u_{\hat{\mathbf{j}}} = \mathbf{u} \cdot \hat{\mathbf{j}}$: velocity in the $\hat{\mathbf{j}}$ direction
3. $u_{\hat{\mathbf{k}}} = \mathbf{u} \cdot \hat{\mathbf{k}}$: velocity in the $\hat{\mathbf{k}}$ direction
4. $P = \rho\epsilon(\gamma - 1)$: ideal gas pressure
5. $b_{\hat{\mathbf{i}}} = \mathbf{b} \cdot \hat{\mathbf{i}} = \mu_0^{-1/2}\mathbf{B} \cdot \hat{\mathbf{i}}$: magnetic field normalized by permeability of free-space in the $\hat{\mathbf{i}}$ direction
6. $b_{\hat{\mathbf{j}}} = \mathbf{b} \cdot \hat{\mathbf{j}} = \mu_0^{-1/2}\mathbf{B} \cdot \hat{\mathbf{j}}$: magnetic field normalized by permeability of free-space in the $\hat{\mathbf{j}}$ direction
7. $b_{\hat{\mathbf{k}}} = \mathbf{b} \cdot \hat{\mathbf{k}} = \mu_0^{-1/2}\mathbf{B} \cdot \hat{\mathbf{k}}$: magnetic field normalized by permeability of free-space in the $\hat{\mathbf{k}}$ direction

8. ψ: correction potential
9. P_{electron}: electron pressure

`Time Step` (*dynVector*, 1-component) When combined with *timeStepRestrictionUpdater (1d, 2d, 3d)*, the equation system returns the time step consisten with the CFL condition across the entire simulation domain.

`Fastest Wave Speed` (*dynVector*, 1-component) When combined with *hyperbolic (1d, 2d, 3d)*, the equation system returns the fastest wave speed across the entire simulation domain, c_{fast}.

11.9.3 Examples

The following block demonstrates the simpleTwoTemperatureMhdDednerEqn used in combination with *classicMusclUpdater (1d, 2d, 3d)* to compute $\nabla \cdot \mathcal{F}(\mathbf{w})$

```
 <Updater hyper>
   kind = classicMuscl1d
   onGrid = domain

# input data-structures
   in = [q,electricField]
# output data-structures
   out = [qnew]
# the time integration scheme, rk1 for first order runge-kutta
   timeIntegrationScheme = none
# the numerical flux to use
   numericalFlux = roeFlux
# CFL number to use
   cfl = 0.4
# Form of variables to limit
   variableForm = primitive
# Limiter to use
   limiter = [muscl,muscl]

   waveSpeeds = [waveSpeed]

# list of equations to solve
   equations = [mhd]

   <Equation mhd>
     kind = simpleTwoTemperatureMhdDednerEqn
     gasGamma = GAS_GAMMA
     electronGamma = $ELECTRON_GAMMA$
     basementDensity = $BASEMENT_DENSITY$
     basementPressure = $BASEMENT_PRESSURE$
     externalEfield = "electricField"
   </Equation>

 </Updater>
```

The following block demonstrates the simpleTwoTemperatureMhdDednerEqn used in combination with *computePrimitiveState(1d, 2d, 3d)* to compute $\mathbf{w}(\mathbf{q})$

```
 <Updater computePrimitiveState>
   kind = computePrimitiveState1d

   onGrid = domain
```

```
# input data-structures
   in = [q,electricField]

# ouput data-structures
   out = [w]

   <Equation mhd>
     kind = simpleTwoTemperatureMhdDednerEqn
     gasGamma = GAS_GAMMA
     electronGamma = $ELECTRON_GAMMA$
     basementDensity = $BASEMENT_DENSITY$
     basementPressure = $BASEMENT_PRESSURE$
     externalEfield = "electricField"
   </Equation>

 </Updater>
```

The following block demonstrates the simpleTwoTemperatureMhdDednerEqn used in combination with *timeStepRestrictionUpdater (1d, 2d, 3d)*, *hyperbolic (1d, 2d, 3d)* and *quadratic (1d, 2d, 3d)* to compute $dt_{\min}$, dt_{diff} and c_{fast} for resistive two-temperature MHD:

```
<Updater getHypDT>
 kind = timeStepRestrictionUpdater1d
  in = [q,electricField]
 onGrid = domain
 waveSpeeds = [waveSpeed]
 timeSteps = [diffDT]
 restrictions = [idealMhd]
 courantCondition = CFL

 <TimeStepRestriction idealMhd>
   kind = hyperbolic1d
   cfl = CFL
   model = simpleTwoTemperatureMhdDednerEqn
   gasGamma = GAS_GAMMA
   electronGamma = $ELECTRON_GAMMA$
   correctNans = true
   correct = true
   correctNans = true
   basementDensity = $BASEMENT_DENSITY$
   basementPressure = $BASEMENT_PRESSURE$
   externalEfield = "electricField"
   storeTimeStep = False
 </TimeStepRestriction>

</Updater>
```

11.10 twoTemperatureMhdDednerEqn

Defines the equations of ideal compressible magnetohydrodynamics with divergence cleaning and an electron entropy equation:

$$\begin{aligned}
\frac{\partial \rho}{\partial t} + \nabla \cdot [\rho \, \mathbf{u}] &= 0 \\
\frac{\partial \rho \, \mathbf{u}}{\partial t} + \nabla \cdot \left[\rho \, \mathbf{u} \, \mathbf{u}^T - \mathbf{b} \, \mathbf{b}^T + \mathbb{I} \left(P_{\text{tot}} + \tfrac{1}{2} |\mathbf{b}|^2 \right) \right] &= 0 \\
\frac{\partial E}{\partial t} + \nabla \cdot [(E + P) \, \mathbf{u} + \mathbf{e} \times \mathbf{b}] &= 0 \\
\frac{\partial \mathbf{b}^{\text{plasma}}}{\partial t} + \nabla \times \mathbf{e} + \nabla \psi &= 0 \\
\frac{\partial \psi}{\partial t} + \nabla \cdot \left[c_{\text{fast}}^2 \mathbf{b} \right] &= 0 \\
\frac{\partial S_{\text{electron}}}{\partial t} + \nabla \cdot [S_{\text{electron}} \, \mathbf{u}_{\text{electron}}] &= 0
\end{aligned}$$

Here, $\mathbb{I}$ is the identity matrix, $P_{\text{tot}} = P_{\text{ion}} + P_{\text{electron}} = \rho_{\text{ion}} \epsilon_{\text{ion}} (\gamma_{\text{ion}} - 1) + \rho_{\text{electron}} \epsilon_{\text{electron}} (\gamma_{\text{electron}} - 1)$ is the total plasma pressure, $\epsilon_{\text{ion,electron}}$ is the specific internal energy of ions and electrons and $\gamma_{\text{ion,electron}}$ is the adiabatic index (ratio of specific heats) for the ions and electrons. The quantity c_{fast} corresponds to the fastest wave speed over the entire simulation domain; divergence errors are advected out of the domain with this speed.

In order to track the electron temperature, USim evolves the electron entropy, defined as:

$$S_{\text{electron}} = P_{\text{electron}} n_{\text{electron}}^{-(\gamma_{\text{electron}}+1)}; \;\; n_{\text{electron}} = \frac{\rho}{m_{\text{electron}} + \frac{m_{\text{ion}}}{Z}}$$

Here, n_{electron} is the electron number density, m_{electron} is the electron mass, m_{ion} is the ion mass and Z is the ion charge state. The electron entropy is advected by the electron velocity, $\mathbf{u}_{\text{electron}}$, computed as:

$$\mathbf{u}_{\text{electron}} = -\frac{\mathbf{J}^{\text{plasma}} - q Z m_{\text{ion}}^{-1} \rho \mathbf{u}}{q n_{\text{electron}}}; \;\; \mathbf{J}^{\text{plasma}} = \mu_0^{-1/2} \nabla \times \mathbf{b}^{\text{plasma}} = \mu_0^{-1} \nabla \times \mathbf{B}^{\text{plasma}}$$

Here, $\mathbf{J}^{\text{plasma}}$ is the total (ion+electron) plasma current and q is the fundamental change ($-q$ is the charge on an electron). As defined above, the electron entropy is advected with the electron density. If, for example, heating of electrons by (for example) magnetic dissipation is required, then this can be accomplished by adding source terms of the electron entropy equation, see, e.g. *mhdSrc*.

The electromagnetic fields are defined as:

$$\begin{aligned}
\mathbf{b} &= \mathbf{b}^{\text{plasma}} + \mathbf{b}^{\text{external}} = \mu_0^{-1/2} \left(\mathbf{B}^{\text{plasma}} + \mathbf{B}^{\text{external}} \right) \\
\mathbf{e} &= -\mathbf{u} \times \mathbf{b} + \mathbf{e}^{\text{external}} = \mu_0^{-1/2} \left(-\mathbf{u} \times \mathbf{B} + \mathbf{E}^{\text{external}} \right)
\end{aligned}$$

Here, $\mathbf{b}^{\text{plasma}}$ is the magnetic field induced in the plasma by the inductive electric field, $\mathbf{e}$, while $\mathbf{e}^{\text{external}}$ and $\mathbf{b}^{\text{external}}$ are electromagnetic fields computed "externally" to the ideal magnetohydrodynamic equations.

11.10.1 Parameters

`basementPressure` (float, optional) The minimum pressure allowed. Pressures below this value will be replaced with this value for primitive state, eigensystem and flux computations. Defaults to zero.

`basementDensity` (float, optional) The minimum density allowed. Densities below this value will be replaced with this value for primitive state, eigensystem and flux computations. Defaults to zero.

`gasGamma` (float, optional) Specifies the adiabatic index (ratio of specific heats) for the total pressure, γ. Defaults to 5/3.

`electronGamma` (float, optional) Specifies the adiabatic index (ratio of specific heats) for the electrons, γ_{electron}. Defaults to 5/3.

`electronMass` (float, optional) Specifies the electron mass, m_{electron}. Defaults to $(1836)^{-1}$.

`ionMass` (float, optional) Specifies the ion mass, m_{ion}. Defaults to 1.

`chargeState` (float, optional) Specifies the charge on an ion, Z. Defaults to 1.

`currentVector` (string, required) Specifies the name of the data structure containing the total (ion + electron) plasma current, $\mathbf{J}^{\text{plasma}}$.

`externalEfield` (string, optional) Specifies the name of the data structure containing the externally computed electric field, $\mathbf{e}^{\text{external}}$.

`externalBfield` (string, optional) Specifies the name of the data structure containing the externally computed magnetic field, $\mathbf{b}^{\text{external}}$.

11.10.2 Parent Updater Data

`in` (string vector, required)

`Vector of Conserved Quantities` (*nodalArray*, 10-components, required) The vector of conserved quantities, $\mathbf{q}$ has 10 entries:

0. ρ: mass density
1. $\rho\, u_{\hat{\mathbf{i}}} = \rho\mathbf{u}\cdot\hat{\mathbf{i}}$: momentum density in the $\hat{\mathbf{i}}$ direction
2. $\rho\, u_{\hat{\mathbf{j}}} = \rho\mathbf{u}\cdot\hat{\mathbf{j}}$: momentum density in the $\hat{\mathbf{j}}$ direction
3. $\rho\, u_{\hat{\mathbf{k}}} = \rho\mathbf{u}\cdot\hat{\mathbf{k}}$: momentum density in the $\hat{\mathbf{k}}$ direction
4. $E = \frac{P}{\gamma-1} + \frac{1}{2}\rho|\mathbf{u}|^2 + \frac{1}{2}|\mathbf{b}|^2$: total energy density
5. $b_{\hat{\mathbf{i}}} = \mathbf{b}\cdot\hat{\mathbf{i}} = \mu_0^{-1/2}\mathbf{B}\cdot\hat{\mathbf{i}}$: magnetic field normalized by permeability of free-space in the $\hat{\mathbf{i}}$ direction
6. $b_{\hat{\mathbf{j}}} = \mathbf{b}\cdot\hat{\mathbf{j}} = \mu_0^{-1/2}\mathbf{B}\cdot\hat{\mathbf{j}}$: magnetic field normalized by permeability of free-space in the $\hat{\mathbf{j}}$ direction
7. $b_{\hat{\mathbf{k}}} = \mathbf{b}\cdot\hat{\mathbf{k}} = \mu_0^{-1/2}\mathbf{B}\cdot\hat{\mathbf{k}}$: magnetic field normalized by permeability of free-space in the $\hat{\mathbf{k}}$ direction
8. ψ: correction potential
9. S_{electron}: electron entropy

`Current Density` (*nodalArray*, 3-components, required) The total (ion and electron) current in the plasma, typically calculated from from pre-Maxwell form of Ampere's law, $\mathbf{J}^{\text{plasma}} = \mu_0^{1/2}\nabla\times\mathbf{b}^{\text{plasma}}$, which can be computed through, e.g. *vector (1d, 2d, 3d)*. The data structure containing $\mathbf{J}^{\text{plasma}}$ is specified by the "currentVector" option described below.

0. $J^{\text{plasma}}_{\hat{\mathbf{i}}} = \mathbf{J}^{\text{plasma}}\cdot\hat{\mathbf{i}}$: total (ion and electron) current in the plasma in the $\hat{\mathbf{i}}$ direction.
1. $J^{\text{plasma}}_{\hat{\mathbf{j}}} = \mathbf{J}^{\text{plasma}}\cdot\hat{\mathbf{j}}$: total (ion and electron) current in the plasma in the $\hat{\mathbf{j}}$ direction
2. $J^{\text{plasma}}_{\hat{\mathbf{k}}} = \mathbf{J}^{\text{plasma}}\cdot\hat{\mathbf{k}}$: total (ion and electron) current in the plasma in the $\hat{\mathbf{k}}$ direction

`Fastest Wave Speed` (*dynVector*, 1-component, required) The fastest wave speed across the entire simulation domain, c_{fast}. Can be computed using *hyperbolic (1d, 2d, 3d)* (see below).

`Externally Computed Electric Field` (*nodalArray*, 3-components, optional) Additional terms in the generalized Ohm's law, $\mathbf{E}^{\text{external}}$, computed "externally" to the ideal magnetohydrodynamic system. The data structure containing $\mathbf{e}^{\text{external}}$ is specified by the "externalEField" option described below.

0. $e^{\text{external}}_{\hat{\mathbf{i}}} = \mathbf{e}^{\text{external}} \cdot \hat{\mathbf{i}} = \mu_0^{-1/2}\mathbf{E}^{\text{external}} \cdot \hat{\mathbf{i}}$: "externally" computed electric field normalized by permeability of free-space in the $\hat{\mathbf{i}}$ direction.
1. $e^{\text{external}}_{\hat{\mathbf{j}}} = \mathbf{e}^{\text{external}} \cdot \hat{\mathbf{j}} = \mu_0^{-1/2}\mathbf{E}^{\text{external}} \cdot \hat{\mathbf{j}}$:"externally" computed electric field normalized by permeability of free-space in the $\hat{\mathbf{j}}$ direction
2. $e^{\text{external}}_{\hat{\mathbf{k}}} = \mathbf{e}^{\text{external}} \cdot \hat{\mathbf{k}} = \mu_0^{-1/2}\mathbf{E}^{\text{external}} \cdot \hat{\mathbf{k}}$: "externally" computed electric field normalized by permeability of free-space in the $\hat{\mathbf{k}}$ direction

`Externally Computed Magnetic Field` (*nodalArray*, 3-components, optional) Additional contribution to the magnetic field, $\mathbf{b}^{\text{external}}$, which is not evolved by the induction equation, but does contribute to the Lorentz force and the work done on the plasma. The data structure containing $\mathbf{b}^{\text{external}}$ is specified by the "externalBField" option described below.

0. $b^{\text{external}}_{\hat{\mathbf{i}}} = \mathbf{b}^{\text{external}} \cdot \hat{\mathbf{i}} = \mu_0^{-1/2}\mathbf{B}^{\text{external}} \cdot \hat{\mathbf{i}}$: magnetic field normalized by permeability of free-space in the $\hat{\mathbf{i}}$ direction
1. $b^{\text{external}}_{\hat{\mathbf{j}}} = \mathbf{b}^{\text{external}} \cdot \hat{\mathbf{j}} = \mu_0^{-1/2}\mathbf{B}^{\text{external}} \cdot \hat{\mathbf{j}}$: magnetic field normalized by permeability of free-space in the $\hat{\mathbf{j}}$ direction
2. $b^{\text{external}}_{\hat{\mathbf{k}}} = \mathbf{b}^{\text{external}} \cdot \hat{\mathbf{k}} = \mu_0^{-1/2}\mathbf{B}^{\text{external}} \cdot \hat{\mathbf{k}}$: magnetic field normalized by permeability of free-space in the $\hat{\mathbf{k}}$ direction

`out` (string vector, required) For the mhdDednerEqn, one of four output variables are computed, depending on whether the equation is combined with an updater capable of computing fluxes (*classicMusclUpdater (1d, 2d, 3d)*), primitive variables (*computePrimitiveState(1d, 2d, 3d)*), the time step associated with the CFL condition (*timeStepRestrictionUpdater (1d, 2d, 3d)*) or the fastest wave speed in the grid (*hyperbolic (1d, 2d, 3d)*).

`Vector of Fluxes` (*nodalArray*, 9-components) When combined with an updater that computes $\nabla \cdot \mathcal{F}(\mathbf{w})$ (e.g. *classicMusclUpdater (1d, 2d, 3d)*), the equation system returns:

0. $\nabla \cdot \mathcal{F}(\rho)$: mass flux
1. $\nabla \cdot \mathcal{F}\left(\rho\, u_{\hat{\mathbf{i}}}\right)$: $\hat{\mathbf{i}}$ momentum flux
2. $\nabla \cdot \mathcal{F}\left(\rho\, u_{\hat{\mathbf{j}}}\right)$: $\hat{\mathbf{j}}$ momentum flux
3. $\nabla \cdot \mathcal{F}\left(\rho\, u_{\hat{\mathbf{k}}}\right)$: $\hat{\mathbf{k}}$ momentum flux
4. $\nabla \cdot \mathcal{F}(E)$: total energy flux
5. $\nabla \cdot \mathcal{F}\left(b_{\hat{\mathbf{i}}}\right)$: $\hat{\mathbf{i}}$ magnetic field flux
6. $\nabla \cdot \mathcal{F}\left(b_{\hat{\mathbf{j}}}\right)$: $\hat{\mathbf{j}}$ magnetic field flux
7. $\nabla \cdot \mathcal{F}\left(b_{\hat{\mathbf{k}}}\right)$: $\hat{\mathbf{k}}$ magnetic field flux
8. $\nabla \cdot \mathcal{F}(\psi)$: correction potential flux
9. $\nabla \cdot \mathcal{F}(S_{\text{electron}})$: electron entropy flux

`Vector of Primitive States` (*nodalArray*, 9-components) When combined with an updater that computes $\mathbf{w} = \mathbf{w}(\mathbf{q})$ (e.g. *computePrimitiveState(1d, 2d, 3d)*), the equation systen returns:

0. ρ: mass density
1. $u_{\hat{\mathbf{i}}} = \mathbf{u} \cdot \hat{\mathbf{i}}$: velocity in the $\hat{\mathbf{i}}$ direction
2. $u_{\hat{\mathbf{j}}} = \mathbf{u} \cdot \hat{\mathbf{j}}$: velocity in the $\hat{\mathbf{j}}$ direction
3. $u_{\hat{\mathbf{k}}} = \mathbf{u} \cdot \hat{\mathbf{k}}$: velocity in the $\hat{\mathbf{k}}$ direction
4. $P = \rho\epsilon(\gamma - 1)$: ideal gas pressure
5. $b_{\hat{\mathbf{i}}} = \mathbf{b} \cdot \hat{\mathbf{i}} = \mu_0^{-1/2} \mathbf{B} \cdot \hat{\mathbf{i}}$: magnetic field normalized by permeability of free-space in the $\hat{\mathbf{i}}$ direction
6. $b_{\hat{\mathbf{j}}} = \mathbf{b} \cdot \hat{\mathbf{j}} = \mu_0^{-1/2} \mathbf{B} \cdot \hat{\mathbf{j}}$: magnetic field normalized by permeability of free-space in the $\hat{\mathbf{j}}$ direction
7. $b_{\hat{\mathbf{k}}} = \mathbf{b} \cdot \hat{\mathbf{k}} = \mu_0^{-1/2} \mathbf{B} \cdot \hat{\mathbf{k}}$: magnetic field normalized by permeability of free-space in the $\hat{\mathbf{k}}$ direction
8. ψ: correction potential
9. P_{electron}: electron pressure

`Time Step` (*dynVector*, 1-component) When combined with *timeStepRestrictionUpdater (1d, 2d, 3d)*, the equation system returns the time step consisten with the CFL condition across the entire simulation domain.

`Fastest Wave Speed` (*dynVector*, 1-component) When combined with *hyperbolic (1d, 2d, 3d)*, the equation system returns the fastest wave speed across the entire simulation domain, c_{fast}.

11.10.3 Examples

The following block demonstrates the twoTemperatureMhdDednerEqn used in combination with *classicMusclUpdater (1d, 2d, 3d)* to compute $\nabla \cdot \mathcal{F}(\mathbf{w})$, including resistive effects

```
 <Updater hyper>
   kind = classicMuscl1d
   onGrid = domain

# input data-structures
   in = [q,electricField,current,chargeState,resistivity]
# output data-structures
   out = [qnew]
# the time integration scheme, rk1 for first order runge-kutta
   timeIntegrationScheme = none
# the numerical flux to use
   numericalFlux = roeFlux
# CFL number to use
   cfl = 0.4
# Form of variables to limit
   variableForm = primitive
# Limiter to use
   limiter = [muscl,muscl,muscl,muscl,muscl]

   waveSpeeds = [waveSpeed]

# list of equations to solve
   equations = [mhd]

# list of sources to add
   source = [mhdSource]
```

```
    <Equation mhd>
      kind = twoTemperatureMhdDednerEqn
      gasGamma = GAS_GAMMA
      electronGamma = $ELECTRON_GAMMA$
      basementDensity = $BASEMENT_DENSITY$
      basementPressure = $BASEMENT_PRESSURE$
      externalEfield = "electricField"
      currentVector = "current"
    </Equation>

    <Source mhdSource>
      kind = mhdSrc
      model = twoTemperatureMhdDednerEqn
      externalEfield = true
      inputVariables = [q, electricField,current,chargeState,resistivity]
      ionMass = ION_MASS
      fundamentalCharge = FUNDAMENTAL_CHARGE
    </Source>

  </Updater>
```

The following block demonstrates the twoTemperatureMhdDednerEqn used in combination with *computePrimitiveState(1d, 2d, 3d)* to compute $\mathbf{w}\left(\mathbf{q}\right)$

```
  <Updater computePrimitiveState>
    kind = computePrimitiveState1d

    onGrid = domain
# input data-structures
    in = [q,electricField,current,chargeState,resistivity]

# ouput data-structures
    out = [w]

    <Equation mhd>
      kind = twoTemperatureMhdDednerEqn
      gasGamma = GAS_GAMMA
      electronGamma = $ELECTRON_GAMMA$
      basementDensity = $BASEMENT_DENSITY$
      basementPressure = $BASEMENT_PRESSURE$
      externalEfield = "electricField"
      currentVector = "current"
    </Equation>

  </Updater>
```

The following block demonstrates the twoTemperatureMhdDednerEqn used in combination with *timeStepRestrictionUpdater (1d, 2d, 3d)*, *hyperbolic (1d, 2d, 3d)* and *quadratic (1d, 2d, 3d)* to compute $dt_{\min}$, dt_{diff} and c_{fast} for resistive two-temperature MHD:

```
  <Updater getHypDT>
   kind = timeStepRestrictionUpdater1d
    in = [q,electricField,current,chargeState,resistivity]
   onGrid = domain
   waveSpeeds = [waveSpeed]
   timeSteps = [diffDT]
   restrictions = [idealMhd,quadratic]
   courantCondition = CFL
```

```
    <TimeStepRestriction idealMhd>
      kind = hyperbolic1d
      cfl = CFL
      model = twoTemperatureMhdDednerEqn
      gasGamma = GAS_GAMMA
      electronGamma = $ELECTRON_GAMMA$
      correctNans = true
      correct = true
      correctNans = true
      basementDensity = $BASEMENT_DENSITY$
      basementPressure = $BASEMENT_PRESSURE$
      externalEfield = "electricField"
      currentVector = "current"
      storeTimeStep = False
    </TimeStepRestriction>

    <TimeStepRestriction quadratic>
      kind = quadratic1d
      in = [resistivity]
      cfl = CFL
    </TimeStepRestriction>
  </Updater>
```

The following equations can be used to simulate Maxwell's equations and non-neutral plasmas:

11.11 maxwellEqn

Fluxes and eigensystem for Maxwell's equations in vacuum with divergence cleaning.

$$\frac{\partial \mathbf{E}}{\partial t} + c^2 \nabla \times \mathbf{B} + \nabla \Phi = 0$$
$$\frac{\partial \mathbf{B}}{\partial t} - \nabla \times \mathbf{E} + \nabla \psi = 0$$
$$\frac{\partial \Phi}{\partial t} + \nabla \cdot \left[c_{\text{fast}}^2 \mathbf{E} \right] = 0$$
$$\frac{\partial \psi}{\partial t} + \nabla \cdot \left[c_{\text{fast}}^2 \mathbf{B} \right] = 0$$

Coupling of Maxwell's equations to a plasma is accomplished using *current*.

11.11.1 Parameters

`c0` (float) The speed of light

`gamma` (float) Magnetic correction potential propagation factor. γc_0 is the magnetic correction potential propagation speed.

`chi` (float) Electric correction potential propagation factor. χc_0 is the correction propagation speed.

11.11.2 Parent Updater Data

`in` (string vector, required)

> **`Vector of Conserved Quantities`** (*nodalArray*, **8-components, required**) The vector of conserved quantities, $\mathbf{q}$ has 8 entries:

0. $E_{\hat{\mathbf{i}}} = \mathbf{E} \cdot \hat{\mathbf{i}}$: electric field in the $\hat{\mathbf{i}}$ direction.
1. $E_{\hat{\mathbf{j}}} = \mathbf{E} \cdot \hat{\mathbf{j}}$: electric field in the $\hat{\mathbf{j}}$ direction
2. $E_{\hat{\mathbf{k}}} = \mathbf{E} \cdot \hat{\mathbf{k}}$: electric field in the $\hat{\mathbf{k}}$ direction
3. $B_{\hat{\mathbf{i}}} = \mathbf{B} \cdot \hat{\mathbf{i}}$: magnetic field in the $\hat{\mathbf{i}}$ direction
4. $B_{\hat{\mathbf{j}}} = \mathbf{B} \cdot \hat{\mathbf{j}}$: magnetic field in the $\hat{\mathbf{j}}$ direction
5. $B_{\hat{\mathbf{k}}} = \mathbf{B} \cdot \hat{\mathbf{k}}$: magnetic field in the $\hat{\mathbf{k}}$ direction
6. Φ electric field correction potential
7. Ψ magnetic field correction potential

`out` (string vector, required) For the maxwellEqn, one of three output variables are computed, depending on whether the equation is combined with an updater capable of computing fluxes (*classicMusclUpdater (1d, 2d, 3d)*), the time step associated with the CFL condition (*timeStepRestrictionUpdater (1d, 2d, 3d)*) or the fastest wave speed in the grid (*hyperbolic (1d, 2d, 3d)*).

`Vector of Fluxes` (*nodalArray*, 9-components) When combined with an updater that computes $\nabla \cdot \mathcal{F}(\mathbf{w})$ (e.g. *classicMusclUpdater (1d, 2d, 3d)*), the equation system returns:

0. $\nabla \cdot \mathcal{F}\left(E_{\hat{\mathbf{i}}}\right)$: $\hat{\mathbf{i}}$ electric field flux
1. $\nabla \cdot \mathcal{F}\left(E_{\hat{\mathbf{j}}}\right)$: $\hat{\mathbf{i}}$ electric field flux
2. $\nabla \cdot \mathcal{F}\left(E_{\hat{\mathbf{k}}}\right)$: $\hat{\mathbf{j}}$ electric field flux
3. $\nabla \cdot \mathcal{F}\left(B_{\hat{\mathbf{i}}}\right)$: $\hat{\mathbf{i}}$ magnetic field flux
4. $\nabla \cdot \mathcal{F}\left(B_{\hat{\mathbf{j}}}\right)$: $\hat{\mathbf{i}}$ magnetic field flux
5. $\nabla \cdot \mathcal{F}\left(B_{\hat{\mathbf{k}}}\right)$: $\hat{\mathbf{j}}$ magnetic field flux
6. $\nabla \cdot \mathcal{F}(\psi)$: electric correction potential flux
7. $\nabla \cdot \mathcal{F}(\psi)$: magnetic correction potential flux

`Time Step` (*dynVector*, 1-component) When combined with *timeStepRestrictionUpdater (1d, 2d, 3d)*, the equation system returns the time step consisten with the CFL condition across the entire simulation domain.

`Fastest Wave Speed` (*dynVector*, 1-component) When combined with *hyperbolic (1d, 2d, 3d)*, the equation system returns the fastest wave speed across the entire simulation domain, c_{fast}.

11.11.3 Example

The following block demonstrates the maxwellEqn used in combination with *classicMusclUpdater (1d, 2d, 3d)* to compute $\nabla \cdot \mathcal{F}(\mathbf{w})$:

```
<Updater hyperEm>
  kind = classicMuscl3d
  onGrid = domain
  timeIntegrationScheme = none
  numericalFlux = fWaveFlux
  limiterType = component
  limiter = [minmod, none, none]
  variableForm = conservative
```

```
    in = [em, electrons, ions]
    out = [emNew]

    cfl = CFL
    equations = [maxwell]

    <Equation maxwell>
      kind = maxwellEqn
      c0 = SPEED_OF_LIGHT
      gamma = BP
      chi = 0.0
    </Equation>

  </Updater>
```

The following block demonstrates the maxwellEqn used in combination with *timeStepRestrictionUpdater (1d, 2d, 3d)* and *hyperbolic (1d, 2d, 3d)* to compute c_{fast}:

```
  <Updater getWaveSpeed>
   kind = timeStepRestrictionUpdater2d
   in = [q]
   waveSpeeds = [waveSpeed]
   onGrid = domain
   restrictions = [hyperbolic]
   cfl = CFL
   courantCondition = CFL

   <TimeStepRestriction hyperbolic>
    kind = hyperbolic2d
    model = maxwellEqn
    cfl = CFL
    c0 = SPEED_OF_LIGHT
    gamma = 0.0
    chi = 0.0
    includeInTimeStep = False
  </TimeStepRestriction>
 </Updater>
```

11.12 maxwellDednerEqn

Fluxes and eigensystem for Maxwell's equations in vacuum with divergence cleaning.

$$\begin{aligned}
\frac{\partial \mathbf{E}}{\partial t} + c^2 \nabla \times \mathbf{B} + \nabla \Phi &= 0 \\
\frac{\partial \mathbf{B}}{\partial t} - \nabla \times \mathbf{E} + \nabla \psi &= 0 \\
\frac{\partial \Phi}{\partial t} + \nabla \cdot \left[c_{\text{fast}}^2 \mathbf{E} \right] &= 0 \\
\frac{\partial \psi}{\partial t} + \nabla \cdot \left[c_{\text{fast}}^2 \mathbf{B} \right] &= 0
\end{aligned}$$

Coupling of Maxwell's equations to a plasma is accomplished using *current*.

11.12.1 Parameters

`mu0` (float, optional) Permeability of free space. Default value is 1.256e-06.

`epsilon0` (float, optional) Permittivity of free space. Default value is 8.854e-12.

`cfl` (float, optional) CFL number. Default value is 1.0.

11.12.2 Parent Updater Data

`in` (string vector, required)

`Vector of Conserved Quantities` (*nodalArray*, 8-components, required) The vector of conserved quantities, $\mathbf{q}$ has 8 entries:

0. $E_{\hat{\mathbf{i}}} = \mathbf{E} \cdot \hat{\mathbf{i}}$: electric field in the $\hat{\mathbf{i}}$ direction.
1. $E_{\hat{\mathbf{j}}} = \mathbf{E} \cdot \hat{\mathbf{j}}$: electric field in the $\hat{\mathbf{j}}$ direction
2. $E_{\hat{\mathbf{k}}} = \mathbf{E} \cdot \hat{\mathbf{k}}$: electric field in the $\hat{\mathbf{k}}$ direction
3. $B_{\hat{\mathbf{i}}} = \mathbf{B} \cdot \hat{\mathbf{i}}$: magnetic field in the $\hat{\mathbf{i}}$ direction
4. $B_{\hat{\mathbf{j}}} = \mathbf{B} \cdot \hat{\mathbf{j}}$: magnetic field in the $\hat{\mathbf{j}}$ direction
5. $B_{\hat{\mathbf{k}}} = \mathbf{B} \cdot \hat{\mathbf{k}}$: magnetic field in the $\hat{\mathbf{k}}$ direction
6. Φ electric field correction potential
7. Ψ magnetic field correction potential

`Fastest Wave Speed` (*dynVector*, 1-component, required) The fastest wave speed across the entire simulation domain, c_{fast}. Can be computed using *hyperbolic (1d, 2d, 3d)* (see below).

`out` (string vector, required) For the maxwellDednerEqn, one of three output variables are computed, depending on whether the equation is combined with an updater capable of computing fluxes (*classicMusclUpdater (1d, 2d, 3d)*), the time step associated with the CFL condition (*timeStepRestrictionUpdater (1d, 2d, 3d)*) or the fastest wave speed in the grid (*hyperbolic (1d, 2d, 3d)*).

`Vector of Fluxes` (*nodalArray*, 9-components) When combined with an updater that computes $\nabla \cdot \mathcal{F}(\mathbf{w})$ (e.g. *classicMusclUpdater (1d, 2d, 3d)*), the equation system returns:

0. $\nabla \cdot \mathcal{F}\left(E_{\hat{\mathbf{i}}}\right)$: $\hat{\mathbf{i}}$ electric field flux
1. $\nabla \cdot \mathcal{F}\left(E_{\hat{\mathbf{j}}}\right)$: $\hat{\mathbf{i}}$ electric field flux
2. $\nabla \cdot \mathcal{F}\left(E_{\hat{\mathbf{k}}}\right)$: $\hat{\mathbf{j}}$ electric field flux
3. $\nabla \cdot \mathcal{F}\left(B_{\hat{\mathbf{i}}}\right)$: $\hat{\mathbf{i}}$ magnetic field flux
4. $\nabla \cdot \mathcal{F}\left(B_{\hat{\mathbf{j}}}\right)$: $\hat{\mathbf{i}}$ magnetic field flux
5. $\nabla \cdot \mathcal{F}\left(B_{\hat{\mathbf{k}}}\right)$: $\hat{\mathbf{j}}$ magnetic field flux
6. $\nabla \cdot \mathcal{F}(\psi)$: electric correction potential flux
7. $\nabla \cdot \mathcal{F}(\psi)$: magnetic correction potential flux

`Time Step` (*dynVector*, 1-component) When combined with *timeStepRestrictionUpdater (1d, 2d, 3d)*, the equation system returns the time step consisten with the CFL condition across the entire simulation domain.

`Fastest Wave Speed` (*dynVector*, 1-component) When combined with *hyperbolic (1d, 2d, 3d)*, the equation system returns the fastest wave speed across the entire simulation domain, c_{fast}.

11.12.3 Example

The following block demonstrates the maxwellDednerEqn used in combination with *classicMusclUpdater (1d, 2d, 3d)* to compute $\nabla \cdot \mathcal{F}(\mathbf{w})$:

```
<Updater hyper>
  kind=classicMuscl2d
  onGrid=domain
  timeIntegrationScheme=none
  numericalFlux=hlleFlux
  limiter=[none]
  variableForm=conservative
  preservePositivity=false
  in=[q]
  out=[qNew]
  waveSpeeds=[waveSpeed]
  cfl=0.4
  equations=[maxwell]

  <Equation maxwell>
    kind=maxwellDednerEqn
    epsilon0=1.0
    mu0=1.0
    cfl=0.4
  </Equation>

</Updater>
```

The following block demonstrates the maxwellDednerEqn used in combination with *timeStepRestrictionUpdater (1d, 2d, 3d)* and *hyperbolic (1d, 2d, 3d)* to compute c_{fast}:

```
<Updater getWaveSpeed>
  kind=timeStepRestrictionUpdater2d
  in=[q]
  waveSpeeds=[waveSpeed]
  onGrid=domain
  restrictions=[hyperbolic]
  cfl=0.4
  courantCondition=0.4

  <TimeStepRestriction hyperbolic>
    kind=hyperbolic2d
    model=maxwellEqn
    cfl=0.4
    c0=1.0
    gamma=0.0
    chi=0.0
    includeInTimeStep=False
  </TimeStepRestriction>

</Updater>
```

11.13 gasDynamicMaxwellDednerEqn

Defines the equations of inviscid fluid dynamics coupled to Maxwell's equations in source term form with divergence cleaning:

$$\frac{\partial \rho}{\partial t} + \nabla \cdot [\rho \, \mathbf{u}] = 0$$

$$\frac{\partial (\rho \mathbf{u})}{\partial t} + \nabla \cdot \left[\rho \, \mathbf{u} \, \mathbf{u}^T + \mathbb{I} P\right] = \sum_{\text{species}} \left(q^{\text{species}} \mathbf{E} + \mathbf{J}^{\text{species}} \times \mathbf{B}\right)$$

$$\frac{\partial E}{\partial t} + \nabla \cdot [(E + P) \, \mathbf{u}] = \sum_{\text{species}} \mathbf{J}^{\text{species}} \cdot \mathbf{E}$$

$$\frac{\partial \mathbf{B}^{\text{plasma}}}{\partial t} + \nabla \times \mathbf{E} + \nabla \Psi = 0$$

$$\frac{\partial \mathbf{E}^{\text{plasma}}}{\partial t} - c^2 \nabla \times \mathbf{B} + \nabla \Phi = -\epsilon_0^{-1} \sum_{\text{species}} \mathbf{J}^{\text{species}}$$

$$\frac{\partial \Psi}{\partial t} + \nabla \cdot \left[c_{\text{fast}}^2 \mathbf{B}^{\text{plasma}}\right] = 0$$

$$\frac{\partial \Phi}{\partial t} + \nabla \cdot \left[c_{\text{fast}}^2 \mathbf{E}^{\text{plasma}}\right] = \sum_{\text{species}} q^{\text{species}}$$

Here, q^{species} is the species charge density, $\mathbf{J}^{\text{species}}$ is the species current density, $\mathbb{I}$ is the identity matrix, $P = \rho\epsilon(\gamma - 1)$ is the pressure of an ideal gas, ϵ is the specific internal energy and γ is the adiabatic index (ratio of specific heats). The quantity c_{fast} corresponds to the fastest wave speed over the entire simulation domain; divergence errors are advected out of the domain with this speed.

In order to integrate these equations, USim casts them into flux-conservative form using the following standard identities (note that the use of these identities does not require an assumption of quasi-neutrality):

$$\sum_{\text{species}} \left(q^{\text{species}} \mathbf{E} + \mathbf{J}^{\text{species}} \times \mathbf{B}\right) = -\frac{\partial c^{-2} \mathbf{S}^{\text{EM}}}{\partial t} + \nabla \cdot \mathcal{T}^{\text{EM}}$$

$$\sum_{\text{species}} \mathbf{J}^{\text{species}} \cdot \mathbf{E} = -\frac{\partial E^{\text{EM}}}{\partial t} - \nabla \cdot \mathbf{S}^{\text{EM}}$$

Here, $\mathcal{T}^{\text{EM}}$ is the electromagnetic stress tensor and $\mathbf{S}^{\text{EM}}$ is the electromagnetic energy (Poynting) flux vector, which are defined as:

$$\mathcal{T}^{\text{EM}} = \frac{1}{\mu_0} \left(\frac{\mathbf{E}\mathbf{E}^T}{c^2} + \mathbf{B}\mathbf{B}^T\right) + \mathbb{I} E_{\text{EM}} = \frac{\mathbf{e}\mathbf{e}^T}{c^2} + \mathbf{b}\mathbf{b}^T + \mathbb{I} E_{\text{EM}}$$

$$\mathbf{S}^{\text{EM}} = \mu_0^{-1} \mathbf{E} \times \mathbf{B} = \mathbf{e} \times \mathbf{b}$$

$$E^{\text{EM}} = \frac{1}{2\mu_0} \left(\frac{|\mathbf{E}|^2}{c^2} + |\mathbf{B}|^2\right) = \frac{1}{2} \left(\frac{|\mathbf{e}|^2}{c^2} + |\mathbf{b}|^2\right)$$

Here, E^{EM} is the electromagnetic energy density and the electromagnetic fields are defined as:

$$\mathbf{b} = \mathbf{b}^{\text{plasma}} + \mathbf{b}^{\text{external}} = \mu_0^{-1/2} \left(\mathbf{B}^{\text{plasma}} + \mathbf{B}^{\text{external}}\right) = \mu_0^{-1/2} \mathbf{B}$$

$$\mathbf{e} = \mathbf{e}^{\text{plasma}} + \mathbf{e}^{\text{external}} = \mu_0^{-1/2} \left(\mathbf{E}^{\text{plasma}} + \mathbf{E}^{\text{external}}\right) = \mu_0^{-1/2} \mathbf{E}$$

Here, $\mathbf{b}^{\text{plasma}}$ is the magnetic field induced in the plasma, $\mathbf{e}^{\text{plasma}}$ is the electric field associated with net charge in the plasma, while $\mathbf{e}^{\text{external}}$ and $\mathbf{b}^{\text{external}}$ are electromagnetic fields computed "externally" to Maxwell's

equations inside the plasma. With these identitifications, the fluid part of the gasDynamicMaxwellDednerEqn takes the form:

$$\begin{aligned}
\frac{\partial \rho}{\partial t} + \nabla \cdot [\rho\, \mathbf{u}] &= 0 \\
\frac{\partial \left(\rho \mathbf{u} + c^{-2}\mathbf{S}^{\mathrm{EM}}\right)}{\partial t} + \nabla \cdot \left[\rho\, \mathbf{u}\, \mathbf{u}^T + \mathbb{I} P - \mathcal{T}^{\mathrm{EM}}\right] &= 0 \\
\frac{\partial \left(E + E^{\mathrm{EM}}\right)}{\partial t} + \nabla \cdot \left[(E+P)\, \mathbf{u} + \mathbf{S}^{\mathrm{EM}}\right] &= 0
\end{aligned}$$

The electromagnetic part of the system is solved in USim as:

$$\begin{aligned}
\frac{\partial \mathbf{b}^{\mathrm{plasma}}}{\partial t} - \nabla \times \mathbf{e} + \nabla \psi &= 0 \\
\frac{\partial \mathbf{e}^{\mathrm{plasma}}}{\partial t} + f^2 c_{\mathrm{fast}}^2 \nabla \times \mathbf{b} + \nabla \phi &= -f^2 c_{\mathrm{fast}}^2 \mu_0^{1/2} \sum_{\mathrm{species}} \mathbf{J}^{\mathrm{species}} \\
\frac{\partial \psi}{\partial t} + \nabla \cdot \left[c_{\mathrm{fast}}^2 \mathbf{b}^{\mathrm{plasma}}\right] &= 0 \\
\frac{\partial \phi}{\partial t} + \nabla \cdot \left[c_{\mathrm{fast}}^2 \mathbf{e}^{\mathrm{plasma}}\right] &= \mu_0^{-1/2} \sum_{\mathrm{species}} q^{\mathrm{species}}
\end{aligned}$$

Here, we have written $c^2 = f^2 c_{\mathrm{fast}}^2 = (\epsilon_0 \mu_0)^{-1}$, where f is a dimensionless number that defines the ratio of the speed of light to the fatest wave in the mesh and we have further defined $\psi = \mu_0^{-1/2}\Psi$ and $\Phi = \mu_0^{-1/2}\Phi$.

In order to close the electromagnetic part of the equations, a model for the current density and charge is required. An example of such a model that is provided with USim is *mhdSrc*. However, the user is also free to construct their own closure that returns:

$$\mu_0^{-1/2} \sum_{\mathrm{species}} q^{\mathrm{species}}; \;\; \mu_0^{1/2} \sum_{\mathrm{species}} \mathbf{J}^{\mathrm{species}}$$

11.13.1 Parameters

`lightSpeed` (float, optional) The speed of light in m/s. Used to specify the speed of light in the fluid momentum and energy equations. Defaults to 2.99792458e8.

`lightSpeedFactor` (float, optional) Dimensionless number, used to specify the ratio of the speed of light to the fastest wave speed in the grid. Defaults to 1.0e3.

`basementPressure` (float, optional) The minimum pressure allowed. Pressures below this value will be replaced with this value for primitive state, eigensystem and flux computations. Defaults to zero.

`basementDensity` (float, optional) The minimum density allowed. Densities below this value will be replaced with this value for primitive state, eigensystem and flux computations. Defaults to zero.

`gasGamma` (float, optional) Specifies the adiabatic index (ratio of specific heats), γ. Defaults to 5/3.

`externalEfield` (string, optional) Specifies the name of the data structure containing the externally computed electric field, $\mathbf{e}^{\mathrm{external}}$.

`externalBfield` (string, optional) Specifies the name of the data structure containing the externally computed magnetic field, $\mathbf{b}^{\mathrm{external}}$.

11.13.2 Parent Updater Data

`in` (string vector, required)

`Vector of Conserved Quantities` (***nodalArray*, 12-components, required**) The vector of conserved quantities, $\mathbf{q}$ has 9 entries:

0. ρ: mass density
1. $\rho\, u_{\hat{\mathbf{i}}} + c^{-2} S^{\mathrm{EM}}_{\hat{\mathbf{i}}} = \left(\rho\mathbf{u} + c^{-2}\mathbf{S}^{\mathrm{EM}}\right)\cdot\hat{\mathbf{i}}$: total momentum density in the $\hat{\mathbf{i}}$ direction
2. $\rho\, u_{\hat{\mathbf{j}}} + c^{-2} S^{\mathrm{EM}}_{\hat{\mathbf{j}}} = \left(\rho\mathbf{u} + c^{-2}\mathbf{S}^{\mathrm{EM}}\right)\cdot\hat{\mathbf{j}}$: total momentum density in the $\hat{\mathbf{j}}$ direction
3. $\rho\, u_{\hat{\mathbf{k}}} + c^{-2} S^{\mathrm{EM}}_{\hat{\mathbf{k}}} = \left(\rho\mathbf{u} + c^{-2}\mathbf{S}^{\mathrm{EM}}\right)\cdot\hat{\mathbf{k}}$: total momentum density in the $\hat{\mathbf{k}}$ direction
4. $E + E^{\mathrm{EM}} = \frac{P}{\gamma-1} + \frac{1}{2}\rho|\mathbf{u}|^2 + E^{\mathrm{EM}}$: total energy density
5. $b_{\hat{\mathbf{i}}} = \mathbf{b}\cdot\hat{\mathbf{i}} = \mu_0^{-1/2}\mathbf{B}\cdot\hat{\mathbf{i}}$: magnetic field normalized by permeability of free-space in the $\hat{\mathbf{i}}$ direction
6. $b_{\hat{\mathbf{j}}} = \mathbf{b}\cdot\hat{\mathbf{j}} = \mu_0^{-1/2}\mathbf{B}\cdot\hat{\mathbf{j}}$: magnetic field normalized by permeability of free-space in the $\hat{\mathbf{j}}$ direction
7. $b_{\hat{\mathbf{k}}} = \mathbf{b}\cdot\hat{\mathbf{k}} = \mu_0^{-1/2}\mathbf{B}\cdot\hat{\mathbf{k}}$: magnetic field normalized by permeability of free-space in the $\hat{\mathbf{k}}$ direction
8. $e_{\hat{\mathbf{i}}} = \mathbf{e}\cdot\hat{\mathbf{i}} = \mu_0^{-1/2}\mathbf{E}\cdot\hat{\mathbf{i}}$: electric field normalized by permeability of free-space in the $\hat{\mathbf{i}}$ direction
9. $e_{\hat{\mathbf{j}}} = \mathbf{e}\cdot\hat{\mathbf{j}} = \mu_0^{-1/2}\mathbf{E}\cdot\hat{\mathbf{j}}$: electric field normalized by permeability of free-space in the $\hat{\mathbf{j}}$ direction
10. $e_{\hat{\mathbf{k}}} = \mathbf{e}\cdot\hat{\mathbf{k}} = \mu_0^{-1/2}\mathbf{E}\cdot\hat{\mathbf{k}}$: electric field normalized by permeability of free-space in the $\hat{\mathbf{k}}$ direction
11. ψ: magnetic field correction potential
12. ϕ: electric field correction potential

`Fastest Wave Speed` (***dynVector*, 1-component, required**) The fastest wave speed across the entire simulation domain, c_{fast}. Can be computed using *hyperbolic (1d, 2d, 3d)* (see below).

`Externally Computed Electric Field` (***nodalArray*, 3-components, optional**) Additional contribution to the electric field, $\mathbf{e}^{\mathrm{external}}$, which is not evolved by Ampere's equation, but does contribution to the induction equation, the Lorentz force and the work done on the plasma. The data structure containing $\mathbf{e}^{\mathrm{external}}$ is specified by the "externalEField" option described below.

0. $e^{\mathrm{external}}_{\hat{\mathbf{i}}} = \mathbf{e}^{\mathrm{external}}\cdot\hat{\mathbf{i}} = \mu_0^{-1/2}\mathbf{E}^{\mathrm{external}}\cdot\hat{\mathbf{i}}$: "externally" computed electric field normalized by permeability of free-space in the $\hat{\mathbf{i}}$ direction.
1. $e^{\mathrm{external}}_{\hat{\mathbf{j}}} = \mathbf{e}^{\mathrm{external}}\cdot\hat{\mathbf{j}} = \mu_0^{-1/2}\mathbf{E}^{\mathrm{external}}\cdot\hat{\mathbf{j}}$:"externally" computed electric field normalized by permeability of free-space in the $\hat{\mathbf{j}}$ direction
2. $e^{\mathrm{external}}_{\hat{\mathbf{k}}} = \mathbf{e}^{\mathrm{external}}\cdot\hat{\mathbf{k}} = \mu_0^{-1/2}\mathbf{E}^{\mathrm{external}}\cdot\hat{\mathbf{k}}$: "externally" computed electric field normalized by permeability of free-space in the $\hat{\mathbf{k}}$ direction

`Externally Computed Magnetic Field` (***nodalArray*, 3-components, optional**) Additional contribution to the magnetic field, $\mathbf{b}^{\mathrm{external}}$, which is not evolved by the induction equation, but does contribute to Ampere's equation, the Lorentz force and the work done on the plasma. The data structure containing $\mathbf{b}^{\mathrm{external}}$ is specified by the "externalBField" option described below.

0. $b^{\mathrm{external}}_{\hat{\mathbf{i}}} = \mathbf{b}^{\mathrm{external}}\cdot\hat{\mathbf{i}} = \mu_0^{-1/2}\mathbf{B}^{\mathrm{external}}\cdot\hat{\mathbf{i}}$: magnetic field normalized by permeability of free-space in the $\hat{\mathbf{i}}$ direction

1. $b_{\hat{\mathbf{j}}}^{\text{external}} = \mathbf{b}^{\text{external}} \cdot \hat{\mathbf{j}} = \mu_0^{-1/2} \mathbf{B}^{\text{external}} \cdot \hat{\mathbf{j}}$: magnetic field normalized by permeability of free-space in the $\hat{\mathbf{j}}$ direction
2. $b_{\hat{\mathbf{k}}}^{\text{external}} = \mathbf{b}^{\text{external}} \cdot \hat{\mathbf{k}} = \mu_0^{-1/2} \mathbf{B}^{\text{external}} \cdot \hat{\mathbf{k}}$: magnetic field normalized by permeability of free-space in the $\hat{\mathbf{k}}$ direction

`out` (string vector, required) For the gasDynamicMhdDednerEqn, one of four output variables are computed, depending on whether the equation is combined with an updater capable of computing fluxes (*classicMusclUpdater (1d, 2d, 3d)*), primitive variables (*computePrimitiveState(1d, 2d, 3d)*), the time step associated with the CFL condition (*timeStepRestrictionUpdater (1d, 2d, 3d)*) or the fastest wave speed in the grid (*hyperbolic (1d, 2d, 3d)*).

`Vector of Fluxes` (*nodalArray*, 12-components) When combined with an updater that computes $\nabla \cdot \mathcal{F}(\mathbf{w})$ (e.g. *classicMusclUpdater (1d, 2d, 3d)*), the equation system returns:

0. $\nabla \cdot \mathcal{F}(\rho)$: mass flux
1. $\nabla \cdot \mathcal{F}\left(\rho\, u_{\hat{\mathbf{i}}} + c^{-2} S_{\hat{\mathbf{i}}}^{\text{EM}}\right)$: $\hat{\mathbf{i}}$ momentum flux
2. $\nabla \cdot \mathcal{F}\left(\rho\, u_{\hat{\mathbf{j}}} + c^{-2} S_{\hat{\mathbf{j}}}^{\text{EM}}\right)$: $\hat{\mathbf{j}}$ momentum flux
3. $\nabla \cdot \mathcal{F}\left(\rho\, u_{\hat{\mathbf{k}}} + c^{-2} S_{\hat{\mathbf{k}}}^{\text{EM}}\right)$: $\hat{\mathbf{k}}$ momentum flux
4. $\nabla \cdot \mathcal{F}(E)$: total energy flux
5. $\nabla \cdot \mathcal{F}\left(b_{\hat{\mathbf{i}}}\right)$: $\hat{\mathbf{i}}$ magnetic field flux
6. $\nabla \cdot \mathcal{F}\left(b_{\hat{\mathbf{j}}}\right)$: $\hat{\mathbf{j}}$ magnetic field flux
7. $\nabla \cdot \mathcal{F}\left(b_{\hat{\mathbf{k}}}\right)$: $\hat{\mathbf{k}}$ magnetic field flux
8. $\nabla \cdot \mathcal{F}\left(e_{\hat{\mathbf{i}}}\right)$: $\hat{\mathbf{i}}$ electric field flux
9. $\nabla \cdot \mathcal{F}\left(e_{\hat{\mathbf{j}}}\right)$: $\hat{\mathbf{j}}$ electric field flux
10. $\nabla \cdot \mathcal{F}\left(e_{\hat{\mathbf{k}}}\right)$: $\hat{\mathbf{k}}$ electric field flux
11. $\nabla \cdot \mathcal{F}(\psi)$: magnetic correction potential flux
12. $\nabla \cdot \mathcal{F}(\phi)$: electric correction potential flux

`Vector of Primitive States` (*nodalArray*, 9-components) When combined with an updater that computes $\mathbf{w} = \mathbf{w}(\mathbf{q})$ (e.g. *computePrimitiveState(1d, 2d, 3d)*), the equation systen returns:

0. ρ: mass density
1. $u_{\hat{\mathbf{i}}} = \mathbf{u} \cdot \hat{\mathbf{i}}$: velocity in the $\hat{\mathbf{i}}$ direction
2. $u_{\hat{\mathbf{j}}} = \mathbf{u} \cdot \hat{\mathbf{j}}$: velocity in the $\hat{\mathbf{j}}$ direction
3. $u_{\hat{\mathbf{k}}} = \mathbf{u} \cdot \hat{\mathbf{k}}$: velocity in the $\hat{\mathbf{k}}$ direction
4. $P = \rho\epsilon(\gamma - 1)$: ideal gas pressure
5. $b_{\hat{\mathbf{i}}} = \mathbf{b} \cdot \hat{\mathbf{i}} = \mu_0^{-1/2} \mathbf{B} \cdot \hat{\mathbf{i}}$: magnetic field normalized by permeability of free-space in the $\hat{\mathbf{i}}$ direction
6. $b_{\hat{\mathbf{j}}} = \mathbf{b} \cdot \hat{\mathbf{j}} = \mu_0^{-1/2} \mathbf{B} \cdot \hat{\mathbf{j}}$: magnetic field normalized by permeability of free-space in the $\hat{\mathbf{j}}$ direction
7. $b_{\hat{\mathbf{k}}} = \mathbf{b} \cdot \hat{\mathbf{k}} = \mu_0^{-1/2} \mathbf{B} \cdot \hat{\mathbf{k}}$: magnetic field normalized by permeability of free-space in the $\hat{\mathbf{k}}$ direction

8. $e_{\hat{\mathbf{i}}} = \mathbf{e} \cdot \hat{\mathbf{i}} = \mu_0^{-1/2} \mathbf{E} \cdot \hat{\mathbf{i}}$: electric field normalized by permeability of free-space in the $\hat{\mathbf{i}}$ direction
9. $e_{\hat{\mathbf{j}}} = \mathbf{e} \cdot \hat{\mathbf{j}} = \mu_0^{-1/2} \mathbf{E} \cdot \hat{\mathbf{j}}$: electric field normalized by permeability of free-space in the $\hat{\mathbf{j}}$ direction
10. $e_{\hat{\mathbf{k}}} = \mathbf{e} \cdot \hat{\mathbf{k}} = \mu_0^{-1/2} \mathbf{E} \cdot \hat{\mathbf{k}}$: electric field normalized by permeability of free-space in the $\hat{\mathbf{k}}$ direction
11. ψ: magnetic field correction potential
12. ϕ: electric field correction potential

`Time Step` (*dynVector*, 1-component) When combined with *timeStepRestrictionUpdater (1d, 2d, 3d)*, the equation system returns the time step consisten with the CFL condition across the entire simulation domain.

`Fastest Wave Speed` (*dynVector*, 1-component) When combined with *hyperbolic (1d, 2d, 3d)*, the equation system returns the fastest wave speed across the entire simulation domain, c_{fast}.

11.13.3 Examples

The following block demonstrates the gasDynamicMaxwellDednerEqn used in combination with *classicMusclUpdater (1d, 2d, 3d)* to compute $\nabla \cdot \mathcal{F}(\mathbf{w})$ with an externally supplied magnetic field:

```
<Updater hyper>
  kind=classicMuscl1d
  onGrid=domain

  # input nodal component arrays
  in=[q   backgroundB]

  # output nodal component array
  out=[qnew]

  # input dynVector containing fastest wave speed
  waveSpeeds=[waveSpeed]

  # the numerical flux to use
  numericalFlux= hlldFlux

  # CFL number to use
  cfl=0.3
  # Form of variables to limit
  variableForm= primitive

  # Limiter; one per input nodal component array
  limiter=[minmod   minmod]

  # list of equations to solve
  equations=[mhd]

  <Equation mhd>
    kind = gasDynamicMaxwellDednerEqn
    gasGamma = GAS_GAMMA
    lightSpeedFactor  =LIGHT_SPEED_FACTOR
    externalBfield = EXTERNAL_FIELD
    basementPressure = BASEMENT_PRESSURE
    basementDensity = BASEMENT_DENSITY
  </Equation>

  <Source mhdSrc>
```

```
    kind = gasDynamicMhdDednerSrc
    scalarConductivity = $1.0/OHMIC_RESISTIVITY$
  </Source>

  <Source mhdClean>
    kind = mhdSrc
    model = mhdDednerEqn
    momentumEnergySource = 1
    inputVariables = [q,divB,gradPsi]
  </Source>

</Updater>
```

The following block demonstrates the gasDynamicMaxwellDednerEqn used in combination with *computePrimitiveState(1d, 2d, 3d)* to compute $\mathbf{w}$:

```
<Updater computePrimitiveState>
  kind = computePrimitiveState$NDIM$d

  onGrid = domain
  # input array
  in = [q]

  # ouput data-structures
  out = [w]

  <Equation fluid>
    kind = gasDynamicMaxwellDednerEqn
    gasGamma = GAS_GAMMA
    lightSpeedFactor  =LIGHT_SPEED_FACTOR
    externalBfield = EXTERNAL_FIELD
    basementPressure = BASEMENT_PRESSURE
    basementDensity = BASEMENT_DENSITY
  </Equation>

</Updater>
```

The following block demonstrates the gasDynamicMhdDednerEqn used in combination with *timeStepRestrictionUpdater (1d, 2d, 3d)* and *hyperbolic (1d, 2d, 3d)* to compute c_{fast} with an externally supplied magnetic field:

```
<Updater getWaveSpeed>
  kind=timeStepRestrictionUpdater1d
  onGrid=domain

  # input nodal component arrays
  in=[q   backgroundB]

  # output dynVector containing fastest wave speed
  waveSpeeds=[waveSpeed]

  # list of equations to compute fastest wave speed for
  restrictions=[idealMhd]

  # courant condition to apply to the timestep
  courantCondition=1.0

 <TimeStepRestriction idealMhd>
   kind = hyperbolic1d
```

```
    model = gasDynamicMaxwellDednerEqn
     gasGamma = GAS_GAMMA
     lightSpeedFactor  =LIGHT_SPEED_FACTOR
     externalBfield = EXTERNAL_FIELD
     basementPressure = BASEMENT_PRESSURE
     basementDensity = BASEMENT_DENSITY
  </TimeStepRestriction>
 </Updater>
```

11.14 twoFluidEqn

Two fluid equations written as total mass density, momentum density, total charge density total current density and ion and electron energy. The two-fluid equations can also be written as two separate sets of euler equations, however, this form has the advantage that numerical diffusion is applied to the total charge density so that quasi-neutrality is enforced numerically.

$$\frac{\partial}{\partial t}\begin{pmatrix} \rho \\ \rho\, u_x \\ \rho\, u_y \\ \rho\, u_z \\ \rho_c \\ j_x \\ j_y \\ j_z \\ e_i \\ e_e \end{pmatrix} + \nabla \cdot P = 0$$

where P is defined as

$$\begin{pmatrix} \rho_i\, u_{x\,i} + \rho_i\, u_{x\,e} & \rho_i\, u_{y\,i} + \rho_e\, u_{y\,e} & \rho_i\, u_{z\,i} + \rho_e\, u_{z\,e} \\ \rho_i\, u_{x\,i}^2 + P_i + \rho_e\, u_{x\,e}^2 + P_e & \rho_i\, u_{x\,i}\, u_{y\,i} + \rho_e\, u_{x\,e}\, u_{y\,e} & \rho_i\, u_{x\,i}\, u_{z\,i} + \rho_e\, u_{x\,e}\, u_{z\,e} \\ \rho_i\, u_{y\,i}\, u_{x\,i} + \rho_e\, u_{y\,e}\, u_{x\,e} & \rho_i\, u_{y\,i}\, u_{y\,i} + P_i + \rho_e\, u_{y\,e}\, u_{y\,e} + P_e & \rho_i\, u_{y\,i}\, u_{z\,i} + \rho_e\, u_{y\,e}\, u_{z\,e} \\ \rho_i\, u_{z\,i}\, u_{x\,i} + \rho_e\, u_{z\,e}\, u_{x\,e} & \rho_i\, u_{z\,i}\, u_{y\,i} + \rho_e\, u_{z\,e}\, u_{y\,e} & \rho_i\, u_{z\,i}\, u_{z\,i} + P_i + \rho_e\, u_{z\,e}\, u_{z\,e} + P_e \\ \rho_i\, u_{x\,i} + \rho_i\, u_{x\,e} & \rho_i\, u_{y\,i} + \rho_e\, u_{y\,e} & \rho_i\, u_{z\,i} + \rho_e\, u_{z\,e} \\ r_i(\rho_i\, u_{x\,i}^2 + P_i) + r_e(\rho_e\, u_{x\,e}^2 + P_e) & r_i\rho_i\, u_{x\,i}\, u_{y\,i} + r_e\rho_e\, u_{x\,e}\, u_{y\,e} & r_i\rho_i\, u_{x\,i}\, u_{z\,i} + r_e\rho_e\, u_{x\,e}\, u_{z\,e} \\ r_i\rho_i\, u_{y\,i}\, u_{x\,i} + r_e\rho_e\, u_{y\,e}\, u_{x\,e} & r_i(\rho_i\, u_{y\,i}\, u_{y\,i} + P_i) + r_e(\rho_e\, u_{y\,e}\, u_{y\,e} + P_e) & r_i\rho_i\, u_{y\,i}\, u_{z\,i} + r_e\rho_e\, u_{y\,e}\, u_{z\,e} \\ r_i\rho_i\, u_{z\,i}\, u_{x\,i} + r_e\rho_e\, u_{z\,e}\, u_{x\,e} & r_i\rho_i\, u_{z\,i}\, u_{y\,i} + r_e\rho_e\, u_{z\,e}\, u_{y\,e} & r_i(\rho_i\, u_{z\,i}\, u_{z\,i} + P_i) + r_e(\rho_e\, u_{z\,e}\, u_{z\,e} + P_e \\ u_{x\,i}\,(e_i + P_i) & u_{y\,i}\,(e_i + P_i) & u_{z\,i}\,(e_i + P_i) \\ u_{x\,e}\,(e_e + P_e) & u_{y\,e}\,(e_e + P_e) & u_{z\,e}\,(e_e + P_e) \end{pmatrix}$$

With $r_i = q_i/m_i$ and $r_e = q_e/m_e$ where q_e is the electron charge, q_i is the ion charge, m_e is the electron mass and m_i is the ion mass. In addition the variables $(\rho_\alpha, u_{x\,\alpha}, u_{y\,\alpha}, u_{x\,\alpha}, e_\alpha, P_\alpha)$ are the species mass density, species x velocity, species y velocity, species z velocity, species total energy density, and species pressure respectively. In this case α represents the species, either e for electron or i for ion.

11.14.1 Parameters

`ionGamma` (float) Specific heat ratio for the ions

`electronGamma` (float) Specific heat ratio for the electrons. Defaults to 5/3

`ionMass` (float) ion mass

`electronMass` (float) electron mass

ionCharge (float) ion charge

electronCharge electron charge

basementPressure (float) The minimum pressure allowed. Defaults to 0.

basementDensity (float) The minimum density allowed for the ions. Defaults to 0. The electron basement density is determined by multiplying by the mass ratio, therefore $basementDensityElectrons = (me/mi)basementDensity$

11.14.2 Parent Updater Data

in (string vector, required)

1st Input Variable

0. ρ mass density
1. $\rho\, u_x$ x momentum density
2. $\rho\, u_y$ y momentum density
3. $\rho\, u_z$ z momentum density
4. ρ_c total charge density
5. j_x x current density
6. j_y y current density
7. j_z z current density
8. e_i ion energy density
9. e_e electron energy density

11.14.3 Example

An example *twoFluidEqn* equation block is given below:

```
<Equation twoFluid>
  kind = twoFluidEqn
  ionGamma = GAS_GAMMA
  electronGamma = GAS_GAMMA
  ionMass = ION_MASS
  electronMass = ELECTRON_MASS
  ionCharge = ION_CHARGE
  electronCharge = ELECTRON_CHARGE
  basementDensity = BASEMENT_DENSITY
  basementPressure = BASEMENT_PRESSURE
</Equation>
```

The following equation can be used to implement a hyperbolic equation system at the input file level:

11.15 userDefinedEqn

Define an arbitrary hyperbolic system. Built in hyperbolic equations should be used when they are available as they are faster.

11.15.1 Parameters

indVars_inName For each input variable an "indVars" array must be defined. So if in = [E, B] then indVars_E and indVars_B must be defined. If indVars_E = ["Ex","Ey","Ez"] then operations are performed on "Ex","Ey" and "Ez" in the expression evaluator.

transform_inName For each variable there must be a vector that tells how the data is transformed upon rotation. For example, for an electric field E, the transform would be transform_E = [vector] so that USim knows the input data is a vector. If the input data is density, momentum, energy as in the euler equations then we would have transform_q = [scalar, vector, scalar] which assumes that momentum has 3 components. The previous example transforms the first variable as if it were a scalar, then the next 3 variables as if they were part of a tensor and then the last variable as if it were a scalar. Available options are *scalar*, *vector* and *tensor*. It is assumed that *vector* has 3 components even in 1D and 2D simulations. Also it's assumed that *tensor* has 6 components in the order Txx, Txy, Tx, Tyy, Tyz, Tzz and that the remaining components are symmetric so are redundant.

preExprs (string vector) Strings must be put in quotes. The preExprs is used to compute quantities based on indVars that can later be used in the *exprs* to evaluate the output. Available commands are defined by the muParser (http://muparser.sourceforge.net)

flux (string vector) Strings must be put in quotes. The strings are used to evaluate the flux in the x-direction. The fluxes in other directions are obtained through rotation of the input vector. Available command are defined by the muParser (http://muparser.sourceforge.net/)

eigenvalues (string vector) Strings must be put in quotes. The strings are evaluated and placed in the output array and are used to define the set of eigenvalues for the system. The eigenvalues are technically the eigenvalues in the x-direction and values in other directions are obtained through rotation. Available command are defined by the muParser (http://muparser.sourceforge.net/)

other (variable definition) In addition, an arbitrary number of constants can be defined that can then be used in evaluating expression in both *preExprs* and *flux* and *eigenvalues*.

11.15.2 Parent Updater Data

in (string vector) Input 1 to N are input nodalArray on which operations will be performed. Example in = [E, B]

out (string vector) output nodalArray where the result of the operation is stored

11.15.3 Example

```
<Updater hyper>
  kind = classicMuscl1d
  onGrid = domain

  in = [q]
  out = [qnew]
  timeIntegrationScheme = none
  numericalFlux = $RIEMANN_SOLVER$
  cfl = CFL
  variableForm = $VARIABLE_FORM$
  limiter = [$LIMITER$]

  equations = [euler]

  <Equation euler>
```

```
        kind = userDefinedEqn

        indVars_q = ["rho","mx","my","mz","en"]
        transform_q = [scalar, vector, scalar]

        gamma = GAS_GAMMA
        preExprs = ["p=(gamma-1.0)*(en-0.5*((mx*mx+my*my+mz*mz)/rho))"]
        flux = ["mx","(mx*mx/rho)+p","(mx*my/rho)","(mx*mz/rho)","(mx/rho)*(en+p)"]
        eigenvalues = ["sqrt(p*gamma/rho)+(mx/rho)"]

      </Equation>

    </Updater>
```

CHAPTER

TWELVE

ALGEBRAIC EQUATIONS

A **Source** or **Equation** block in USim that defines a **local** non-linear algebraic transformation of a set of input *nodalArray*s, $\mathbf{q}_{\text{Input}}$ into a single output *nodalArray* through:

$$\mathbf{q}_{\text{Output}} = \mathcal{S}\left(\mathbf{q}_{\text{Input}}, x, y, z, t\right)$$

where $\mathcal{S}\left(\mathbf{q}_{\text{Input}}, x, y, z, t\right)$ represents a non-linear algebraic transformation that is applied locally, i.e. $\mathcal{S}\left(\mathbf{q}_{\text{Input}}, x, y, z, t\right)$ **only** depends on the data in an individual element in the *Grid* and **not** on elements adjacent to that element.

Source and *Equation* blocks can be used for a range of purposes in USim. One particular example is the addition of terms to the right-hand side of a hyperbolic equation system

$$\frac{\partial \mathbf{q}}{\partial t} + \nabla \cdot \left[\mathcal{F}\left(\mathbf{w}\right)\right] = \mathcal{S}\left(\mathbf{w}, x, y, z, t\right)$$

A specific example of a source block that can be used in this way is the *exprHyperSrc* to apply a gravitational acceleration to a neutral fluid:

```
<Source gravity>
  kind = exprHyperSrc
  gravity = GRAVITY
  indVars = ["rho", "rhou", "rhov", "rhow", "Er"]
  exprs = ["0.0", "0.0", "-rho*gravity", "0.0", "-gravity*rhov"]
</Source>
```

Note: Any *kind* listed below can be used as an *Equation* block in the *localOdeIntegrator (1d, 2d, 3d)* Updater or the *equation (1d, 2d, 3d)* Updater.

Note: The *firstOrderMusclUpdater (1d, 2d, 3d)*, *classicMusclUpdater (1d, 2d, 3d)*, *unstructMusclUpdater (1d, 2d, 3d)*, and *thirdOrderMusclUpdater (1d, 2d, 3d)* Updaters are the only updaters that use the following kinds as *Source* blocks.

Note: None of the *kind*s listed below can be used as *Equation* blocks in the muscl updaters. The *Equation* blocks in the muscle updaters are *Hyperbolic Equations*

The following parameters are common to all *Source* blocks:

`kind` (string) All *Source* and *Equation* blocks take a string *kind* that specifies the type of source.

The following kinds can be combined with *Hyperbolic Equations* to enable the use of curvilinear coordinates for hyperbolic problems:

12.1 eulerSym

The *eulerSym* source computes additional (source) terms associated with curvilinear (cylindrical and spherical) coordinate systems for inviscid compressible hydrodynamics. The *eulerSym* source is combined with (e.g.) *classicMusclUpdater (1d, 2d, 3d)* and one of *eulerEqn*, eulerTwoTemp, eulerThreeTemp or *realGasEosEqn*. Note that formulation of the cylindrical source term assumes axisymmetry and so can be used in 1- and 2-dimensions, while the formulation of spherical coordinates assumes spherical symmetry and so can only be used in one-dimension.

12.1.1 Parameters

`symmetryType` (string, required) The curvilinear cordinate system to be used. Available options are *cylindrical* or *spherical*

`model` (string, required) Determines the equation that the source term will be computed for. Available options are:

`eulerEqn` Compute curvilinear source terms associated with *eulerEqn*.

`realGasEosEqn` Compute curvilinear source terms associated with *realGasEosEqn*.

`eulerTwoTemp` Compute curvilinear source terms associated with eulerTwoTemp.

`eulerThreeTemp` Compute curvilinear source terms associated with eulerThreeTemp.

`gasGamma` (float, required if *model=eulerEqn*) Specifies the adiabatic index (ratio of specific heats), γ.

12.1.2 Parent Updater Data

`in` (string vector, required) The number and form of the input variables for *eulerSym* depends on the choice of the parameter *model*.

`model = eulerEqn` (1 input variable)

`Vector of Conserved Quantities` (*nodalArray*, 5-components, required) The vector of conserved quantities, $\mathbf{q}$ has 5 entries:

0. ρ: mass density
1. $\rho\, u_{\hat{\mathbf{i}}} = \rho\mathbf{u}\cdot\hat{\mathbf{i}}$: momentum density in the $\hat{\mathbf{i}}$ direction
2. $\rho\, u_{\hat{\mathbf{j}}} = \rho\mathbf{u}\cdot\hat{\mathbf{j}}$: momentum density in the $\hat{\mathbf{j}}$ direction
3. $\rho\, u_{\hat{\mathbf{k}}} = \rho\mathbf{u}\cdot\hat{\mathbf{k}}$: momentum density in the $\hat{\mathbf{k}}$ direction
4. $E = \frac{P}{\gamma-1} + \frac{1}{2}\rho|\mathbf{u}|^2$: total energy density

`model = realGasEosEqn` (2 input variables)

`Vector of Conserved Quantities` (*nodalArray*, 5-components, required) The vector of conserved quantities, $\mathbf{q}$ has 5 entries:

0. ρ: mass density
1. $\rho\, u_{\hat{\mathbf{i}}} = \rho\mathbf{u}\cdot\hat{\mathbf{i}}$: momentum density in the $\hat{\mathbf{i}}$ direction
2. $\rho\, u_{\hat{\mathbf{j}}} = \rho\mathbf{u}\cdot\hat{\mathbf{j}}$: momentum density in the $\hat{\mathbf{j}}$ direction
3. $\rho\, u_{\hat{\mathbf{k}}} = \rho\mathbf{u}\cdot\hat{\mathbf{k}}$: momentum density in the $\hat{\mathbf{k}}$ direction
4. $E = \frac{P}{\gamma-1} + \frac{1}{2}\rho|\mathbf{u}|^2$: total energy density

`Gas Pressure` (*nodalArray*, **1-component, required**) The gas pressure, P.

`model = eulerTwoTemp` **(3 input variables)**

`Vector of Conserved Quantities` (*nodalArray*, **6-components, required**) The vector of conserved quantities, $\mathbf{q}$ has 6 entries:

0. ρ: mass density
1. $\rho\, u_{\hat{\mathbf{i}}} = \rho \mathbf{u} \cdot \hat{\mathbf{i}}$: momentum density in the $\hat{\mathbf{i}}$ direction
2. $\rho\, u_{\hat{\mathbf{j}}} = \rho \mathbf{u} \cdot \hat{\mathbf{j}}$: momentum density in the $\hat{\mathbf{j}}$ direction
3. $\rho\, u_{\hat{\mathbf{k}}} = \rho \mathbf{u} \cdot \hat{\mathbf{k}}$: momentum density in the $\hat{\mathbf{k}}$ direction
4. $E = \frac{P_{\text{heavy}}}{\gamma_{\text{heavy}}-1} + \frac{P_{\text{electron}}}{\gamma_{\text{electron}}-1} + \frac{1}{2}\rho|\mathbf{u}|^2$: total energy density
5. $E_{\text{electron}} = \frac{P_{\text{electron}}}{\gamma_{\text{electron}}-1}$: electron internal energy density

`Total Gas Pressure` (*nodalArray*, **1-component, required**) The total gas pressure, $P_{\text{heavy}} + P_{\text{electron}}$.

`Electron Pressure` (*nodalArray*, **1-component, required**) The electron pressure, P_{electron}.

`model = eulerThreeTemp` **(3 input variables)**

`Vector of Conserved Quantities` (*nodalArray*, **7-components, required**) The vector of conserved quantities, $\mathbf{q}$ has 7 entries:

0. ρ: mass density
1. $\rho\, u_{\hat{\mathbf{i}}} = \rho \mathbf{u} \cdot \hat{\mathbf{i}}$: momentum density in the $\hat{\mathbf{i}}$ direction
2. $\rho\, u_{\hat{\mathbf{j}}} = \rho \mathbf{u} \cdot \hat{\mathbf{j}}$: momentum density in the $\hat{\mathbf{j}}$ direction
3. $\rho\, u_{\hat{\mathbf{k}}} = \rho \mathbf{u} \cdot \hat{\mathbf{k}}$: momentum density in the $\hat{\mathbf{k}}$ direction
4. $E = \frac{P_{\text{heavy}}}{\gamma_{\text{heavy}}-1} + \frac{P_{\text{electron}}}{\gamma_{\text{electron}}-1} + \frac{1}{2}\rho|\mathbf{u}|^2$: total energy density
5. $E_{\text{electron}} = \frac{P_{\text{electron}}}{\gamma_{\text{electron}}-1}$: electron internal energy density
6. E_{vibr}: vibrational energy density

`Total Gas Pressure` (*nodalArray*, **1-component, required**) The total gas pressure, $P_{\text{heavy}} + P_{\text{electron}}$.

`Electron Pressure` (*nodalArray*, **1-component, required**) The electron pressure, P_{electron}.

`out` **(string vector, required)** The form of the output variables for *eulerSym* depends on the choice of the parameter *model* and *symmetryType*:

`model = eulerEqn`

`Vector of Source terms` (*nodalArray*, **5-components**) When *symmetryType* = *cylindrical*, the output source vector has components:

0. $\mathcal{S}(\rho) = -r^{-1}\rho\, u_{\hat{\mathbf{i}}}$: mass source
1. $\mathcal{S}\left(\rho\, u_{\hat{\mathbf{i}}}\right) = -r^{-1}\left(\rho\, u_{\hat{\mathbf{i}}}^2 + \rho\, u_{\hat{\mathbf{j}}}^2\right)$: $\hat{\mathbf{i}}$ momentum source
2. $\mathcal{S}\left(\rho\, u_{\hat{\mathbf{j}}}\right) = -2r^{-1}\rho\, u_{\hat{\mathbf{i}}}\, u_{\hat{\mathbf{j}}}$: $\hat{\mathbf{j}}$ momentum source
3. $\mathcal{S}\left(\rho\, u_{\hat{\mathbf{k}}}\right) = -r^{-1}\rho\, u_{\hat{\mathbf{i}}}\, u_{\hat{\mathbf{k}}}$: $\hat{\mathbf{k}}$ momentum source
4. $\mathcal{S}(E) = -r^{-1}u_{\hat{\mathbf{i}}}\,(E+P)$: total energy source

When *symmetryType* = *spherical*, the output source vector has components:

0. $\mathcal{S}(\rho) = -2r^{-1}\rho\, u_{\hat{\mathbf{i}}}$: mass source
1. $\mathcal{S}\left(\rho\, u_{\hat{\mathbf{i}}}\right) = -2r^{-1}\rho\, u_{\hat{\mathbf{i}}}^2 + \rho\, u_{\hat{\mathbf{j}}}^2$: $\hat{\mathbf{i}}$ momentum source
2. $\mathcal{S}\left(\rho\, u_{\hat{\mathbf{j}}}\right) = -3r^{-1}\rho\, u_{\hat{\mathbf{i}}}\, u_{\hat{\mathbf{j}}}$: $\hat{\mathbf{j}}$ momentum source
3. $\mathcal{S}\left(\rho\, u_{\hat{\mathbf{k}}}\right) = -3r^{-1}\rho\, u_{\hat{\mathbf{i}}}\, u_{\hat{\mathbf{k}}}$: $\hat{\mathbf{k}}$ momentum source
4. $\mathcal{S}(E) = -2r^{-1}u_{\hat{\mathbf{i}}}\,(E + P)$: total energy source

`model = realGasEosEqn`

`Vector of Source terms` **(*nodalArray*, 5-components)** When *symmetryType* = *cylindrical*, the output source vector has components:

0. $\mathcal{S}(\rho) = -r^{-1}\rho\, u_{\hat{\mathbf{i}}}$: mass source
1. $\mathcal{S}\left(\rho\, u_{\hat{\mathbf{i}}}\right) = -r^{-1}\rho\, u_{\hat{\mathbf{i}}}^2 + \rho\, u_{\hat{\mathbf{j}}}^2$: $\hat{\mathbf{i}}$ momentum source
2. $\mathcal{S}\left(\rho\, u_{\hat{\mathbf{j}}}\right) = -2r^{-1}\rho\, u_{\hat{\mathbf{i}}}\, u_{\hat{\mathbf{j}}}$: $\hat{\mathbf{j}}$ momentum source
3. $\mathcal{S}\left(\rho\, u_{\hat{\mathbf{k}}}\right) = -r^{-1}\rho\, u_{\hat{\mathbf{i}}}\, u_{\hat{\mathbf{k}}}$: $\hat{\mathbf{k}}$ momentum source
4. $\mathcal{S}(E) = -r^{-1}u_{\hat{\mathbf{i}}}\,(E + P)$: total energy source

When *symmetryType* = *spherical*, the output source vector has components:

0. $\mathcal{S}(\rho) = -2r^{-1}\rho\, u_{\hat{\mathbf{i}}}$: mass source
1. $\mathcal{S}\left(\rho\, u_{\hat{\mathbf{i}}}\right) = -2r^{-1}\rho\, u_{\hat{\mathbf{i}}}^2 + \rho\, u_{\hat{\mathbf{j}}}^2$: $\hat{\mathbf{i}}$ momentum source
2. $\mathcal{S}\left(\rho\, u_{\hat{\mathbf{j}}}\right) = -3r^{-1}\rho\, u_{\hat{\mathbf{i}}}\, u_{\hat{\mathbf{j}}}$: $\hat{\mathbf{j}}$ momentum source
3. $\mathcal{S}\left(\rho\, u_{\hat{\mathbf{k}}}\right) = -3r^{-1}\rho\, u_{\hat{\mathbf{i}}}\, u_{\hat{\mathbf{k}}}$: $\hat{\mathbf{k}}$ momentum source
4. $\mathcal{S}(E) = -2r^{-1}u_{\hat{\mathbf{i}}}\,(E + P)$: total energy source

`model = eulerTwoTemp`

`Vector of Source terms` **(*nodalArray*, 6-components)** When *symmetryType* = *cylindrical*, the output source vector has components:

0. $\mathcal{S}(\rho) = -r^{-1}\rho\, u_{\hat{\mathbf{i}}}$: mass source
1. $\mathcal{S}\left(\rho\, u_{\hat{\mathbf{i}}}\right) = -r^{-1}\rho\, u_{\hat{\mathbf{i}}}^2 + \rho\, u_{\hat{\mathbf{j}}}^2$: $\hat{\mathbf{i}}$ momentum source
2. $\mathcal{S}\left(\rho\, u_{\hat{\mathbf{j}}}\right) = -2r^{-1}\rho\, u_{\hat{\mathbf{i}}}\, u_{\hat{\mathbf{j}}}$: $\hat{\mathbf{j}}$ momentum source
3. $\mathcal{S}\left(\rho\, u_{\hat{\mathbf{k}}}\right) = -r^{-1}\rho\, u_{\hat{\mathbf{i}}}\, u_{\hat{\mathbf{k}}}$: $\hat{\mathbf{k}}$ momentum source
4. $\mathcal{S}(E) = -r^{-1}u_{\hat{\mathbf{i}}}\,(E + P)$: total energy source
5. $\mathcal{S}(E_{\text{electron}}) = -r^{-1}u_{\hat{\mathbf{i}}}\,(E_{\text{electron}} + P_{\text{electron}})$: electron internal energy source

When *symmetryType* = *spherical*, the output source vector has components:

0. $\mathcal{S}(\rho) = -2r^{-1}\rho\, u_{\hat{\mathbf{i}}}$: mass source
1. $\mathcal{S}\left(\rho\, u_{\hat{\mathbf{i}}}\right) = -2r^{-1}\rho\, u_{\hat{\mathbf{i}}}^2 + \rho\, u_{\hat{\mathbf{j}}}^2$: $\hat{\mathbf{i}}$ momentum source
2. $\mathcal{S}\left(\rho\, u_{\hat{\mathbf{j}}}\right) = -3r^{-1}\rho\, u_{\hat{\mathbf{i}}}\, u_{\hat{\mathbf{j}}}$: $\hat{\mathbf{j}}$ momentum source

3. $\mathcal{S}\left(\rho\, u_{\hat{\mathbf{k}}}\right) = -3r^{-1}\rho\, u_{\hat{\mathbf{i}}}\, u_{\hat{\mathbf{k}}}$: $\hat{\mathbf{k}}$ momentum source
4. $\mathcal{S}\left(E\right) = -2r^{-1}u_{\hat{\mathbf{i}}}\ (E + P)$: total energy source
5. $\mathcal{S}\left(E_{\text{electron}}\right) = -2r^{-1}u_{\hat{\mathbf{i}}}\ (E_{\text{electron}} + P_{\text{electron}})$: electron internal energy source

model = eulerThreeTemp

Vector of Source terms (*nodalArray*, 7-components) When *symmetryType = cylindrical*, the output source vector has components:

0. $\mathcal{S}\left(\rho\right) = -r^{-1}\rho\, u_{\hat{\mathbf{i}}}$: mass source
1. $\mathcal{S}\left(\rho\, u_{\hat{\mathbf{i}}}\right) = -r^{-1}\rho\, u_{\hat{\mathbf{i}}}^2 + \rho\, u_{\hat{\mathbf{j}}}^2$: $\hat{\mathbf{i}}$ momentum source
2. $\mathcal{S}\left(\rho\, u_{\hat{\mathbf{j}}}\right) = -2r^{-1}\rho\, u_{\hat{\mathbf{i}}}\, u_{\hat{\mathbf{j}}}$: $\hat{\mathbf{j}}$ momentum source
3. $\mathcal{S}\left(\rho\, u_{\hat{\mathbf{k}}}\right) = -r^{-1}\rho\, u_{\hat{\mathbf{i}}}\, u_{\hat{\mathbf{k}}}$: $\hat{\mathbf{k}}$ momentum source
4. $\mathcal{S}\left(E\right) = -r^{-1}u_{\hat{\mathbf{i}}}\ (E + P)$: total energy source
5. $\mathcal{S}\left(E_{\text{electron}}\right) = -r^{-1}u_{\hat{\mathbf{i}}}\ (E_{\text{electron}} + P_{\text{electron}})$: electron internal energy source
6. $\mathcal{S}\left(E_{\text{vibr}}\right) = -r^{-1}u_{\hat{\mathbf{i}}}\, E_{\text{vibr}}$: vibrational energy source

When *symmetryType = spherical*, the output source vector has components:

0. $\mathcal{S}\left(\rho\right) = -2r^{-1}\rho\, u_{\hat{\mathbf{i}}}$: mass source
1. $\mathcal{S}\left(\rho\, u_{\hat{\mathbf{i}}}\right) = -2r^{-1}\rho\, u_{\hat{\mathbf{i}}}^2 + \rho\, u_{\hat{\mathbf{j}}}^2$: $\hat{\mathbf{i}}$ momentum source
2. $\mathcal{S}\left(\rho\, u_{\hat{\mathbf{j}}}\right) = -3r^{-1}\rho\, u_{\hat{\mathbf{i}}}\, u_{\hat{\mathbf{j}}}$: $\hat{\mathbf{j}}$ momentum source
3. $\mathcal{S}\left(\rho\, u_{\hat{\mathbf{k}}}\right) = -3r^{-1}\rho\, u_{\hat{\mathbf{i}}}\, u_{\hat{\mathbf{k}}}$: $\hat{\mathbf{k}}$ momentum source
4. $\mathcal{S}\left(E\right) = -2r^{-1}u_{\hat{\mathbf{i}}}\ (E + P)$: total energy source
5. $\mathcal{S}\left(E_{\text{electron}}\right) = -2r^{-1}u_{\hat{\mathbf{i}}}\ (E_{\text{electron}} + P_{\text{electron}})$: electron internal energy source
6. $\mathcal{S}\left(E_{\text{vibr}}\right) = -2r^{-1}u_{\hat{\mathbf{i}}}\, E_{\text{vibr}}$: vibrational energy source

12.1.3 Example

The following block demonstrates *eulerSym* used in combination with *classicMusclUpdater (1d, 2d, 3d)* and *eulerEqn* to compute $\nabla \cdot \left[\mathcal{F}\left(\mathbf{w}\right)\right] - \mathcal{S}\left(\mathbf{w}, x, y, z, t\right)$:

```
<Updater hyper>
  kind = classicMuscl1d
  onGrid = domain
  timeIntegrationScheme = none
  variableForm = conservative
  preservePositivity = true

  in = [q]
  out = [qnew]

  cfl = 0.5
  limiter = [minmod]
  numericalFlux = hllcEulerFlux

  equations = [euler]
  sources = [geometricSrc]
```

```
  <Equation euler>
     kind = eulerEqn
     gasGamma = GAMMA # gas constant
  </Equation>

   <Source geometricSrc>
     kind = eulerSym
     symmetryType = spherical
     model = eulerEqn
     gasGamma = GAMMA
   </Source>
</Updater>
```

12.2 mhdSym

The *mhdSym* source computes additional (source) terms associated with curvilinear coordinate systems for magnetohydrodynamics. Currently only cylindrical coordinates are supported. The *mhdSym* source is combined with *classicMusclUpdater (1d, 2d, 3d)* and *mhdDednerEqn*. Note that formulation of the cylindrical source term assumes axisymmetry and so can be used in 1- and 2-dimensions.

12.2.1 Parameters

`symmetryType` (string, required) The curvilinear cordinate system to be used. Available options are *cylindrical*.

`model` (string, required) Determines the equation that the source term will be computed for. Available options are:

`mhdDednerEqn` Compute curvilinear source terms associated with *mhdDednerEqn*.

`gasGamma` (float, required) Specifies the adiabatic index (ratio of specific heats), γ.

`basementPressure` (float, optional) The minimum pressure allowed. Pressures below this value will be replaced with this value. Defaults to zero.

`basementDensity` (float, optional) The minimum density allowed. Densities below this value will be replaced with this value. Defaults to zero.

12.2.2 Parent Updater Data

`in` (string vector, required)

`Vector of Conserved Quantities` (*nodalArray*, 9-components, required) The vector of conserved quantities, $\mathbf{q}$ has 9 entries:

0. ρ: mass density
1. $\rho\, u_{\hat{\mathbf{i}}} = \rho\mathbf{u}\cdot\hat{\mathbf{i}}$: momentum density in the $\hat{\mathbf{i}}$ direction
2. $\rho\, u_{\hat{\mathbf{j}}} = \rho\mathbf{u}\cdot\hat{\mathbf{j}}$: momentum density in the $\hat{\mathbf{j}}$ direction
3. $\rho\, u_{\hat{\mathbf{k}}} = \rho\mathbf{u}\cdot\hat{\mathbf{k}}$: momentum density in the $\hat{\mathbf{k}}$ direction
4. $E = \frac{P}{\gamma-1} + \frac{1}{2}\rho|\mathbf{u}|^2 + \frac{1}{2}|\mathbf{b}|^2$: total energy density
5. $b_{\hat{\mathbf{i}}} = \mathbf{b}\cdot\hat{\mathbf{i}} = \mu_0^{-1/2}\mathbf{B}\cdot\hat{\mathbf{i}}$: magnetic field normalized by permeability of free-space in the $\hat{\mathbf{i}}$ direction

6. $b_{\hat{\mathbf{j}}} = \mathbf{b} \cdot \hat{\mathbf{j}} = \mu_0^{-1/2} \mathbf{B} \cdot \hat{\mathbf{j}}$: magnetic field normalized by permeability of free-space in the $\hat{\mathbf{j}}$ direction
7. $b_{\hat{\mathbf{k}}} = \mathbf{b} \cdot \hat{\mathbf{k}} = \mu_0^{-1/2} \mathbf{B} \cdot \hat{\mathbf{k}}$: magnetic field normalized by permeability of free-space in the $\hat{\mathbf{k}}$ direction
8. ψ: correction potential

`out` (string vector, required)

`model = mhdDednerEqn`

`Vector of Source terms` (*nodalArray*, 9-components, required) When *symmetryType* = *cylindrical*, the output source vector has components:

0. $\mathcal{S}(\rho) = -r^{-1} \rho \, u_{\hat{\mathbf{i}}}$: mass source
1. $\mathcal{S}\left(\rho \, u_{\hat{\mathbf{i}}}\right) = -r^{-1} \left(\rho \, u_{\hat{\mathbf{i}}}^2 - b_{\hat{\mathbf{i}}}^2 + \rho \, u_{\hat{\mathbf{j}}}^2 - b_{\hat{\mathbf{j}}}^2 + |\mathbf{b} \cdot \mathbf{b}|\right)$: $\hat{\mathbf{i}}$ momentum source
2. $\mathcal{S}\left(\rho \, u_{\hat{\mathbf{j}}}\right) = -2r^{-1} \left(\rho \, u_{\hat{\mathbf{i}}} \, u_{\hat{\mathbf{j}}} - b_{\hat{\mathbf{i}}} b_{\hat{\mathbf{j}}}\right)$: $\hat{\mathbf{j}}$ momentum source
3. $\mathcal{S}\left(\rho \, u_{\hat{\mathbf{k}}}\right) = -r^{-1} \left(\rho \, u_{\hat{\mathbf{i}}} \, u_{\hat{\mathbf{k}}} - b_{\hat{\mathbf{i}}} b_{\hat{\mathbf{k}}}\right)$: $\hat{\mathbf{k}}$ momentum source
4. $\mathcal{S}(E) = -r^{-1} \left[u_{\hat{\mathbf{i}}} \, (E + P) + (\mathbf{e} \times \mathbf{b}) \cdot \hat{\mathbf{i}}\right]$: total energy source
5. $\mathcal{S}\left(b_{\hat{\mathbf{i}}}\right) = 0$: $\hat{\mathbf{i}}$ magnetic field source
6. $\mathcal{S}\left(b_{\hat{\mathbf{j}}}\right) = 0$: $\hat{\mathbf{j}}$ magnetic field source
7. $\mathcal{S}\left(b_{\hat{\mathbf{k}}}\right) = -e_{\hat{\mathbf{j}}}$: $\hat{\mathbf{k}}$ magnetic field source
8. $\mathcal{S}(\psi) = -c_{\text{fast}}^2 b_{\hat{\mathbf{i}}}$: correction potential source

12.2.3 Example

The following block demonstrates the *mhdSym* source used in combination with *classicMusclUpdater (1d, 2d, 3d)* and *mhdDednerEqn* to compute $\nabla \cdot [\mathcal{F}(\mathbf{w})] - \mathcal{S}(\mathbf{w}, x, y, z, t)$:

```
<Updater hyper>
  kind = classicMuscl2d
  timeIntegrationScheme = none
  onGrid = domain
  limiter = [muscl,none,none]

  variableForm = primitive
  numericalFlux = roeFlux
  preservePositivity = true
  correctUnphysicalCells = false

  orderAccuracy = 3
  numberOfInterpolationPoints = 20
  formulation = spline
  leastSquaresBasisOrder = 6

  in = [q,divB,gradPsi]
  out = [qnew]
  waveSpeeds = [waveSpeed]

  cfl = CFL
```

```
  equations = [mhd]
  sources = [axisymmetricSource]

  <Equation mhd>
    kind = mhdDednerEqn
    mu0 = 1.0
    gasGamma = ADIABATIC_INDEX
    correctionSpeed = CORRECTION_SPEED
    basementdensity = BASEMENTDENSITY
    basementpressure = BASEMENTPRESSURE
  </Equation>

   <Source axisymmetricSource>
     kind = mhdSym
     symmetryType = cylindrical
     model = mhdDednerEqn
     gasGamma = ADIABATIC_INDEX
     correctionSpeed = CORRECTION_SPEED
   </Source>

</Updater>
```

12.3 maxwellSym

Computes axisymmetric source term for the conservative form of the perfectly hyperbolic Maxwell's equations. The source term can be used as a source in a hyperbolic algorithm to solve axisymmetric problems.

$$s = \frac{1}{r}\begin{pmatrix} 0 \\ 0 \\ c^2 B_y \\ 0 \\ 0 \\ -E_y \\ -\chi E_x \\ -\gamma c^2 B_x \end{pmatrix}$$

Where c is the speed of light, χ is electric correction potential speed factor, γ is the magnetic field correction potential speed factor, E_x is x electric field, E_y is the y electric field, B_x is x magnetic field, B_y is the y magnetic field and r is the radial position.

12.3.1 Parameters

`speedOfLight` (float) speed of light

`gamma` (float) Magnetic field divergence error correction speed factor, speed=gamma*c0

`chi` (float) Electric field poisson error correction speed factor, speed=chi*c0

12.3.2 Example

```
<Source emAxisymmetricSource>
  kind = maxwellSym
  speedOfLight = SPEEDOFLIGHT
```

```
  gamma = BP
  chi = EP
</Source>
```

12.4 multiSpeciesSym

The multiSpeciesSym provides symmetry sources for the multi species continuity equations (multiSpeciesSingleVelocity) for example. Choices are cylindrical and spherical symmetry. For a set of n continuity equations the cylindrical source is given by

$$s_i = -\frac{1}{r}\left(\; n_i \, u_x \; \right)$$

and the spherical symmetry source by

$$s_i = -\frac{2}{r}\left(\; n_i \, u_x \; \right)$$

12.4.1 Parameters

`basementDensity` (float) If the number density is below basementDensity then the density is set to the basementNumber density in evaluating this source.

`numberOfSpecies` (int) The number of species

`symmetryType` (string) Either *cylindrical* or *spherical*

12.4.2 Parent Updater Data

`in` (string vector, required)

`1st Variable` (nodalArray) There are *n* species number densities

0. n_i species number density

`2nd Variable (density and momentum)` (nodalArray)

0. ρ mass density
1. $\rho \, u_x$ x momentum density
2. $\rho \, u_y$ y momentum density
3. $\rho \, u_z$ z momentum density

12.4.3 Example

```
<Source multiSpeciesAxiSrc>
  kind = multiSpeciesSym
  symmetryType = cylindrical
  numberOfSpecies = NSPECIES
</Source>
```

12.5 twoFluidSym

Computes symmetry source term for the conservative for of the combined two-fluid equations. The source just converts the combined two-fluid into two separate euler fluids and computes the source for each fluid separately. The result is then transformed back into the combined fluid.

12.5.1 Parameters common to all systems

`model` (string) The model whose source term will be computed

`symmetryType` (string) The symmetry type that will be used. This can be either *cylindrical* or *spherical*

12.5.2 Parameters (twoFluidEqn)

`ionGamma` (float) Specific heat ratio of the ions

`electronGamma` (float) Specific heat ratio of the electrons

`mi` (float) ion mass

`me` (float) electron mass

`qi` (float) ion charge

`qe` (float) electron charge

`basementDensity` (float) basement density of the ions. Basement density of the electrons is (me/me)*basementDensity.

`basementPressure` (float) basement pressure of the electrons and ions separately.

12.5.3 Parent Updater Data (twoFluidEqn)

`in` (string vector, required) 10 primary variables

0. ρ mass density
1. $\rho\, u_x$ x momentum density
2. $\rho\, u_y$ y momentum density
3. $\rho\, u_z$ z momentum density
4. ρ_c total charge density
5. j_x x current density
6. j_y y current density
7. j_z z current density
8. e_i ion energy density
9. e_e electron energy density

12.5.4 Example

```
<Source axiSymSource>
  kind = twoFluidSymSrc
  symmetryType = cylindrical
  model = twoFluidEqn
  ionGamma = 1.666
  electronGamma = 1.6666
  qi = ION_CHARGE
  qe = ELECTRON_CHARGE
  mi = ION_MASS
  me = ELECTRON_MASS
  basementDensity = 0.0
  basementPressure = 0.0
</Source>
```

The following kinds can be combined with *Hyperbolic Equations* to enable additional physics in hyperbolic problems:

12.6 exprHyperSrc

Computes a source term based on input variables from N *nodalArray*s in combination with geometric factors.

12.6.1 Parameters

`indVars_inName` (string vector, required) For each input variable an "indVars" string vector must be defined. So if *in = [magneticField, electricField]* where *magneticField* and *electricField* are each 3-component *nodalArray*s then the combiner block must define *indVars_magneticField = ["bx", "by", "bz"]* and *indVars_electricField = ["ex", "ey", "ez"]*. Note that the labels *"bx", "by", "bz"* and *"ex", "ey", "ez"* are arbitrary; the requirement is that there is a unique name for each component of each input data structure.

`exprs` (string vector, required) Strings must be put in quotes. The strings are evaluated and placed in the output array. The number of strings must be identical to the number of components in the output array. Available command are defined by the muParser (http://muparser.sourceforge.net/)

`preExprs` (string vector, optional) Strings must be put in quotes. The preExprs is used to compute quantities based on indVars that can later be used in the *exprs* to evaluate the output. Available commands are defined by the muParser (http://muparser.sourceforge.net)

`other` (strings, optional) In addition, an arbitrary number of constants can be defined that can then be used in evaluating expression in both *preExprs* and *exprs*.

12.6.2 Parent Updater Data

`in` (string vector, required) Inputs 1 to N are input *nodalArray*s which will be supplied to the expression evaluator.

`out` (string vector, required) Output is a *nodalArray* which will contain evaluated source. The number of components in the output array must be equal to the number of expressions.

12.6.3 Example

The following block demonstrates *exprHyperSrc* used in combination with *classicMusclUpdater (1d, 2d, 3d)* and *eulerEqn* to compute $\nabla \cdot [\mathcal{F}(\mathbf{w})] - \mathcal{S}(\mathbf{w}, x, y, z, t)$:

```
 <Updater hyper>
  kind = classicMuscl2d
  onGrid = domain
  timeIntegrationScheme = none
  numericalFlux = hllcEulerFlux

  limiter = [muscl]
  variableForm = primitive

  in = [q]
  out = [qnew]

  cfl = 0.3

  equations = [euler]
  sources = [gravity]

  <Equation euler>
    kind = eulerEqn
    correctNans = false
    gasGamma = GAMMA
  </Equation>

  <Source gravity>
    kind = exprHyperSrc
     inpRange = [0,1,2,3,4]
     outRange = [0,1,2,3,4]
     gravity = GRAVITY # m/s^2
     indVars = ["rho", "rhou", "rhov", "rhow", "Er"]
     exprs = ["0.0", "0.0", "-rho*gravity", "0.0", "-gravity*rhov"]
  </Source>

</Updater>
```

12.7 mhdSrc

Many of the MHD system require source terms as the hyperbolic part is not sufficient to describe the simple system. This source contains the source terms that should be added to these models.

Sources include:

The idealMHD source terms. The ideal MHD case only applies when one does not convert the source terms to a conservative flux. Note that we do not need separate source for general equation of state since the source terms

are independent of the relation between internal energy and pressure.

$$s = \begin{pmatrix} 0 \\ J_y B_z - J_z B_y \\ J_z B_x - J_x B_z \\ J_x B_y - J_y B_x \\ J_x E_x + J_y E_y + J_z E_z \\ 0 \\ 0 \\ 0 \end{pmatrix}$$

The gasDynamicMhdEqn source terms (which include an electron energy equation)

$$s = \begin{pmatrix} 0 \\ J_y B_z - J_z B_y \\ J_z B_x - J_x B_z \\ J_x B_y - J_y B_x \\ J_x E_x + J_y E_y + J_z E_z \\ 0 \\ 0 \\ 0 \\ 0 \\ J_{x\,e} E_x + J_{y\,e} E_y + J_{z\,e} E_z \end{pmatrix}$$

The twoTemperatureMhdEqn source terms

$$s = \begin{pmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ J_{x\,e} E_x + J_{y\,e} E_y + J_{z\,e} E_z \end{pmatrix}$$

12.7.1 Parameters (twoTemperatureMhdEqn and twoTemperatureMhdEosEqn)

`ionMass` (float) The mass of an ion

`fundamentalCharge` proton charge

12.7.2 Parent Updater Data (idealMhd)

`in` (string vector, required) 1st Variable

0. ρ mass density
1. $\rho\, u_x$ x momentum density
2. $\rho\, u_y$ y momentum density
3. $\rho\, u_z$ z momentum density
4. e energy density

5. B_x x magnetic field
6. B_y y magnetic field
7. B_z z magnetic field

2nd Variable (current density)

0. J_x x current density
1. J_y y current density
2. J_z z current density

3rd Variable (electric field)

0. e_x x electric field
1. e_y y electric field
2. e_z z electric field

12.7.3 Parent Updater Data (twoTemperatureMhdEqn and gasDynamicMhdEqn)

`in` (string vector, required) 1st Variable

0. ρ mass density
1. $\rho\, u_x$ x momentum density
2. $\rho\, u_y$ y momentum density
3. $\rho\, u_z$ z momentum density
4. e energy density
5. B_x x magnetic field
6. B_y y magnetic field
7. B_z z magnetic field
8. Phi correction potential
9. e_e electron energy

2nd Variable (current density)

0. J_x x current density
1. J_y y current density
2. J_z z current density

3rd Variable (electric field)

0. e_x x electric field
1. e_y y electric field
2. e_z z electric field

4th Variable (charge state)

0. Z charge state

12.7.4 Parent Updater Data (twoTemperatureMhdEosEqn)

`in` (string vector, required) 1st Variable

0. ρ mass density
1. $\rho\, u_x$ x momentum density
2. $\rho\, u_y$ y momentum density
3. $\rho\, u_z$ z momentum density
4. e energy density
5. B_x x magnetic field
6. B_y y magnetic field
7. B_z z magnetic field
8. e_e electron energy equation

2nd Variable (current density)

0. J_x x current density
1. J_y y current density
2. J_z z current density

3rd Variable (electric field)

0. e_x x electric field
1. e_y y electric field
2. e_z z electric field

4th Variable (charge state)

0. Z charge state

12.7.5 Example

```
<Source axisymmetricSource>
  kind = mhdSrc
  model = gasDynamicMhdSrc
</Source>
```

12.8 tenMomentFluidSrc

Computes the "lorentz force" for a 10 moment fluid given, particle mass, charge and permittivity.

$$s = \begin{pmatrix} 0 \\ r\,\rho\,(E_x + u_y B_z - u_z B_y) \\ r\,\rho\,(E_y + u_z B_x - u_x B_z) \\ r\,\rho\,(E_z + u_x B_y - u_y B_x) \\ 2\,r\,\rho\,u_x E_x + 2r\,(B_z \mathbf{P_{xy}} - B_y \mathbf{P_{xz}}) \\ r\,\rho\,(u_x E_y + u_y E_x) + r\,(B_z \mathbf{P_{yy}} + B_y \mathbf{P_{yz}} - B_z \mathbf{P_{xx}} + B_x \mathbf{P_{xz}}) \\ r\,\rho\,(u_x E_z + u_z E_x) + r\,(B_z \mathbf{P_{yz}} + B_y \mathbf{P_{xx}} - B_y \mathbf{P_{xx}} - B_x \mathbf{P_{yy}}) \\ 2r\,\rho\,u_y E_y + 2r\,(B_x \mathbf{P_{yz}} - B_z \mathbf{P_{xy}}) \\ r\,\rho\,(u_y E_z + u_z E_y) + r\,(B_y \mathbf{P_{xy}} - B_z \mathbf{P_{xz}} + B_x \mathbf{P_{zz}} - B_x \mathbf{P_{yy}}) \\ 2r\,\rho\,u_z E_z + 2r\,(B_y \mathbf{P_{xz}} - B_x \mathbf{P_{yz}}) \end{pmatrix}$$

where q is the species charge, m is the species mass ϵ_0 is the permittivity, ρ is the fluid mass density, u_x is the fluid x velocity, u_y is the fluid y velocity, u_z is the fluid z velocity, E_x is the x electric field, E_y is the y electric field, E_z is the z electric field, B_x is the x magnetic field, B_y is the y magnetic field and B_z is the z magnetic field. $\mathbf{P_{ij}} = P_{i\,j} + \rho\,u_i u_j$ with $P_{i\,j}$ the pressure tensor and ρ the mass density and $r = q/m$ the charge to mass ratio.

12.8.1 Parameters

`mass` (float) The mass of the fluid species

`charge` (float) The charge of the fluid species

`type` (string, default='unsplit') One of either *split* or *unsplit*

12.8.2 Data

`inputVariables` (string vector)

`1st Variable` (nodalArray)

0. ρ mass density
1. $\rho\,u_x$ x momentum density
2. $\rho\,u_y$ y momentum density
3. $\rho\,u_z$ z momentum density
4. $\rho\,u_x^2 + P_{x\,x}$ xx energy density
5. $\rho\,u_x\,u_y + P_{x\,y}$ xy energy density
6. $\rho\,u_x\,u_z + P_{x\,z}$ xz energy density
7. $\rho\,u_y^2 + P_{y\,y}$ yy energy density
8. $\rho\,u_y\,u_z + P_{y\,z}$ yz energy density
9. $\rho\,u_z^2 + P_{z\,z}$ zz energy density

`2nd Variable` (nodaArray)

0. E_x x electric field
1. E_y y electric field

2. E_z z electric field
3. B_x x magnetic field
4. B_y y magnetic field
5. B_z z magnetic field

12.8.3 Parent Updater Data

`in` (string vector, required) The *nodalArray*s should match the inputVariables in the source block.

`1st Variable` (nodalArray)

0. ρ mass density
1. $\rho\, u_x$ x momentum density
2. $\rho\, u_y$ y momentum density
3. $\rho\, u_z$ z momentum density
4. $\rho\, u_x^2 + P_{x\,x}$ xx energy density
5. $\rho\, u_x\, u_y + P_{x\,y}$ xy energy density
6. $\rho\, u_x\, u_z + P_{x\,z}$ xz energy density
7. $\rho\, u_y^2 + P_{y\,y}$ yy energy density
8. $\rho\, u_y\, u_z + P_{y\,z}$ yz energy density
9. $\rho\, u_z^2 + P_{z\,z}$ zz energy density

`2nd Variable` (nodaArray)

0. E_x x electric field
1. E_y y electric field
2. E_z z electric field
3. B_x x magnetic field
4. B_y y magnetic field
5. B_z z magnetic field

`out` (string vector, required) The output *nodalArray* is a length 10 vector, but the first component is 0 so that it works simply as a fluid source for the ten moment equations.

12.8.4 Example

```
<Updater hyperIons>
  kind = classicMuscl2d
  onGrid = domain
  timeIntegrationScheme = none
  numericalFlux = hlleFlux
  preservePositivity = true
  limiter = [mc,none]

  variableForm = conservative

  in = [ions, em]
```

```
    out = [ionsNew]

    cfl = CFL
    equations = [euler]
    sources = [lorentz]

    <Equation euler>
      kind = tenMomentEqn
      basementDensity = BASEMENT_DENSITY
      basementPressure = BASEMENT_PRESSURE
    </Equation>

    <Source lorentz>
      kind = tenMomentFluidSrc
      type = split
      inputVariables = [ions, em]
      mass = ION_MASS
      charge = ION_CHARGE
    </Source>

  </Updater>
```

12.9 twoFluidSrc

Applies the implicit source operator to the 5 moment two-fluid (ion, electron, EM) system or the two-fluid system written as combined variables *twoFluidEqn* or the 10-5 system which is 10 moment ions and 5 moment electrons and EM.

We want to solve the hyperbolic part of the multi-fluid equations explicitly and the source term implicitly. For a first order scheme the discretization becomes.

$$Q^{n+1} = Q^n + \Delta t \nabla f^n + \Delta t \psi^{n+1} \tag{12.-9}$$

In the case of the two-fluid and 10 moment systems ψ^{n+1} can be re-written exactly as $A^{n+1}Q^{n+1}$ where A is a matrix. As a result the equation can be re-written

$$\left(1 - \Delta t A^{n+1}\right) Q^{n+1} = \Delta t \nabla f^n \tag{12.-9}$$

and therefore

$$Q^{n+1} = \left(1 - \Delta t A^{n+1}\right)^{-1} \Delta t \nabla f^n \tag{12.-9}$$

Now, for a Runge-Kutta scheme this update is performed for each substep and Δt is replaced by $\alpha \Delta t$ where α is the fractional Δt for each substep. The result is a high order time accurate implicit integration of the source terms.

For the 5 moment system (and combined system) only one call to this source term is required. For the 10-5 system the update must be performed in 2 steps. Examples are given below. The technique used was originally described for the 5 moment two-fluid system in

Kumar, Harish, and Siddhartha Mishra. "Entropy Stable Numerical Schemes for Two-Fluid Plasma Equations." Journal of Scientific Computing 52.2 (2012): 401-425.

12.9.1 Parameters (All types)

`type` string Specifies the type of implicit matrix. Options are 5Moment for the 5 moment two-fluid system, 10MomentIonsStep1 for the first step of the 10 moment ion, 5 moment electron system. 10MomentIonStep2 for the second step of the 10 moment ion, 5 moment electron two-fluid system.

12.9.2 Parameters (5Moment or 5MomentCombined or 10MomentIonStep1)

`useImposedField` (bool) Tell USim if there will be an imposed magnetic field applied to the model. If true an imposed field will be assumed, if false there is no imposed field.

`electronCharge` (float) The charge of the electron

`electronMass` (float) The mass of the electron

`ionCharge` (float) The charge of the ion

`ionMass` (float) The mass of the ion

`epsilon0` Permittivity of free space

12.9.3 Parameters (10MomentIonStep2)

`ionCharge` (float) The charge of the ion

`ionMass` (float) The mass of the ion

12.9.4 Parent Updater Data (5Moment)

`in` (string vector, required) 1st Variable

0. ρ electron mass density
1. $\rho\, u_x$ electron x momentum density
2. $\rho\, u_y$ electron y momentum density
3. $\rho\, u_z$ electron z momentum density
4. e electron energy density

2nd Variable

0. ρ ion mass density
1. $\rho\, u_x$ ion x momentum density
2. $\rho\, u_y$ ion y momentum density
3. $\rho\, u_z$ ion z momentum density
4. e ion energy density

3rd Variable

0. E_x x electric field
1. E_y y electric field
2. E_z z electric field
3. B_x x magnetic field
4. B_y y magnetic field
5. B_z z magnetic field
6. Ψ_E electric field correction potential
7. Ψ_B magnetic field correction potential

4th Variable (if useImposedField = true)

This term stores the perturbed field (the total field - the imposed field)

0. E_x x electric field
1. E_y y electric field
2. E_z z electric field
3. B_x x magnetic field
4. B_y y magnetic field
5. B_z z magnetic field
6. Ψ_E electric field correction potential
7. Ψ_B magnetic field correction potential

`out` (string vector, required) In all cases the output is Q^{n+1}. For the 5 moment system there are 3 outputs corresponding to electrons, ions and em (in that order).

12.9.5 Parent Updater Data (5Moment Combined)

`in` (string vector, required) 1st Variable

0. ρ mass density
1. $\rho\, u_x$ x momentum density
2. $\rho\, u_y$ y momentum density
3. $\rho\, u_z$ z momentum density
4. ρ_c total charge density
5. j_x x current density
6. j_y y current density
7. j_z z current density
8. e_i ion energy density
9. e_e electron energy density

2nd Variable

0. E_x x electric field
1. E_y y electric field
2. E_z z electric field
3. B_x x magnetic field
4. B_y y magnetic field
5. B_z z magnetic field
6. Ψ_E electric field correction potential
7. Ψ_B magnetic field correction potential

3rd Variable (if useImposedField = true)

This term stores the externally imposed field

0. E_x x electric field
1. E_y y electric field
2. E_z z electric field
3. B_x x magnetic field
4. B_y y magnetic field
5. B_z z magnetic field
6. Ψ_E electric field correction potential
7. Ψ_B magnetic field correction potential

`out` (string vector, required) In all cases the output is Q^{n+1}. For the combined 5 moment systems there are 2 outputs, one for the combined fluid and the other for em (in that order).

12.9.6 Parent Updater Data (10MomentIonsStep1)

`in` (string vector, required) 1st Variable

0. ρ electron mass density
1. $\rho\, u_x$ electron x momentum density
2. $\rho\, u_y$ electron y momentum density
3. $\rho\, u_z$ electron z momentum density
4. e electron energy density

2nd Variable

0. ρ ion mass density
1. $\rho\, u_x$ ion x momentum density
2. $\rho\, u_y$ ion y momentum density
3. $\rho\, u_z$ ion z momentum density
4. $\rho\, u_x^2 + P_{x\,x}$ ion xx energy density
5. $\rho\, u_x\, u_y + P_{x\,y}$ ion xy energy density
6. $\rho\, u_x\, u_z + P_{x\,z}$ ion xz energy density
7. $\rho\, u_y^2 + P_{y\,y}$ ion yy energy density
8. $\rho\, u_y\, u_z + P_{y\,z}$ ion yz energy density
9. $\rho\, u_z^2 + P_{z\,z}$ ion zz energy density

3rd Variable

0. E_x x electric field
1. E_y y electric field
2. E_z z electric field
3. B_x x magnetic field
4. B_y y magnetic field
5. B_z z magnetic field

4th Variable (if useImposedField = true)

This term stores the externally imposed field

0. E_x x electric field
1. E_y y electric field
2. E_z z electric field
3. B_x x magnetic field
4. B_y y magnetic field
5. B_z z magnetic field
6. Ψ_E electric field correction potential
7. Ψ_B magnetic field correction potential

`out` (string vector, required) In all cases the output is Q^{n+1}. For the 5 moment electron, 10 moment ion system in the case where type=10MomentIonStep1 the output is 5 moment electrons, 10 moment ions and 8 component EM system.

12.9.7 Parent Updater Data (10MomentIonsStep2)

`in` (string vector, required) 1st Variable

0. ρ ion mass density
1. $\rho\, u_x$ ion x momentum density
2. $\rho\, u_y$ ion y momentum density
3. $\rho\, u_z$ ion z momentum density
4. $\rho\, u_x^2 + P_{x\,x}$ ion xx energy density
5. $\rho\, u_x\, u_y + P_{x\,y}$ ion xy energy density
6. $\rho\, u_x\, u_z + P_{x\,z}$ ion xz energy density
7. $\rho\, u_y^2 + P_{y\,y}$ ion yy energy density
8. $\rho\, u_y\, u_z + P_{y\,z}$ ion yz energy density
9. $\rho\, u_z^2 + P_{z\,z}$ ion zz energy density

2nd Variable

0. E_x x electric field
1. E_y y electric field
2. E_z z electric field
3. B_x x magnetic field
4. B_y y magnetic field
5. B_z z magnetic field

3rd Variable (if useImposedField = true)

This term stores the externally imposed field

0. E_x x electric field
1. E_y y electric field

2. E_z z electric field
3. B_x x magnetic field
4. B_y y magnetic field
5. B_z z magnetic field
6. Ψ_E electric field correction potential
7. Ψ_B magnetic field correction potential

out (string vector, required) In all cases the output is Q^{n+1}. In the case where type=10MomentIonStep2 the output is 10 moment ions.

12.9.8 Example

```
<Equation twofluidLorentz>
  kind = twoFluidSrc
  type = 5Moment
  electronCharge = ELECTRON_CHARGE
  electronMass = ELECTRON_MASS
  ionCharge = ION_CHARGE
  ionMass = ION_MASS
  epsilon0 = EPSILON0
</Equation>

<Equation twofluidLorentz>
  kind = twoFluidSrc
  type = 5MomentCombined
  useImposedField = false
  electronCharge = ELECTRON_CHARGE
  electronMass = ELECTRON_MASS
  ionCharge = ION_CHARGE
  ionMass = ION_MASS
  epsilon0 = EPSILON0
</Equation>

<Equation twofluidLorentz>
  kind = twoFluidSrc
  type = 10MomentIonsStep1
  electronCharge = ELECTRON_CHARGE
  electronMass = ELECTRON_MASS
  ionCharge = ION_CHARGE
  ionMass = ION_MASS
  epsilon0 = EPSILON0
</Equation>

<Equation twofluidLorentz>
  kind = twoFluidSrc
  type = 10MomentIonsStep2
  ionCharge = ION_CHARGE
  ionMass = ION_MASS
</Equation>
```

The following kinds can be used to couple fluid models with equations of state:

12.10 idealGasVariables

This source allows the user to compute specific internal energy (ϵ), pressure (P), density (ρ), and temperature (T) from the ideal gas law,

$$\rho\epsilon = P/(\Gamma - 1) = \rho k_B T/m_i(\Gamma - 1).$$

Here Γ is the adiabatic gas index, m_i is the species mass and k_B is the Boltzmann constant.

This updater is intended for usage as an example. As many equation updaters in USim assume an ideal gas equation of state use of this updater is redundant and largely unnecessary. Rather, this is updater is used to provide examples of EOS usage when EOS tables are not available.

12.10.1 Parameters

`operations` (string vector, required) The operation(s) to be performed. The standard direct operation is "computeEOSFromTemperatureAndDensity" where "EOS" should be replaced by one of the following values (the first word of each option should be used, the remainder offers a brief description and the default units after conversion):

- energy - (Jkg^{-1})
- pressure - (Pa)

To compute an inverse operation, simply permute the string to be "computeTemperatureFromEOSAndDensity" or "computeDensityFromTemperatureAndEOS". The input is not case sensitive.

`gasGamma` (float, required) Specifies the adiabatic index (ratio of specific heats), Γ.

`speciesMass` (float, required) Specifies the species mass in kg.

`kboltz` (float, optional) Specifies the Boltzmann constant. Defaults to 1.3806×10^{-23} (J/K).

`useParticleDensity` (int, optional) Whether to use the particle (true) or mass (false) density. Default is false.

`densityConversionCoefficient` (float, optional) Custom density unit conversion factor. Conversion to MKS mass density (kgm^{-3}) is the default. The default conversion factor is divided by the custom conversion factor. Thus if using alternative units, set the unit conversion factor to to the MKS value that corresponds to unity in the alternative units.

`temperatureConversionCoefficient` (float, optional) Custom temperature unit conversion factor. Conversion to MKS temperature (K) is the default. The default conversion factor is divided by the custom conversion factor. Thus if using alternative units, set the unit conversion factor to to the MKS value that corresponds to unity in the alternative units.

`conversionCoefficients` (float vector, optional) Custom unit conversion factors for EOS values. Conversion to MKS units is the default. The default conversion factor is divided by the custom conversion factor. Thus if using alternative units, set the unit conversion factor to to the MKS value that corresponds to unity in the alternative units.

12.10.2 Parent Updater Data

`in` (string vector, required)

`input variables` (*nodalArray*, 1-component each, 2 required)

The specific input variables and order depend on the **operations** input option. For direct EOS evaluation, the input variables should be in = [temperature, density]. The order is critical where temperature

must be the first input and density must be the second input. For inverse operations, the temperature, if an input, must be the first input and the density, if an input, must be the second input. The EOS input should be placed in the correspondingly empty input location. Inputs are of type *nodalArray* with one component each.

`out` (string vector, required)

`output variables` (*nodalArray*, 1-component each, required)

The number of out variables should be the same as the number of entries into the list of **operations**. The result of each operation will be placed into the corresponding output variable, respectively. Outputs are of type *nodalArray* with one component each.

12.10.3 Example

```
<Updater computeEOS>
  kind=equation2d
  onGrid=domain
  in=[temperature, density]
  out=[energy, pressure]
  <Equation thisGas>
    kind=idealGasVariables
    operations=["computeEnergyFromDensityAndTemperature", \
                "computePressureFromDensityAndTemperature"]
  </Equation>
</Updater>
```

12.11 idealGasComputeVariables

This source allows the user to compute specific internal energy (ϵ), pressure (P), density (ρ), and temperature (T) from the ideal gas law,

$$\rho\epsilon = P/(\Gamma - 1) = \rho k_B T/m_i(\Gamma - 1).$$

Here Γ is the adiabatic gas index, m_i is the species mass and k_B is the Boltzmann constant.

In this updater, the sound speed squared is computed from a formula for the generalized sound speed:

$$c_s^2 = \frac{\partial P}{\partial \epsilon}\frac{P}{\rho^2} + \frac{\partial P}{\partial \rho}$$

This updater is intended for usage as an example. As many equation updaters in USim assume an ideal gas equation of state use of this updater is redundant and largely unnecessary. Rather, this is updater is used to provide examples of EOS usage when EOS tables are not available.

12.11.1 Parameters

`gasGamma` (float, required) Specifies the adiabatic index (ratio of specific heats), Γ.

`speciesMass` (float, required) Specifies the species mass in kg.

`kboltz` (float, optional) Specifies the Boltzmann constant. Defaults to 1.3806×10^{-23} (J/K).

`delta` (float, optional) A finite difference operation is applied to evaluate partial derivatives. This factor determines the relative width of the stencil. The default is 10^{-6}.

soundSpeedSquaredFloor (float, optional) Sets a minimum value for the output sound speed squared. The default is 0.

useParticleDensity (int, optional) Whether to use the particle (true) or mass (false) density. Default is false.

densityConversionCoefficient (float, optional) Custom density unit conversion factor. Conversion to MKS mass density (kgm^{-3}) is the default. The default conversion factor is divided by the custom conversion factor. Thus if using alternative units, set the unit conversion factor to to the MKS value that corresponds to unity in the alternative units.

temperatureConversionCoefficient (float, optional) Custom temperature unit conversion factor. Conversion to MKS temperature (K) is the default. The default conversion factor is divided by the custom conversion factor. Thus if using alternative units, set the unit conversion factor to to the MKS value that corresponds to unity in the alternative units.

conversionCoefficients (float vector, optional) Custom unit conversion factors for EOS values. Conversion to MKS units is the default. The default conversion factor is divided by the custom conversion factor. Thus if using alternative units, set the unit conversion factor to to the MKS value that corresponds to unity in the alternative units.

outputPeRhoInv (int, optional) Boolean that determines if the partial derivative of the pressure with respect to specific energy divided by the density, $\rho^{-1}\partial P/\partial\epsilon$, is output. This output is required to compute the EOS system eigenvectors. The default is false.

12.11.2 Parent Updater Data

`in` (string vector, required)

`input variables` (*nodalArray*, 1-component each, 2 required)

The input variables (exactly 2) must be the density and the internal energy, in that order. Inputs are of type *nodalArray* with one component each.

`out` (string vector, required)

`output variables` (*nodalArray*, 1-component each, 2 required and 3rd optional)

The output variables are the pressure and the sound speed squared, in that order. If outputPeRhoInv is true, a third output variable that is the partial derivative of the pressure with respect to specific energy divided by the density $\rho^{-1}\partial P/\partial\epsilon$. This output is required to compute the EOS system eigenvectors. Outputs are of type *nodalArray* with one component each.

12.11.3 Example

```
<Updater computePressureAndSoundSpeedSquared>
  kind=equation2d
  onGrid=domain
  in=[rho, intEnergy]
  out=[pressure, soundSqr]
  <Equation thisGas>
    kind=idealGasComputeVariables
    delta=1.e-5
    speciesMass=MI
    gasGamma=1.667
  </Equation>
</Updater>
```

12.12 propaceosVariables

This source allows the user to read in data from a PROPACEOS table and then compute energy, density, temperature and single group and multi-group emissivities. PROPACEOS tables can be obtained from Prism Computational Sciences (PROPACEOS link). Alternatively the PROPACEOS format can be used to create your own tables. Tables specify an equation of state (EOS) for energy, single group and multi-group emissivities as a function of temperature and density. To solve for temperature or density as a function of the EOS table value an inverse operation must be applied. This operation holds the input temperature or density constant and assumes the EOS table data is a monotonic function of the dependent variables (density and temperature). If these assumptions do not hold, incorrect results may be produced.

A note on units. Units in USim are all MKS units. However, the PROPACEOS tables use CGS units and eV for temperature. These units are converted to MKS by USim. This is important if one writes their own PROPACEOS tables. The ability to specify custom unit conversion factors is available as an optional input.

Before running any case using the PROPACEOS EOS tables, it is prudent to make basic sanity checks by running a modified version of the *verifyEOSTable* example with the specific PROPACEOS table that is intended for use.

12.12.1 Parameters

`filename` (string, required) Name of file that contains the PROPACEOS formatted table.

`operations` (string vector, required) The operation(s) to be performed. The standard direct operation is "computeEOSTableFromTemperatureAndDensity" where "EOSTable" is computed from the PROPACEOS tables and should be replaced by one of the following values (the first word of each option should be used, the remainder offers a brief description and the default units after conversion):

- Zbar - the charge state
- Eint - Total internal energy (Jkg^{-1})
- Eion - Ion internal energy (Jkg^{-1})
- Eele - Electron internal energy (Jkg^{-1})
- Pion - Ion pressure (Pa)
- Pele - Electron pressure (Pa)
- Ptot - Pion+Pele (Pa)
- IntRosseland - Integrated Rosseland Mean Opacity (m^2kg^{-1})
- IntAbsPlanck - Integrated Planck Mean Opacity (m^2kg^{-1})
- IntEmisPlank - Integrated Planck Mean Opacity (m^2kg^{-1})
- Zeffective - effective Z for Bremstrahlung radiation
- Rosseland - Rosseland mean opacity for frequency group (m^2kg^{-1})
- AbsPlanck - absorption Planck mean opacity for frequency group (m^2kg^{-1})
- EmisPlanck - emission Planck mean opacity for frequency group (m^2kg^{-1})
- IonizationFraction - the ionization fraction

To compute an inverse operation, simply permute the string to be "computeTemperatureFromEOSTableAndDensity" or "computeDensityFromTemperatureAndEOSTable". The input is not case sensitive.

`speciesMass` (float, required) Mass of the species as required to convert from number to mass density.

useParticleDensity (int, optional) Whether to use the particle (true) or mass (false) density. Default is false.

element (string, optional) Element name used for computing ionization fraction.

elementList (string vector, optional) List of elements used in the table for computing Zeffective.

fixRanges (int vector, optional)

Whether the variables should be allowed to go beyond the table ranges or not. fixRanges = [1] means that the first variable cannot go beyond the table ranges and if it does, it's value is set to the maximum (or minimum) of the table value. The default is false.

logInterpolation (int vector, optional)

Whether to use logarithmic interpolation when evaluating EOS table values. The default is false.

densityConversionCoefficient (float, optional)

Custom density unit conversion factor. Conversion to MKS mass density (kgm^{-3}) is the default. The default conversion factor is divided by the custom conversion factor. Thus if using alternative units, set the unit conversion factor to to the MKS value that corresponds to unity in the alternative units.

temperatureConversionCoefficient (float, optional)

Custom temperature unit conversion factor. Conversion to MKS temperature (K) is the default. The default conversion factor is divided by the custom conversion factor. Thus if using alternative units, set the unit conversion factor to to the MKS value that corresponds to unity in the alternative units.

conversionCoefficients (float vector, optional)

Custom unit conversion factors for EOS table values. Conversion to MKS units is the default. The default conversion factor is divided by the custom conversion factor. Thus if using alternative units, set the unit conversion factor to to the MKS value that corresponds to unity in the alternative units.

12.12.2 Parent Updater Data

`in` (string vector, required)

`input variables` (*nodalArray*, 1-component each, 2 required)

The specific input variables and order depend on the **operations** input option. For direct EOS evaluation, the input variables should be in = [temperature, density]. The order is critical where temperature must be the first input and density must be the second input. For inverse operations, the temperature, if an input, must be the first input and the density, if an input, must be the second input. The EOS input should be placed in the correspondingly empty input location. Inputs are of type *nodalArray* with one component each.

`out` (string vector, required)

`output variables` (*nodalArray*, 1-component each, required)

The number of out variables should be the same as the number of entries into the list of **operations**. The result of each operation will be placed into the corresponding output variable, respectively. Outputs are of type *nodalArray* with one component each.

12.12.3 Example

```
<Updater computeZavg>
  kind = equation3d
  onGrid = domain
  in =  [temperature, density]
  out = [zAvg, intEmisPlanck]

  <Equation thisGas>
    kind = propaceosVariables
    fixRanges = [1, 1]
    filename = Ar_Ni_1e^10_10group_NLTE_20110427.prp
    operations = ["computeZbarFromTemperatureAndDensity" \
                  "computeIntEmisPlanckFromTemperatureAndDensity"]
    speciesMass=MI
    elementList = [Ar]
  </Equation>
</Updater>
```

12.13 propaceosComputeVariables

This source allows the user to read in data from a PROPACEOS table and then compute pressure and the sound speed squared from density and the internal energy. PROPACEOS tables can be obtained from Prism Computational Sciences (PROPACEOS link). Alternatively the PROPACEOS format can be used to create your own tables. Tables specify an equation of state (EOS) for energy and pressure as functions of temperature and density. Thus to solve for temperature, as an intermediate step, as a function of the internal energy an inverse operation must be applied. This operation holds the input temperature or density constant and assumes the EOS table data is a monotonic function of the dependent variables (density and temperature). If these assumptions do not hold, incorrect results may be produced.

In this updater, the sound speed squared is computed from a formula for the generalized sound speed:

$$c_s^2 = \frac{\partial P}{\partial \epsilon}\frac{P}{\rho^2} + \frac{\partial P}{\partial \rho}$$

A note on units. Units in USim are all MKS units. However, the PROPACEOS tables use CGS units and eV for temperature. These units are converted to MKS by USim. This is important if one writes their own PROPACEOS tables. The ability to specify custom unit conversion factors is available as an optional input.

Before running any case using the PROPACEOS EOS tables, it is prudent to make basic sanity checks by running a modified version of the *verifyEOSTable* example with the specific PROPACEOS table that is intended for use.

12.13.1 Parameters

`filename` (string, required) Name of file that contains the PROPACEOS formatted table.

`speciesMass` (float, required) Mass of the species as required to convert from number to mass density.

`delta` (float, optional) A finite difference operation is applied to evaluate partial derivatives. This factor determines the relative width of the stencil. The default is 10^{-6}.

`soundSpeedSquaredFloor` (float, optional) Sets a minimum value for the output sound speed squared. The default is 0.

useParticleDensity (int, optional) Whether to use the particle (true) or mass (false) density. Default is false.

fixRanges (int vector, options) Whether the variables should be allowed to go beyond the table ranges or not. fixRanges = [1] means that the first variable cannot go beyond the table ranges and if it does, it's value is set to the maximum (or minimum) of the table value. The default is false.

logInterpolation (int vector, optional) Whether to use logarithmic interpolation when evaluating EOS table values. The default is false.

densityConversionCoefficient (float, optional) Custom density unit conversion factor. Conversion to MKS mass density (kgm^{-3}) is the default. The default conversion factor is divided by the custom conversion factor. Thus if using alternative units, set the unit conversion factor to to the MKS value that corresponds to unity in the alternative units.

temperatureConversionCoefficient (float, optional) Custom temperature unit conversion factor. Conversion to MKS temperature (K) is the default. The default conversion factor is divided by the custom conversion factor. Thus if using alternative units, set the unit conversion factor to to the MKS value that corresponds to unity in the alternative units.

conversionCoefficients (float vector, optional) Custom unit conversion factors for EOS table values. Conversion to MKS units is the default. The default conversion factor is divided by the custom conversion factor. Thus if using alternative units, set the unit conversion factor to to the MKS value that corresponds to unity in the alternative units.

outputPeRhoInv (int, optional) Boolean that determines if the partial derivative of the pressure with respect to specific energy divided by the density, $\rho^{-1}\partial P/\partial\epsilon$, is output. This output is required to compute the EOS system eigenvectors. The default is false.

12.13.2 Parent Updater Data

`in` (string vector, required)

`input variables` (*nodalArray*, 1-component each, 2 required)

The input variables (exactly 2) must be the density and the internal energy, in that order. Inputs are of type *nodalArray* with one component each.

`out` (string vector, required)

`output variables` (*nodalArray*, 1-component each, 2 required and 3rd optional)

The output variables are the pressure and the sound speed squared, in that order. If outputPeRhoInv is true, a third output variable that is the partial derivative of the pressure with respect to specific energy divided by the density $\rho^{-1}\partial P/\partial\epsilon$. This output is required to compute the EOS system eigenvectors. Outputs are of type *nodalArray* with one component each.

12.13.3 Example

```
<Updater computePressureAndSoundSpeedSquared>
  kind=equation2d
  onGrid=domain
  in=[rho, intEnergy]
  out=[pressure, soundSqr]
  <Equation thisGas>
    kind=propaceosComputeVariables
    filename=propaceos.prp
    delta=1.e-5
```

```
      speciesMass=MI
    </Equation>
  </Updater>
```

12.14 sesameVariables

This source allows the user to read in data from a SESAME table and then compute energy, pressure, density, temperature and conductivity. SESAME tables can be obtained from Los Alamos National Laboratory (SESAME link). Alternatively the SESAME format can be used to create your own tables. Tables specify an equation of state (EOS) for energy, pressure, conductivities and opacity as a function of temperature and density. To solve for temperature or density as a function of the EOS table value an inverse operation must be applied. This operation holds the input temperature or density constant and assumes the EOS table data is a monotonic function of the dependent variables (density and temperature). If these assumptions do not hold, incorrect results may be produced.

A note on units. Units in USim are all MKS units. However, the SESAME tables use alternative units. These units are converted to MKS by USim. This is important if one writes their own SESAME tables. The ability to specify custom unit conversion factors is available as an optional input.

Before running any case using the SESAME EOS tables, it is prudent to make basic sanity checks by running a modified version of the *verifyEOSTable* example with the specific SESAME table that is intended for use.

12.14.1 Parameters

`filename` (string, required) Name of file that contains the SESAME formatted table.

`operations` (string vector, required) The operation(s) to be performed. The standard direct operation is "computeEOSTableFromTemperatureAndDensity" where "EOSTable" is computed from the SESAME tables and should be replaced by one of the following values (the first word of each option should be used, the remainder offers a brief description and the default units after conversion):

- 301energy - (Jkg^{-1})
- 301freeenergy - (Jkg^{-1})
- 301pressure - (Pa)
- 303energy - (Jkg^{-1})
- 303freeenergy - (Jkg^{-1})
- 303pressure - (Pa)
- 304energy - (Jkg^{-1})
- 304freeenergy - (Jkg^{-1})
- 304pressure - (Pa)
- 305energy - (Jkg^{-1})
- 305freeenergy - (Jkg^{-1})
- 305pressure - (Pa)
- 306energy - (Jkg^{-1})
- 306freeenergy - (Jkg^{-1})
- 306pressure - (Pa)

- 601 - Mean Ion Charge (free electrons per atom)
- 602 - Electrical Conductivity (s^{-1})
- 603 - Thermal Conductivity ($m^{-1}s^{-1}$)
- 604 - Thermoelectric Coefficient ($m^{-1}s^{-1}$)
- 605 - Electron Conductive Opacity (m^2kg^{-1})

Tables 301, and 303-306, which are decomposed into energy, freeenergy and pressure above are described as follows:

- 301 - Total EOS (304+305+306)
- 303 - Ion EOS Plus Cold Curve (305+306)
- 304 - Electron EOS
- 305 - Ion EOS (Including Zero Point)
- 306 - Cold Curve (No Zero Point)

To compute an inverse operation, simply permute the string to be "computeTemperatureFromEOSTableAndDensity" or "computeDensityFromTemperatureAndEOSTable". The input is not case sensitive.

`materialID` (int, required) Identifying material ID in the SESAME table.

`useParticleDensity` (int, optional) Whether to use the particle (true) or mass (false) density. Default is false.

`speciesMass` (float, optional) Mass of the species (in kg) is required to convert if useParticleDensity=true.

`fixRanges` (int vector, options) Whether the variables should be allowed to go beyond the table ranges or not. fixRanges = [1] means that the first variable cannot go beyond the table ranges and if it does, it's value is set to the maximum (or minimum) of the table value. The default is false.

`logInterpolation` (int vector, optional) Whether to use logarithmic interpolation when evaluating EOS table values. The default is false.

`densityConversionCoefficient` (float, optional) Custom density unit conversion factor. Conversion to MKS mass density (kgm^{-3}) is the default. The default conversion factor is divided by the custom conversion factor. Thus if using alternative units, set the unit conversion factor to to the MKS value that corresponds to unity in the alternative units.

`temperatureConversionCoefficient` (float, optional) Custom temperature unit conversion factor. Conversion to MKS temperature (K) is the default. The default conversion factor is divided by the custom conversion factor. Thus if using alternative units, set the unit conversion factor to to the MKS value that corresponds to unity in the alternative units.

`conversionCoefficients` (float vector, optional) Custom unit conversion factors for EOS table values. Conversion to MKS units is the default. The default conversion factor is divided by the custom conversion factor. Thus if using alternative units, set the unit conversion factor to to the MKS value that corresponds to unity in the alternative units.

12.14.2 Parent Updater Data

`in` (string vector, required)

`input variables` (*nodalArray*, 1-component each, 2 required)

The specific input variables and order depend on the **operations** input option. For direct EOS evaluation, the input variables should be in = [temperature, density]. The order is critical where temperature must be the first input and density must be the second input. For inverse operations, the temperature,

if an input, must be the first input and the density, if an input, must be the second input. The EOS input should be placed in the correspondingly empty input location. Inputs are of type *nodalArray* with one component each.

`out` (string vector, required)

`output variables` (*nodalArray*, 1-component each, required)

The number of out variables should be the same as the number of entries into the list of **operations**. The result of each operation will be placed into the corresponding output variable, respectively. Outputs are of type *nodalArray* with one component each.

12.14.3 Example

```
<Updater computeEOS>
  kind=equation2d
  onGrid=domain
  in=[temperature, density]
  out=[energy, pressure]
  <Equation thisGas>
    kind=sesameVariables
    filename=sesame.ses
    materialID=58501
    operations=["compute301EnergyFromDensityAndTemperature", \
                "compute301PressureFromDensityAndTemperature"]
  </Equation>
</Updater>
```

12.15 sesameComputeVariables

This source allows the user to read in data from a SESAME table and then compute pressure and the sound speed squared from density and the internal energy. SESAME tables can be obtained from Los Alamos National Laboratory (SESAME link). Alternatively the SESAME format can be used to create your own tables. Tables specify an equation of state (EOS) for energy and pressure as functions of temperature and density. Thus to solve for temperature, as an intermediate step, as a function of the internal energy an inverse operation must be applied. This operation holds the input temperature or density constant and assumes the EOS table data is a monotonic function of the dependent variables (density and temperature). If these assumptions do not hold, incorrect results may be produced.

In this updater, the sound speed squared is computed from a formula for the generalized sound speed:

$$c_s^2 = \frac{\partial P}{\partial \epsilon}\frac{P}{\rho^2} + \frac{\partial P}{\partial \rho}$$

A note on units. Units in USim are all MKS units. However, the SESAME tables use alternative units. These units are converted to MKS by USim. This is important if one writes their own SESAME tables. The ability to specify custom unit conversion factors is available as an optional input.

Before running any case using the SESAME EOS tables, it is prudent to make basic sanity checks by running a modified version of the *verifyEOSTable* example with the specific SESAME table that is intended for use.

12.15.1 Parameters

`filename` (string, required) Name of file that contains the SESAME formatted table.

`materialID` (int, required) Identifying material ID in the SESAME table.

`delta` (float, optional) A finite difference operation is applied to evaluate partial derivatives. This factor determines the relative width of the stencil. The default is 10^{-6}.

`soundSpeedSquaredFloor` (float, optional) Sets a minimum value for the output sound speed squared. The default is 0.

`useParticleDensity` (int, optional) Whether to use the particle (true) or mass (false) density. Default is false.

`speciesMass` (float, optional) Mass of the species (in kg) is required to convert if useParticleDensity=true.

`fixRanges` (int vector, options) Whether the variables should be allowed to go beyond the table ranges or not. fixRanges = [1] means that the first variable cannot go beyond the table ranges and if it does, it's value is set to the maximum (or minimum) of the table value. The default is false.

`logInterpolation` (int vector, optional) Whether to use logarithmic interpolation when evaluating EOS table values. The default is false.

`densityConversionCoefficient` (float, optional) Custom density unit conversion factor. Conversion to MKS mass density (kgm^{-3}) is the default. The default conversion factor is divided by the custom conversion factor. Thus if using alternative units, set the unit conversion factor to to the MKS value that corresponds to unity in the alternative units.

`temperatureConversionCoefficient` (float, optional) Custom temperature unit conversion factor. Conversion to MKS temperature (K) is the default. The default conversion factor is divided by the custom conversion factor. Thus if using alternative units, set the unit conversion factor to to the MKS value that corresponds to unity in the alternative units.

`conversionCoefficients` (float vector, optional) Custom unit conversion factors for EOS table values. Conversion to MKS units is the default. The default conversion factor is divided by the custom conversion factor. Thus if using alternative units, set the unit conversion factor to to the MKS value that corresponds to unity in the alternative units.

`outputPeRhoInv` (int, optional) Boolean that determines if the partial derivative of the pressure with respect to specific energy divided by the density, $\rho^{-1}\partial P/\partial\epsilon$, is output. This output is required to compute the EOS system eigenvectors. The default is false.

12.15.2 Parent Updater Data

`in` (string vector, required)

`input variables` (*nodalArray*, 1-component each, 2 required)

The input variables (exactly 2) must be the density and the internal energy, in that order. Inputs are of type *nodalArray* with one component each.

`out` (string vector, required)

`output variables` (*nodalArray*, 1-component each, 2 required and 3rd optional)

The output variables are the pressure and the sound speed squared, in that order. If outputPeRhoInv is true, a third output variable that is the partial derivative of the pressure with respect to specific energy divided by the density $\rho^{-1}\partial P/\partial\epsilon$. This output is required to compute the EOS system eigenvectors. Outputs are of type *nodalArray* with one component each.

12.15.3 Example

```
<Updater computePressureAndSoundSpeedSquared>
  kind=equation2d
  onGrid=domain
  in=[rho, intEnergy]
  out=[pressure, soundSqr]
  <Equation thisGas>
    kind=sesameComputeVariables
    filename=sesame.ses
    materialID=58501
    delta=1.e-5
  </Equation>
</Updater>
```

12.16 vanDerWaalsVariables

This source allows the user to compute specific internal energy (ϵ), pressure (P), density (ρ), and temperature (T) from the Van Der Waals gas law,

$$P = \frac{R}{C_V}(\epsilon + \eta_a \rho)\frac{\rho}{1 - \eta_b \rho} - \eta_a \rho^2$$

Here R is the gas constant, C_V is the specific heat at constant volume and η_a and η_b are constants accounting for the intermolecular forces and the molecular size, respectively.

12.16.1 Parameters

`operations` (string vector, required) The operation(s) to be performed. The standard direct operation is "computeEOSFromTemperatureAndDensity" where "EOS" should be replaced by one of the following values (the first word of each option should be used, the remainder offers a brief description and the default units after conversion):

- energy - (Jkg^{-1})
- pressure - (Pa)

To compute an inverse operation, simply permute the string to be "computeTemperatureFromEOSAndDensity" or "computeDensityFromTemperatureAndEOS". The input is not case sensitive.

`Rr` (float, required) Specifies the gas constant, R.

`Cv` (float, required) Specifies the gas specific heat at constant volume , C_V.

`etaA` (float, required) Specifies the intermolecular force constant, η_a in units of m^3kg^{-1}.

`etaB` (float, required) Specifies the molecular size constant, η_b in units of $m^5kg^{-1}s^{-2}$.

`speciesMass` (float, required) Specifies the species mass in kg.

`kboltz` (float, optional) Specifies the Boltzmann constant. Defaults to 1.3806×10^{-23} (J/K).

`useParticleDensity` (int, optional) Whether to use the particle (true) or mass (false) density. Default is false.

`densityConversionCoefficient` (float, optional) Custom density unit conversion factor. Conversion to MKS mass density (kgm^{-3}) is the default. The default conversion factor is divided by the custom conversion factor. Thus if using alternative units, set the unit conversion factor to to the MKS value that corresponds to unity in the alternative units.

temperatureConversionCoefficient (float, optional) Custom temperature unit conversion factor. Conversion to MKS temperature (K) is the default. The default conversion factor is divided by the custom conversion factor. Thus if using alternative units, set the unit conversion factor to to the MKS value that corresponds to unity in the alternative units.

conversionCoefficients (float vector, optional) Custom unit conversion factors for EOS values. Conversion to MKS units is the default. The default conversion factor is divided by the custom conversion factor. Thus if using alternative units, set the unit conversion factor to to the MKS value that corresponds to unity in the alternative units.

12.16.2 Parent Updater Data

`in` (string vector, required)

`input variables` (*nodalArray*, 1-component each, 2 required)

The specific input variables and order depend on the **operations** input option. For direct EOS evaluation, the input variables should be in = [temperature, density]. The order is critical where temperature must be the first input and density must be the second input. For inverse operations, the temperature, if an input, must be the first input and the density, if an input, must be the second input. The EOS input should be placed in the correspondingly empty input location. Inputs are of type *nodalArray* with one component each.

`out` (string vector, required)

`output variables` (*nodalArray*, 1-component each, required)

The number of out variables should be the same as the number of entries into the list of **operations**. The result of each operation will be placed into the corresponding output variable, respectively. Outputs are of type *nodalArray* with one component each.

12.16.3 Example

```
<Updater computeEOS>
  kind=equation2d
  onGrid=domain
  in=[temperature, density]
  out=[energy, pressure]
  <Equation thisGas>
    kind=vanDerWaalsVariables
    Rr=2.0769
    Cv=3.1156
    etaA=0.0346
    etaB=0.0238
    speciesMass=6.64e-27
    operations=["computeEnergyFromDensityAndTemperature", \
                "computePressureFromDensityAndTemperature"]
  </Equation>
</Updater>
```

12.17 vanDerWaalsComputeVariables

This source allows the user to compute pressure (P) and the sound speed squared (c_s^2) from density (ρ) and the internal energy ($\rho\epsilon$) with the Van Der Waals gas law,

$$P = \frac{R}{C_V}(\epsilon + \eta_a \rho)\frac{\rho}{1 - \eta_b \rho} - \eta_a \rho^2$$

Here R is the gas constant, C_V is the specific heat at constant volume and η_a and η_b are constants accounting for the intermolecular forces and the molecular size, respectively.

In this updater, the sound speed squared is computed from a formula for the generalized sound speed:

$$c_s^2 = \frac{\partial P}{\partial \epsilon}\frac{P}{\rho^2} + \frac{\partial P}{\partial \rho}$$

12.17.1 Parameters

`Rr` (float, required) Specifies the gas constant, R.

`Cv` (float, required) Specifies the gas specific heat at constant volume , C_V.

`etaA` (float, required) Specifies the intermolecular force constant, η_a in units of $m^3 kg^{-1}$.

`etaB` (float, required) Specifies the molecular size constant, η_b in units of $m^5 kg^{-1} s^{-2}$.

`speciesMass` (float, required) Specifies the species mass in kg.

`kboltz` (float, optional) Specifies the Boltzmann constant. Defaults to 1.3806×10^{-23} (J/K).

`delta` (float, optional) A finite difference operation is applied to evaluate partial derivatives. This factor determines the relative width of the stencil. The default is 10^{-6}.

`soundSpeedSquaredFloor` (float, optional) Sets a minimum value for the output sound speed squared. The default is 0.

`useParticleDensity` (int, optional) Whether to use the particle (true) or mass (false) density. Default is false.

`densityConversionCoefficient` (float, optional) Custom density unit conversion factor. Conversion to MKS mass density (kgm^{-3}) is the default. The default conversion factor is divided by the custom conversion factor. Thus if using alternative units, set the unit conversion factor to to the MKS value that corresponds to unity in the alternative units.

`temperatureConversionCoefficient` (float, optional) Custom temperature unit conversion factor. Conversion to MKS temperature (K) is the default. The default conversion factor is divided by the custom conversion factor. Thus if using alternative units, set the unit conversion factor to to the MKS value that corresponds to unity in the alternative units.

`conversionCoefficients` (float vector, optional) Custom unit conversion factors for EOS values. Conversion to MKS units is the default. The default conversion factor is divided by the custom conversion factor. Thus if using alternative units, set the unit conversion factor to to the MKS value that corresponds to unity in the alternative units.

`outputPeRhoInv` (int, optional) Boolean that determines if the partial derivative of the pressure with respect to specific energy divided by the density, $\rho^{-1}\partial P/\partial \epsilon$, is output. This output is required to compute the EOS system eigenvectors. The default is false.

12.17.2 Parent Updater Data

`in` (string vector, required)

`input variables` (*nodalArray*, 1-component each, 2 required)

The input variables (exactly 2) must be the density and the internal energy, in that order. Inputs are of type *nodalArray* with one component each.

`out` (string vector, required)

`output variables` (*nodalArray*, 1-component each, 2 required and 3rd optional)

The output variables are the pressure and the sound speed squared, in that order. If outputPeRhoInv is true, a third output variable that is the partial derivative of the pressure with respect to specific energy divided by the density $\rho^{-1}\partial P/\partial\epsilon$. This output is required to compute the EOS system eigenvectors. Outputs are of type *nodalArray* with one component each.

12.17.3 Example

```
<Updater computePressureAndSoundSpeedSquared>
  kind=equation2d
  onGrid=domain
  in=[rho, intEnergy]
  out=[pressure, soundSqr]
  <Equation thisGas>
    kind=vanDerWaalsComputeVariables
    delta=1.e-5
    Rr=2.0769
    Cv=3.1156
    etaA=0.0346
    etaB=0.0238
    speciesMass=6.64e-27
  </Equation>
</Updater>
```

The following kinds can be used to couple fluid models with radiation models:

12.18 bremsPowerSrc

Computes the Bremsstrahlung the power density loss term.

$$s = \left(\frac{n_e}{7.69e18}\right)^2 T_{eV}^{1/2} Z_{eff}$$

Where n_e is the electron number density in $1/m^3$, T_{eV} is electron temperature in electron volts (note that the temperature is input in Kelvin), Z_{eff} is the effective ion charge state.

12.18.1 Parent Updater Data

`in` (string vector, required) 1st Variable

0. n number density

2nd Variable (1 component)

0. T_{eV} electron temperature in Kelvin

3rd Variable (1 component)

0. Z Z effective

12.18.2 Example

```
<Source radiationSource>
  kind = bremsPowerSrc
</Source>
```

12.19 radiationAbsorption

Computes the absorbed power for each radiation group given the absorption coefficient and ion number density

12.19.1 Parameters

`ionMass` (float) The mass of the ion species

`numberOfGroup` (int) The number of groups that should be considered

12.19.2 Parent Updater Data

`in` (string vector, required) 1st Variable

The ion number density for the species $1/m^3$

2nd Variable

The group radiation energy density J/m^3 for each group, each component represents a different frequency group

3rd Variable

The absorption coefficient m^2/Kg for each group, each component represents a different frequency group

`out` (string vector, required) The output is the absorbed power density in W/m^3

12.19.3 Example

```
<Updater name>
  kind = equation1d
  onGrid = domain
  in =  [density, radiationEnergy, intAbsPlanck]
  out = [absorbedPower]

  <Equation thisGas>
    kind = radiationAbsorption
    numberOfGroups = 1
    ionMass = MI
  </Equation>
</Updater>
```

12.20 radiationEmission

Computes the the radiated power for a plasma given ion mass density, temperature and the emission coefficient.

$$C_g = 6.49394\,8\,\pi\,k_b^4/\left(c^2\,h^3\right) \tag{12.-17}$$

and the radiated power given by

$$P_g = C_g\,\sigma_P T_e^4 n_i m_i \tag{12.-17}$$

12.20.1 Parameters

`ionMass` (float) The mass of the ion species

`numberOfGroup` (int) The number of groups that should be considered. For now numberOfGroups=1.

12.20.2 Parent Updater Data

`in` (string vector, required) 1st Variable

The ion number density for the species $1/m^3$

2nd Variable

The temperature is in Kelvin

3rd Variable

The planck emission coefficient m^2/Kg

`out` (string vector, required) The output is the radiated power density in W/m^3

12.20.3 Example

```
<Updater emission>
  kind = equation1d
  onGrid = grid
  in =  [density, temperature, intEmisPlanck]
  out = [radiationPower]

  <Equation thisGas>
    kind = radiationEmission
    numberOfGroups = 1
    ionMass = MI
  </Equation>
</Updater>
```

The following kinds can be used to couple fluid systems with electromagnetic systems:

12.21 coilFieldEqn

Computes the analytic magnetic field from a single coil

12.21.1 Parameters

`center` (vector float) center of the coil

`current` (float) current in the coil

`mu0` (float) permeability of free space

`normal` (vector float) normal to the plane of the coil

`radius` (float) radius of the coil

12.21.2 Example

```
<Source coilSource>
  kind = coilFieldEqn
  mu0 = 1.26e-6
  center = [0.5, 0.5, 0.0]
  normal = [1.0, 0.0, 0.0]
  radius = 10.0
  current = 1.0
</Source>
```

12.22 current

Computes the fluid "current" given from fluid variables, particle mass, charge and permittivity. This current would be used as a source term for Maxwell's equations.

$$s = -\frac{1}{\epsilon_0}\frac{q}{m}\begin{pmatrix} \rho\, u_x \\ \rho\, u_y \\ \rho\, u_z \end{pmatrix}$$

where q is the species charge, m is the species mass ϵ_0 is the permittivity, ρ is the fluid mass density, u_x is the fluid x velocity, u_y is the fluid y velocity and u_z is the fluid z velocity.

12.22.1 Parameters

`epsilon0` (float) permittivity of free space

`mass` (float) The mass of the fluid particles

`charge` (float) The charge of the fluid particles

`startIndex` (integer) Tells USim which variable in the input vector should be set to the zero position. For example, if you pass in q from the eulerEqn then *startIndex* would be 1 as the momentum density terms correspond to indexes 1, 2, 3. In that case the 0 index corresponds to mass density. The default value for startIndex is 0.

12.22.2 Parent Updater Data

`in` (string vector, required) 1st Variable

0. $\rho\, u_x$ x momentum density
1. $\rho\, u_y$ y momentum density

2. $\rho\, u_z$ z momentum density

12.22.3 Example

```
<Source ionCurrents>
  kind = current
  startIndex = 1
  charge = ION_CHARGE
  mass = ION_MASS
  epsilon0 = 1.0
</Source>
```

12.23 lorentzForce

Computes the lorentz force given from fluid variables, particle mass, charge and permittivity. This lorentz force would be used as a source term for fluid equations.

$$s = \rho \frac{q}{m} \begin{pmatrix} 0 \\ E_x + u_y B_z - u_z B_y \\ E_y + u_z B_x - u_x B_z \\ E_z + u_x B_y - u_y B_x \\ u_x\, E_x + u_y\, E_y + u_z\, E_z \end{pmatrix}$$

where q is the species charge, m is the species mass ϵ_0 is the permittivity, ρ is the fluid mass density, u_x is the fluid x velocity, u_y is the fluid y velocity, u_z is the fluid z velocity, E_x is the x electric field, E_y is the y electric field, E_z is the z electric field, B_x is the x magnetic field, B_y is the y magnetic field and B_z is the z magnetic field.

In the case where the user wants the Lorentz term for the two-fluid form *twoFluidEqn* the source is written as

$$s = \begin{pmatrix} 0 \\ \rho_c E_x + j_y B_z - j_z B_y \\ \rho_c E_y + j_z B_x - j_x B_z \\ \rho_c E_z + j_x B_y - j_y B_x \\ 0 \\ (r_i^2 \rho_i + r_e^2 \rho_e) E_x + (r_i^2 \rho_i u_{y\,i} + r_e^2 \rho_e u_{y\,e}) B_z - (r_i^2 \rho_i u_{z\,i} + r_e^2 \rho_e u_{z\,e}) B_y \\ (r_i^2 \rho_i + r_e^2 \rho_e) E_y + (r_i^2 \rho_i u_{z\,i} + r_e^2 \rho_e u_{z\,e}) B_x - (r_i^2 \rho_i u_{x\,i} + r_e^2 \rho_e u_{x\,e}) B_z \\ (r_i^2 \rho_i + r_e^2 \rho_e) E_z + (r_i^2 \rho_i u_{x\,i} + r_e^2 \rho_e u_{x\,e}) B_y - (r_i^2 \rho_i u_{y\,i} + r_e^2 \rho_e u_{y\,e}) B_x \\ j_{x\,i}\, E_x + j_{y\,i}\, E_y + j_{z\,i}\, E_z \\ j_{x\,e}\, E_x + j_{y\,e}\, E_y + j_{z\,e}\, E_z \end{pmatrix}$$

and this source can be chosen by choosing type=twoFluidEqn. The variables are defined as follows, $r_i = q_i/m_i$ and $r_e = q_e/m_e$ where q_e is the electron charge, q_i is the ion charge, m_e is the electron mass and m_i is the ion mass. In addition the variables $(\rho_\alpha, u_{x\,\alpha}, u_{y\,\alpha}, u_{x\,\alpha})$ are the species mass density, species x velocity, species y velocity, and species z velocity. In this case α represents the species, either e for electron or i for ion. In addition (j_x, j_y, j_z) are the total current densities in the x, y and z directions.

12.23.1 Parameters common to all systems

`type` (string) The type of source is *split5* (the default), or *twoFluidEqn*

12.23.2 Parameters (type=split5)

`mass` (float) The mass of the fluid species

`charge` (float) The charge of the fluid species

12.23.3 Parameters (type=twoFluidEqn)

`electronMass` (float) The electron mass

`ionMass` (float) The ion mass

`electronCharge` (float) The electron charge

`ionCharge` (float) The ion charge

12.23.4 Parent Updater Data (type=split5) Default

`in` (string vector, required) 1st Variable

0. ρ mass density
1. $\rho\, u_x$ x momentum density
2. $\rho\, u_y$ y momentum density
3. $\rho\, u_z$ z momentum density

2nd Variable

0. e_x x electric field
1. e_y y electric field
2. e_z z electric field
3. b_x x magnetic field
4. b_y y magnetic field
5. b_z z magnetic field

`out` (string vector, required) The output variable is a length 5 vector, but the first component is 0 so that it works simply as a fluid source for the euler equations.

1st Variable

0. 0.0 mass density. No contribution from Lorentz force
1. L_x x momentum density contribution of Lorentz force
2. L_y y momentum density contribution of Lorentz force
3. L_z z momentum density contribution of Lorentz force
4. $E \cdot J$ energy density contribution of Lorentz force

12.23.5 Parent Updater Data (type=twoFluidEqn)

`in` (string vector, required) 1st Variable

0. ρ mass density

1. $\rho\, u_x$ x momentum density
2. $\rho\, u_y$ y momentum density
3. $\rho\, u_z$ z momentum density
4. ρ_c total charge density
5. j_x x current density
6. j_y y current density
7. j_z z current density
8. e_i ion energy density
9. e_e electron energy density

2nd Variable

0. e_x x electric field
1. e_y y electric field
2. e_z z electric field
3. b_x x magnetic field
4. b_y y magnetic field
5. b_z z magnetic field

12.23.6 Example

```
<Source lorentzIon>
  kind = lorentzForce
  mass = ION_MASS
  charge = ION_CHARGE
</Source>

<Source lorentz>
  kind = lorentzForce
  type = twoFluidEqn
  ionMass = ION_MASS
  electronMass = ELECTRON_MASS
  ionCharge = ION_CHARGE
  electronCharge = ELECTRON_CHARGE
</Source>
```

12.24 wireFieldEqn

Computes the analytic magnetic field from a single wire

12.24.1 Parameters

`point` (vector float) a point that the wire passes through

`current` (float) current in the wire

`mu0` (float) permeability of free space

normal (vector float) direction of the wire through the point

12.24.2 Example

```
<Source coilSource>
  kind = wireFieldEqn
  point = [0.0, 0.0, 1.0]
  mu0 = 1.26e-6
  normal = [1.0, 0.0, 0.0]
  current = 1.0
</Source>
```

The following kinds can be used to control divergence errors in electromagnetic problems:

12.25 computeChargeError

Computes the simulation charge error which is measured as the difference in charge as computed from the divergence of the electric field and that computed using the continuity equation. This source does not compute $\nabla \cdot E$, instead this value is passed in.

$$\delta = \nabla \cdot E - \frac{1}{\epsilon_0} \sum_i \frac{q_i}{m_i} \rho_i$$

where E is the electric field, m_i is the species mass, q_i is the species charge, ϵ_0 is the permittivity, ρ_i is the species mass density.

12.25.1 Parameters

speciesCharge (vector float) Species charge in the order they appear in the input list.

speciesMass (vector float) The charge of the fluid species.

epsilon0 (float) The permittivity of free space.

12.25.2 Parent Updater Data

in (string vector, required) 1st Variable

0. $\nabla \cdot E$ The divergence of the electric field

Remaining variables

This source takes an arbitrary number of species mass variables, but requires at least 1.

0. ρ species mass density

out (string vector, required) The output is the charge error

1st Variable

0. δ The charge error.

12.25.3 Example

```
<Equation>
  kind = computeChargeError
  speciesCharge = [ELECTRON_CHARGE, ION_CHARGE]
  speciesMass = [ELECTRON_MASS, ION_MASS]
  epsilon0 = EPSILON0
</Equation>
```

12.26 hyperbolicCleanSym

Provides the axisymmetric symmetry source terms for the hyperbolicCleanEqn

$$s = -\frac{1}{r}\begin{pmatrix} 0 \\ 0 \\ 0 \\ \gamma^2 B_x \end{pmatrix}$$

where γ is the correction wave speed.

12.26.1 Parameters

`waveSpeed` (float) Correction wave speed

12.26.2 Parent Updater Data

`in` (string vector, required) 4 primary variables

0. B_x x magnetic field
1. B_y y magnetic field
2. B_z z magnetic field
3. Ψ correction potential

12.26.3 Example

```
<Source symSource>
  kind = hyperbolicCleanSym
  symmetryType = cylindrical
  waveSpeed = CORRECTIONSPEED
</Source>
```

The following kinds can be used for coupling together multi-species fluid models:

12.27 collisionFrequency

Computes the collision frequency matrix for multiple fluids species or the collision time matrix if the inverse quantities are stored. The two approaches used for collisions in fully ionized plasma are *thermalSpecies* which

ignores the relative drift of the fluids and *ramboAndDenavit* which takes into account the relative drift. thermalSpecies is identical to *ramboAndDenavit* with the velocities set to 0. A description of this collision model is described in

Rambo, P. W., and J. Denavit. "Interpenetration and ion separation in colliding plasmas." Physics of Plasmas 1 (1994): 4050.

For *neutrals* collisions, the collision cross section is obtained using hard sphere model. The relative velocity includes thermal and bulk velocities. Collisions in *partiallyIonized* plasmas use *ramboAndDenavit* if both the colliding particles are charged and uses *neutrals* otherwise.

12.27.1 Parameters

`type` (string) *type* should be either *thermalSpecies* or *ramboAndDenavit* or *neutrals* or *partiallyIonized*. The *thermalSpecies* is the classical collision frequency assuming zero relative velocity between the fluids in consideration. *ramboAndDenavit* assumes that there may be a large relative velocity between species. *neutrals* uses the hard sphere model to compute collision cross section and the relative velocity includes both thermal and bulk velocitis.

`speciesMass` (vector float) The mass of each fluid species

`speciesDia` (vector float) The diameter of each fluid species

`inverse` (boolean) If *inverse* is false the the collision frequency is computed, if *inverse* is true then the collision time is computed.

12.27.2 Parent Updater Data (type = *thermalSpecies*)

`in` (string vector, required) Each species has a *Z*, *T* and *N* variable that must be put into the in variable, so for 2 species in would be

in = [Z1, T1, N1, Z2, T2, N2]

1st Variable

0. *Z* Is the charge state of the species (positive value)

2nd Variable

0. *T* Is the temperature of the species

3rd Variable

0. *N* Is the number density of the species

`out` (string vector, required) The output is the collision matrix. The size of the matrix will be *numSpecies*numSpecies* where here numSpecies is the number of components in the *speciesMass* vector below.

12.27.3 Parent Updater Data (type = *ramboAndDenavit*)

`in` (string vector, required) Each species has a *Z*, *T*, *N*, *V* variable that must be put into the in variable, so for 2 species in would be

in = [Z1, T1, N1, V1, Z2, T2, N2, V2]

1st Variable

0. *Z* Is the charge state of the species (positive value)

2nd Variable

0. *T* Is the temperature of the species

3rd Variable

0. *N* Is the number density of the species

4th Variable

0. *Vx* Is the velocity of the fluid in the X direction
1. *Vy* Is the velocity of the fluid in the Y direction
2. *Vz* Is the velocity of the fluid in the Z direction

out (string vector, required) The output is the collision matrix. The size of the matrix will be *numSpecies*numSpecies* where here numSpecies is the number of components in the *speciesMass* vector below.

12.27.4 Parent Updater Data (type = *neutrals*)

in (string vector, required) Each species has a *T*, *N*, *V* variable that must be put into the in variable, so for 2 species in would be

in = [T1, N1, V1, T2, N2, V2]

1st Variable

0. *T* Is the temperature of the species

2nd Variable

0. *N* Is the number density of the species

3rd Variable

0. *Vx* Is the velocity of the fluid in the X direction
1. *Vy* Is the velocity of the fluid in the Y direction
2. *Vz* Is the velocity of the fluid in the Z direction

out (string vector, required) The output is the collision matrix. The size of the matrix will be *numSpecies*numSpecies* where here numSpecies is the number of components in the *speciesMass* vector below.

12.27.5 Parent Updater Data (type = *partiallylonized*)

in (string vector, required) Each species has a *Z*, *T*, *N*, *V* variable that must be put into the in variable, so for 2 species in would be

in = [Z1, T1, N1, V1, Z2, T2, N2, V2]

1st Variable

0. *Z* Is the charge state of the species (positive value)

2nd Variable

0. *T* Is the temperature of the species

3rd Variable

0. *N* Is the number density of the species

4th Variable

0. *Vx* Is the velocity of the fluid in the X direction
1. *Vy* Is the velocity of the fluid in the Y direction
2. *Vz* Is the velocity of the fluid in the Z direction

out (string vector, required) The output is the collision matrix. The size of the matrix will be *numSpecies*numSpecies* where here numSpecies is the number of components in the *speciesMass* vector below.

12.27.6 Example

```
<Equation thisGas>
  inverse = false
  kind = collisionFrequency
  type = ramboAndDenavit
  speciesMass = [ELECTRON_MASS, ION_MASS, ION_MASS]
</Equation>
```

12.28 conductivityTensor

Specify a tensor that has different conductivity parallel and perpendicular to a given vector field. The Tensor is specified as

given an input vector field B that same field is used to generate a unit vector field b that is then used to define the tensor. Conductivity parallel to the magnetic field is specified as $K_{\parallel}$ and perpendicular to the vector field as $K_{\perp}$ and then the difference in conductivities $dK = K_{\parallel} - K_{\perp}$. The 9 tensor components are given as

$$\begin{pmatrix} K_{\perp} + dK\, b_x^2 & K_{\parallel} b_x b_y & K_{\parallel} b_x b_z \\ K_{\parallel}\, b_x b_y & K_{\perp} + dK\, b_y^2 & K_{\parallel} b_y b_z \\ K_{\parallel}\, b_x b_z & K_{\parallel} b_y b_z & K_{\perp} + dK\, b_z^2 \end{pmatrix}$$

12.28.1 Parent Updater Data

in (string vector, required) 1st Variable

0. V_x x vector component
1. V_y y vector component
2. V_z z vector component

2nd Variable

0. $K_{\parallel}$ parallel conductivity

3rd Variable

0. $K_{\perp}$ perpendicular conductivity

out (string vector, required) The output variable is a length 9 vector containing the 9 components of the conductivity tensor

1st Variable

0. T_{xx}

1. T_{xy}
2. T_{xz}
3. T_{yx}
4. T_{yy}
5. T_{yz}
6. T_{zx}
7. T_{zy}
8. T_{zz}

12.28.2 Example

```
 <Updater initConductivityTensor>
  kind = equation2d
  onGrid = domain

  in = [B, kParallel, kPerpendicular]

  out = [conductivityTensor]

  <Equation a>
    kind = conductivityTensor
  </Equation>
</Updater>
```

12.29 momentumEnergyExchange

Computes the momentum and energy exchange between multiple fluids due to 'friction'. The momentum and energy exchange terms are given by the RHS of the euler equations below. Note that this does NOT include thermal relaxation as that is part of the *temperatureRelaxation* source.

The source for the continuity equation is zero, but added for convenience.

$$\frac{\partial \rho_i}{\partial t} + \nabla \cdot [\rho_i U_i] = 0 \tag{12.-21}$$

The momentum term contains the species exchange term R_i

$$\frac{\partial \rho_i U_i}{\partial t} + \nabla \cdot [\rho U_i U_i + P_i] = R_i \tag{12.-21}$$

And the energy term has a source due to changes in momentum $V \cdot R_i$

$$\frac{\partial e_i}{\partial t} + \nabla \cdot [U_i \cdot (e_i + P_i)] = V \cdot R_i \tag{12.-21}$$

Where V is the bulk velocity given by

$$V = \frac{\sum_i \rho_i U_i}{\sum_i \rho_i} \tag{12.-21}$$

and the momentum exchange term as

$$R_i = -\sum_j n_i \mu_{ij} \tau_{ij}^{-1} (U_i - U_j) \tag{12.-21}$$

Descriptions of this model can be found in

Zhdanov, Viktor Mikhailovich. "Transport processes in multicomponent plasma." Plasma Physics and Controlled Fusion 44.10 (2002): 2283.

12.29.1 Parameters

`speciesMass` (vector float) The particle mass of each fluid species

12.29.2 Parent Updater Data

`in` (string vector, required) 1st Variable

0. ρ mass density
1. $\rho\, u_x$ x momentum density
2. $\rho\, u_y$ y momentum density
3. $\rho\, u_z$ z momentum density
4. e total energy density, fluid and field

2nd Variable

0. ρ mass density
1. $\rho\, u_x$ x momentum density
2. $\rho\, u_y$ y momentum density
3. $\rho\, u_z$ z momentum density
4. e total energy density, fluid and field

Nth Variable

0. ρ mass density
1. $\rho\, u_x$ x momentum density
2. $\rho\, u_y$ y momentum density
3. $\rho\, u_z$ z momentum density
4. e total energy density, fluid and field

(N+1)th Variable

This variable is the collision frequency matrix that can be computed by the source *collisionFrequency*. The order of species should be the same as provided to *collisionFrequency*.

`out` (string vector, required) There are N outputs each at least length 5 corresponding to the source terms for the 1st through Nth inputs. The first component (corresponding to mass density) is always 0 while the remaining 4 components have non-zero values.

12.29.3 Example

```
<Equation thisGas>
  kind = momentumEnergyExchange
  speciesMass = [ELECTRON_MASS, ION_MASS, ION_MASS]
</Equation>
```

12.30 NFluidSrc

Applies the implicit source operator to the 5 moment N-fluid (ion, electron, EM) system. This operator should only really be applied to charged species as all the source terms are zero for neutral species so it can result in an excessively large matrix if neutral species are included. The approach is described for the two-fluid system in,

Kumar, Harish, and Siddhartha Mishra. "Entropy Stable Numerical Schemes for Two-Fluid Plasma Equations." Journal of Scientific Computing 52.2 (2012): 401-425.

The algorithm in USim though can be applied to an arbitrary number of charged species (anywhere from 1 to N species!). A two-fluid version of this source is also in USIM *twoFluidSrc* and should be used when only two charged species are required since the algorithm will be slightly faster.

12.30.1 Parameters (All types)

`type` string Specifies the type of implicit matrix. Options are 5Moment for the 5 moment two-fluid system.

12.30.2 Parameters (5Moment)

`speciesCharge` (float vector) List of species charges in the same order as the species in variables. There should be the same number of charges as there are species.

`speciesMass` (float) List of species masses in the same order as the species in variables. There should be the same number of masses as there are species.

`epsilon0` Permittivity of free space

12.30.3 Parent Updater Data (5Moment)

`in` (string vector, required) 1st Variable

0. ρ electron mass density
1. $\rho\, u_x$ electron x momentum density
2. $\rho\, u_y$ electron y momentum density
3. $\rho\, u_z$ electron z momentum density
4. e electron energy density

2nd Variable

0. ρ ion mass density
1. $\rho\, u_x$ ion x momentum density
2. $\rho\, u_y$ ion y momentum density
3. $\rho\, u_z$ ion z momentum density
4. e ion energy density

Nth Variable

0. ρ ion mass density
1. $\rho\, u_x$ ion x momentum density
2. $\rho\, u_y$ ion y momentum density

3. $\rho\, u_z$ ion z momentum density
4. e ion energy density

(N+1)th Variable

0. E_x x electric field
1. E_y y electric field
2. E_z z electric field
3. B_x x magnetic field
4. B_y y magnetic field
5. B_z z magnetic field
6. Ψ_E electric field correction potential
7. Ψ_B magnetic field correction potential

`out` (string vector, required) In all cases the output is Q^{n+1}. For the 5 moment system there are N+1 outputs corresponding to each of the fluids (the same order as the input) and em field (in that order).

12.30.4 Example

```
<Updater NFluidLorentz>
   kind = equation1d

   onGrid = domain
   in =  [electronsNew, ionsNew, emNew]
   out = [electronsNew, ionsNew, emNew]
   #operation = add

  <Equation fluidLorentz>
    kind = NFluidSrc
    type = 5Moment
    speciesCharge = [ELECTRON_CHARGE, ION_CHARGE]
    speciesMass = [ELECTRON_MASS, ION_MASS]
    epsilon0 = 1.0
  </Equation>
</Updater>
```

12.31 reactionTableRhs

Computes the right hand side of the reaction rate equation and the reaction energy change rate. This can then be used in a time integration scheme. Any number of reactions can be added for a given set of species.

$$\frac{dn_i}{dt} = s_i$$

$$\frac{de}{dt} = s_{re}$$

12.31.1 Parameters

species (string vector) List of species to include in the reactions.

fileName Input file containing the reaction rate constants data (*REACTIONS*), specific heats data (*CP*) and energy of formation (*EOF*). Refer to SpeciesDataFile for the input data format.

maxRate user specified value to limit the maximum rate of reactions.

outputEnergyRate option to specify whether to compute reaction energy using specific heats. NOTE: if the option is *false*, then *in* vector requires only first and second variable as inputs. *out* vector requires only first variable as input.

12.31.2 Parent Updater Data

in (string vector, required) 1st Variable

number densities of species m^{-3}

2nd Variable

average temperature of the fluid K

3rd Variable

specific heat at constant pressure of the species $\frac{J}{kgK}$

out (string vector, required) 1st Variable

time rate of change of species density $\frac{1}{m^3 s}$

2nd Variable

time rate of change of energy $\frac{J}{s}$

12.31.3 Example

```
<Updater sourceUpdater>
  kind = equation1d
  onGrid = domain

  in = [speciesDensity, temperature, specificHeat]
  out = [speciesDensitySource, reactionEnergySource]

  equations = [reactionSource]

   <Equation reactionSource>
     kind = reactionTableRhs
     species = [N2, N, O2, O, NO, NO_p1, e]
     fileName = air7Species.txt
     maxRate = 1.0e28
     outputEnergyRate = 1
  </Equation>

</Updater>
```

12.32 temperatureRelaxation

Computes the relaxation of temperature between separate fluid species due to collisions. The term Q_i below is computed in this source and stored for each input species

The relaxation term is typically added to the energy equation below

$$\frac{\partial e_i}{\partial t} + \nabla \cdot [U_i \cdot (e_i + P_i)] = Q_i \tag{12.-22}$$

and has the form

$$Q_i = -\sum_j 3\,k\,n_i \left(\frac{\mu_{ij}}{m_i + m_j}\right) \tau_{ij}^{-1} \left(T_i - T_j\right) \tag{12.-22}$$

Descriptions of this model can be found in

Zhdanov, Viktor Mikhailovich. "Transport processes in multicomponent plasma." Plasma Physics and Controlled Fusion 44.10 (2002): 2283.

12.32.1 Parameters

`speciesMass` (vector float) The particle mass of each fluid species

`isNumberDensity` (boolean) True if the densities being passed in are number densities, false if they are mass densities

12.32.2 Parent Updater Data

`in` (string vector, required) 1st Variable

0. ρ mass density or number density of the first species

2nd Variable

0. ρ mass density or number density of the second species

Nth Variable

0. ρ mass density or number density of the nth species

(N+1)th Variable

0. T temperature of the first species

(N+2)th Variable

0. T temperature of the second species

(N+N)th Variable

0. T temperature of the Nth species

(2N+1)th Variable

This variable is the collision frequency matrix that can be computed by the source *collisionFrequency*. The order of species should be the same as provided to *collisionFrequency*.

`out` (string vector, required) There are N outputs each of length 1 corresponding to the energy exchange source term for the 1st through Nth inputs.

12.32.3 Example

```
<Equation thisGas>
  kind = temperatureRelaxation
  speciesMass = [ELECTRON_MASS, ION_MASS, ION_MASS]
</Equation>
```

12.33 transportCoeffSrc

Depending on the value of `coeff`, the `transportCoeffSrc` kind of `Equation` can have different outcomes.

coeff =

- *millikanWhiteParkVibTransRelaxationTime*
- *mWpAverageVtRelaxationTime*
- *binaryDiffusionCoeff*
- *chemicalEnergy*
- *tempAvgSpecicHeatCp*
- *massFractionAvg*
- *moleFractionAvg*
- *molecularWeightAvg*

12.33.1 millikanWhiteParkVibTransRelaxationTime

Average relaxtion time for vibration-translation mode of energy of species "l" in a gas mixture

$$\tau_l = \frac{\sum_m x_m}{\sum_{l,m} x_m/\tau_{l,m}}$$

$\tau_{l,m}$ is obtained using Millikan-White curvefit as follows

$\tau_{l,m} = \frac{1}{P/101325} exp\left[A_{l,m}\left(T^{-1/3} - B_{l,m}\right) - 18.42\right]$
and
$A_{l,m} = 0.00116\mu_{l,m}^{1/2}\theta_{v,l}^{4/3}$
$B_{l,m} = 0.015\mu_{l,m}^{1/4}$
$\mu_{l,m} = \frac{M_l M_m}{M_l + M_m}$

Definitions

x_m mole fraction of species *m*

$\tau_{l,m}$ relaxation time of vibration-translation energy between specis *l* and *m*

$\theta_{v,l}$ characteristic temperature of vibration of species *l* (parameter `thetaS`)

M_m molecular weight of species *m* (parameter `molecularWeight`)

T translational temperature

P pressure of gas

Parameters

`kind` (string): transportCoeffSrc (fixed)

`coeff` (string): millikanWhiteParkVibTransRelaxationTime (fixed)

`numSpecies` (int): number of species

`molecularWeight` (vector): molecular weight of species m

`thetaS` (vector): characteristic temperature of vibration of species l. This is a material dependent constant that must be user supplied.

`function` (string): millikanWhiteParkVibTransRelaxationTime (fixed)

Parent Updater Data

`in` (string vector, required) 1st In Variable

average temperature of the species T

2nd In Variable

pressure of gas P

3rd In Variable

number density of the species n

`out` (string vector, required) the average relaxtion time for vibration-translation mode of energy of species "l" in a gas mixture

Example

```
<Equation vTrelaxationTime>
  kind = transportCoeffSrc
  coeff = millikanWhiteParkVibTransRelaxationTime
  numSpecies = 7
  molecularWeight = [M1 M2 M3 M4 M5 M6 M7]
  thetaS = [th1 th2 th3 th4 th5 th6 th7]
  function = millikanWhiteParkVibTransRelaxationTime
</Equation>
```

12.33.2 mWpAverageVtRelaxationTime

Average relaxtion time for vibration-traslation mode of energy of a gas mixture

$$\tau = \frac{\sum_l x_l}{\sum_l x_l/\tau_l}$$

Definitions

x_l mole fraction of species l

τ_l relaxtion time for vibration-translation mode of energy of species "l" in a gas mixture as calculated in coeff `millikanWhiteParkVibTransRelaxationTime`

Parameters

kind (string): transportCoeffSrc (fixed)

coeff (string): mWpAverageVtRelaxationTime (fixed)

numSpecies (int): number of species

molecularWeight (vector): molecular weight of species *m*

thetaS (vector): characteristic temperature of vibration of species *l*. This is a material dependent constant that must be user supplied.

function (string): mWpAverageVtRelaxationTime (fixed)

Parent Updater Data

in (string vector, required) 1st In Variable

average temperature of the species T

2nd In Variable

pressure of gas P

3rd In Variable

number density of the species n

out (string vector, required) the average relaxtion time for vibration-traslation mode of energy of a gas mixture

Example

```
<Equation vTrelaxationTime>
  kind = transportCoeffSrc
  coeff = mWpAverageVtRelaxationTime
  numSpecies = 7
  molecularWeight = [M1 M2 M3 M4 M5 M6 M7]
  thetaS = [th1 th2 th3 th4 th5 th6 th7]
  function = mWpAverageVtRelaxationTime
</Equation>
```

12.33.3 binaryDiffusionCoeff

Average mass diffusion coefficient of species "l" in a gas mixture

$$D_l = \frac{\sum_m x_m}{\sum_{l,m} x_m / D_{l,m}}$$

$D_{l,m}$ is obtained using hard sphere model

$$D_{l,m} = \frac{2.63\times -7}{(P/(101325)\sigma_{l,m})} \left(\frac{T^3(M_l+M_m)}{2.0 M_l M_m}\right)^{1/2}$$

Definitions

x_m mole fraction of species *m*

M_m molecular weight of species *m*

T translational temperature

P pressure of gas

$\sigma_{l,m}$ collision diameter between species *l* and *m*

Parameters

kind (string): transportCoeffSrc (fixed)

coeff (string): binaryDiffusionCoeff (fixed)

numSpecies (int): number of species

molecularWeight (vector): molecular weight of species *m*

molecularDia (vector): molecular diameter of species *m*

function (string): binaryDiffusionCoeff (fixed)

Parent Updater Data

in (string vector, required) **1st In Variable**

average temperature of the species T

2nd In Variable

pressure of gas P

3rd In Variable

number density of the species n

out (string vector, required) the average mass diffusion coefficient of species "l" in a gas mixture

Example

```
<Equation diffusionCoeff>
  kind = transportCoeffSrc
  coeff = binaryDiffusionCoeff
  numSpecies = 7
  molecularWeight = [M1 M2 M3 M4 M5 M6 M7]
  molecularDia = [d1 d2 d3 d4 d5 d6 d7]
  function = binaryDiffusionCoeff
</Equation>
```

12.33.4 chemicalEnergy

Total energy of formation of a mixture

Parameters

kind (string): transportCoeffSrc (fixed)

coeff (string): chemicalEnergy (fixed)

numSpecies (int): number of species

fileName (string): name of the SpeciesDataFile containing the energy of formation data.

Parent Updater Data

in (string vector, required) 1st In Variable

number densities of the species $1/m^3$

2nd In Variable

specific heat at constant pressure of the species $J/(kgK)$

3rd In Variable

average temperature of the species K

out (string vector, required) the energy of formation in J/m^3

Example

```
<Updater computeChemEn>
  kind = equation2d
  onGrid = domain

  in = [speciesDens,cpR,temperature]
  out = [chemEn]

  <Equation cp>
    kind = transportCoeffSrc
    coeff = chemicalEnergy
    numSpecies = NSPECIES
    fileName = REACTIONS_ATOMIC_DATA
  </Equation>
</Updater>
```

12.33.5 tempAvgSpecicHeatCp

Specific heat at constant pressure

Parameters

kind (string): transportCoeffSrc (fixed)

coeff (string): tempAvgSpecicHeatCp (fixed)

numSpecies (int): number of species

fileName (string): name of the SpeciesDataFile containing the cp data.

cpType (string): currently allowed option is *kineticTheory*, which requires molecular data specified in the SpeciesDataFile. Defaults to Shomate polynomial type specificific heat, whch agian required polynomial data specified in the SpeciesDataFile. In case of using using Shomate polynomial, addition parameters *lower*, *upper* and *steps* should also be specified. These parameters specify the temperature range and the number of intervals to evaluate the specific heats.

Parent Updater Data

in (string vector, required) 1st In Variable

average temperature of the species K

out (string vector, required) the specific heat $J/(kgK)$

Example

```
<Updater computeCpR>
  kind = equation2d
  onGrid = domain

  in = [temperature]
  out = [cpR]

  <Equation cpR>
      kind = transportCoeffSrc
      coeff = tempAvgSpecicHeatCp
      fileName = REACTIONS_ATOMIC_DATA
      cpType = kineticTheory
      numSpecies = NSPECIES
      #lower = 300.0
      #upper = 30000.0
      #steps = 100
    </Equation>
</Updater>
```

12.33.6 massFractionAvg

Average mass fraction of the species

Parameters

kind (string): transportCoeffSrc (fixed)

coeff (string): massFractionAvg (fixed)

numSpecies (int): number of species

fileName (string): name of the SpeciesDataFile containing the atomic data. Note that, massFractionAvg requires *MOLECULARWEIGHT* entered in the SpeciesDataFile.

Parent Updater Data

in (string vector, required) 1st In Variable

species number density $1/m^3$

2nd In Variable

species property

out (string vector, required) the average mass fraction

Example

```
<Updater computeCpAvg>
  kind = equation2d
  onGrid = domain

  in = [speciesDens,cpR]
  out = [cpAvg]

  <Equation cp>
    kind = transportCoeffSrc
    coeff = massFractionAvg
    numSpecies = NSPECIES
    fileName = REACTIONS_ATOMIC_DATA
  </Equation>
</Updater>
```

12.33.7 moleFractionAvg

Average molecular fraction of species

Parameters

`kind` (string): transportCoeffSrc (fixed)

`coeff` (string): moleFractionAvg (fixed)

`numSpecies` (int): number of species

`fileName` (string): name of the SpeciesDataFile containing the atomic data.

Parent Updater Data

`in` (string vector, required) 1st In Variable

species number density $1/m^3$

2nd In Variable

species property

`out` (string vector, required) the average molecular fraction

Example

```
<Updater computeCpAvg>
  kind = equation2d
  onGrid = domain

  in = [speciesDens,cpR]
  out = [cpAvg]

  <Equation cp>
    kind = transportCoeffSrc
    coeff = massFractionAvg
    numSpecies = NSPECIES
```

```
    fileName = REACTIONS_ATOMIC_DATA
  </Equation>
</Updater>
```

12.33.8 molecularWeightAvg

Average molecular weight of species

Parameters

kind (string): transportCoeffSrc (fixed)

coeff (string): molecularWeightAvg (fixed)

numSpecies (int): number of species

fileName (string): name of the SpeciesDataFile containing the MOLECULARWEIGHT data.

Parent Updater Data

in (string vector, required) 1st In Variable

species number density $1/m^3$

out (string vector, required) the average-molecular-weight

Example

```
<Updater computeMwAvg>
  kind = equation2d
  onGrid = domain

  in = [speciesDens]
  out = [mwAvg]

  <Equation mwavg>
    kind = transportCoeffSrc
    coeff = molecularWeightAvg
    numSpecies = NSPECIES
    fileName = REACTIONS_ATOMIC_DATA
  </Equation>
</Updater>
```

CHAPTER

THIRTEEN

BOUNDARY CONDITIONS

Defines an **Updater** block that is only applied to the boundary of the domain. Modified boundary values are stored in out. An example boundary condition updater block is given below:

```
<Updater Bc>
   kind = copy2d
   onGrid = domain
   entity = ghost
   out = [q]
</Updater>
```

The following parameters are common to all *Boundary Condition* blocks:

in (string vector) All boundary conditions have the option to take a string vector of input DataStruct. These DataStructs may or may not be used by the boundary condition

out (string vector) All boundary conditions take an output dataStruct.

onGrid (string) All boundary conditions take a string that tells the boundary condition which grid it is applied to

entity (string) All boundary conditions (except the periodicBc) take an entity that tells the updater what boundary the boundary condition will be applied to.

the entity *ghost* represents all boundaries for all USim grid types

the entities *left* (lower x boundary) *right* (upper x boundary) *bottom* (lower y boundary) *top* (upper y boundary) *back* (lower z boundary) *front* (upper z boundary) are defined for ntBodyFitted and cart grids.

kind (string) All boundary condition blocks take a string *kind* that species the type of boundary condtion. The different kinds of boundary condition available in USim are:

13.1 copy (1d, 2d, 3d)

Copies the data on the inside edge of the domain into the ghost cells for the given boundaries. When multiple halo cells are present the data from the first halo is copied to the second, the second to the 3rd etc...

13.1.1 Example

```
    <Updater bcLeft>
      kind = copy2d
      onGrid = domain
      out = [q]
      entity = ghost
    </Updater>
```

13.2 eulerBc (1d, 2d, 3d)

Sets boundary conditions for euler type equation systems

13.2.1 Parameters

`model` (string) Defines the hyperEqn to use, this should generally be eulerEqn.

`bcType` (string) There are currently 3 valid boundary condition types.

- *wall* which is a slip wall boundary condition.
- *noInflow* which is a boundary condition that lets fluid flow out of the domain, but does not let it flow in.
- *noSlip* which is a boundary condition where all components of velocity are set to zero at the wall.

13.2.2 Example

```
<Updater bcLeft>
  kind = eulerBc2d
  model = eulerEqn
  onGrid = domain
  out = [q]
  entity = left
</Updater>
```

13.3 functionBc (1d, 2d, 3d)

Defines the boundary condition using a function

13.3.1 Sub-Blocks

`Function` (block) The function that is used to define the values in *out*

13.3.2 Example

```
<Updater bcLeft>
  kind = functionBc2d

  onGrid = domain
  out = [q]
  entity = left

  <Function func>
    kind = exprFunc

    gamma = GAMMA
    P0 = $2.0*BASEMENT_PRESSURE$
    rho0 = $2.0*BASEMENT_DENSITY$
    b0 = B0
```

```
    preExprs = ["er = P0/(gamma-1)"]
    exprs = ["rho0", "0.0", "0.0", "0.0", "er","0.0","0.0","b0","0.0"]
  </Function>

</Updater>
```

13.4 generalBc (1d, 2d, 3d)

Applies a boundary condition that can be a general function of the input dataStructs as well as a separate list of dynVectors.

13.4.1 Data

dynVectors (string vector) Input 1 to N are input *dynVector*s which will be used in specifying the boundary condition.

13.4.2 Parameters

indVars_ (string vector) For each input variable an "indVars" string vector must be defined. So if *in = [magneticField, electricField]* where *magneticField* and *electricField* are each 3-component *nodalArray*s then the combiner block must define *indVars_magneticField = ["bx", "by", "bz"]* and *indVars_electricField = ["ex", "ey", "ez"]*. Note that the labels *"bx", "by", "bz"* and *"ex", "ey", "ez"* are arbitrary; the requirement is that there is a unique name for each component of each input data structure.

dynVectorVars_ (string vector) For each dynVector variable a "dynVars" string vector must be defined. So if *dynVectors = [a, b]* where *a* and *b* are each 3-component *dynVector*s then the combine block must define *dynVectorVars_a = ["a1","a2","a3"]* and *dynVectorVars_b = ["b1","b2","b3"]*. Note that the labels *"a1","a2","a3"* and *"b1","b2","b3"* are arbitrary; the requirement is that there is a unique name for each component of each input data structure.

preExprs (string vector) contains extra definitions that can be used in evaluating *exprs*. Expressions can also contain the internally defined variables 'x','y','z','t','dt','nX','nY','nZ' where the last 3 variables are surface normals.

exprs (string vector) Must be the size of the output vector q. Contains expressions representing each of the components of the output vector. This expression can also use the same internal variables as described in *preExprs*.

useModel (boolean) If useModel is set to true then *model* will need to be set. An example would be *model=eulerEqn*. With this option the values put into *exprs* are assumed to be in local coordinates and then USim will rotate them to global coordinates. Thus, with *model=eulerEqn* you can specify the input momentum density normal to the surface by specifying the x component of momentum and setting the remaining components to 0.

13.4.3 Example

```
<Updater bcBottom>
  kind = generalBc2d
  onGrid = domain

  in = [q]
```

```
  dynVectors = []

  indVars_q = ["rho","mx","my","mz","en"]
  exprs = ["rho","-mx","-my","-mz","en"]

  out = [q]
  entity = bottom
</Updater>
```

13.5 maxwellBc (1d, 2d, 3d)

Sets boundary conditions specific to Maxwell's equations

13.5.1 Parameters

`model` (string) Defines the hyperEqn to use, this should generally be maxwellEqn.

`bcType` (string) There are currently 2 valid boundary condition types.

- *conductor* which is a conducting wall boundary condition
- *axisymmetric* which is a boundary condition appropriate for cylindrical geometries on axis.

13.5.2 Example

```
<Updater emBcBottom>
  kind = maxwellBc2d
  model = maxwellEqn
  bcType = conductor
  onGrid = domain
  out = [q]
  entity = ghost
</Updater>
```

13.6 mhdBc (1d, 2d, 3d)

Sets the boundary condition for MHD type equations

13.6.1 Parameters

`model` (string) Defines the hyperbolic equation to use, this should generally be idealMhdEqn, mhdDednerEqn, twoTemperatureMhdDednerEqn, twoTemperatureMhdEqn, gasDynamicMhdEqn, idealMhdEosEqn or twoTemperatureMhdEosEqn

`bcType` (string) There are currently 3 valid boundary condition types.

- *conductingWall* which is a slip wall boundary condition.
- *noInflow* which is a boundary condition that lets fluid flow out of the domain, but does not let it flow in.
- *noSlip* which is a boundary condition where all components of velocity are set to zero at the wall.

13.6.2 Example

```
<Updater bcOpen>
  kind = mhdBc2d
  bcType = noInflow
  model = twoTemperatureMhdEosEqn
  onGrid = domain
  out = [q]
  entity = sideSetHalosId1
</Updater>
```

13.7 periodicCartBc (1d, 2d, 3d)

Applies a periodic boundary condition on the boundaries set up in the cart grid as defined by periodicDirs.

13.7.1 Example

```
<Updater periodic>
  kind = periodicCartBc2d
  onGrid = domain
  in = [q]
  out = [q]
</Updater>
```

13.8 simpleBc (1d, 2d, 3d)

Sets the boundary condition for MHD type equations

13.8.1 Parameters

`model` (string) The hyperbolic equation that describes the system being modelled. This should be one of the options available in *Hyperbolic Equations* and should match that used in (e.g) the *classicMusclUpdater (1d, 2d, 3d)* updater used to evolve the system.

`coefficients` (integer vector) The size of the coefficients vector must be the same as the number of elements in the input vector and the hyperbolic equation refered to by *model*. For models containing vector fields, the components of the vector are rotated into the coordinate system of the vector normal to the boundary. Then, the components of the vector are multiplied by the coefficients vector to set the boundary condition.

13.8.2 Example

In the example below, the hyperbolic equation is *eulerEqn*. This corresponds to a 5 component system with primary variables $[rho, mx, my, mz, en]$. Components 2-4 of this equation system correspond to the momentum vector, (mx, my, mz), which are rotated so that mx is aligned with the normal to the boundary. The coefficients for this example are $[1.0, -1.0, 1.0, 1.0, 1.0]$ and so the sign of the normal momentum is reversed at the boundary, which is corresponds to a wall boundary condition. Other boundary conditions can therefore be created by manipulating the coefficient vector.

```
<Updater fluidWall>
  kind = simpleBc2d
  model = eulerEqn
  coefficients = [1.0,-1.0,1.0,1.0,1.0]
  onGrid = domain
  out = [q]
  entity = bottom
</Updater>
```

13.9 sufaceEvaporation (1d, 2d, 3d)

Computes the surface evaporation rate of a compound material.

13.9.1 Data

`dynVectors` (string vector) A list of dynVectors that can be used in computing the boundary condition

`in` contains the surface temperature

13.9.2 Example

```
<Updater bcAbSurfProp>
  kind = surfaceEvaporation2d
  onGrid = domain
  in = [surfTemp]
  dynVectors = []
  variablesType = ablation
  storeSurfaceProperty = 1

  ablationModel = sonic
  numConstituents = 3
  satPressure = [10.0 154699.92824 956.0 10.0 97419.99 617.0 1.3e-6 76899.999 293.0]
  moleFraction = [MolF1 MolF2 MolF3]
  averageMolecularWeight = MWAb

  out = [abSurfProp]

  entity = sideSetHalosId3
</Updater>
```

13.10 tenMomentBc (1d, 2d, 3d)

Sets boundary conditions for tenMomentEqn type equation systems

13.10.1 Parameters

`model` (string) Defines the hyperbolic equation to use, this should generally be tenMomentEqn.

`bcType` (string) There are currently 3 valid boundary condition types.

- *wall* which is a slip wall boundary condition.
- *noInflow* which is a boundary condition that lets fluid flow out of the domain, but does not let it flow in.
- *noSlip* which is a boundary condition where all components of velocity are set to zero at the wall.

13.10.2 Example

```
<Updater bcLeft>
  kind = tenMomentBc2d
  model = tenMomentEqn
  onGrid = domain
  out = [q]
  entity = left
</Updater>
```

CHAPTER

FOURTEEN

TIME STEP RESTRICTION

Computes a minimum time step based on physical quantities, grid quantities and time. It could be used to determine the maximum explicitly stable time step based on wave speeds, or the maximum time step based on oscillations like the electron plasma oscillation. The TimeStepRestriction is used in conjunction with *timeStepRestrictionUpdater (1d, 2d, 3d)*. An example *TimeStepRestriction* is shown below:

```
<TimeStepRestriction wpe>
  kind = plasmaFrequency
  speciesCharge = ELECTRON_CHARGE
  speciesMass = ELECTRON_MASS
  epsilon0 = 1.0
  massDensityIndex = 0
</TimeStepRestriction>
```

The following parameters are common to all *TimeStepRestriction* blocks:

`in` (string vector, optional) Specifies the *nodalArray*s within the *in* attribute for the *timeStepRestrictionUpdater (1d, 2d, 3d)* that should be used for computing this time step restriction.

`includeInTimeStep` (bool, optional) Whether to include this time step restriction in the time step returned by the *timeStepRestrictionUpdater (1d, 2d, 3d)*. Default true.

`storeTimeStep` (bool, optional) Whether to store this time step restriction in the *timeSteps dynVector* specified in the *timeStepRestrictionUpdater (1d, 2d, 3d)*. Default: true.

`storeWaveSpeed` (bool, optional) Whether to store the wave speed associated with this in the *waveSpeeds dynVector* specified in the *timeStepRestrictionUpdater (1d, 2d, 3d)*. Default: true.

`applyCFLRestriction` (bool, optional) Whether to apply the CFL condition specified in the *timeStepRestrictionUpdater (1d, 2d, 3d)* to the time step computed in this restriction. Default: true.

`kind` (string, required) All *TimeStepRestriction* blocks take a string *kind* that species the type of time step restriction. The remainder of this section describes the different options for this parameter that are available in USim.

14.1 cyclotronFrequency (1d, 2d, 3d)

Computes the inverse cyclotron frequency which will then be used in determining the time step restriction.

14.1.1 Parameters

`speciesCharge` (float, required) Charge of the species for which we are computing the cyclotron frequency.

`speciesMass` (float, required) Mass of the species for which we are computing the cyclotron frequency.

magneticFieldIndexes (integer vector, required) The index of the magnetic field in the input data structure in *timeStepRestrictionUpdater (1d, 2d, 3d)*

14.1.2 Parent Updater Data

The following data structures should be specified to the *timeStepRestrictionUpdater (1d, 2d, 3d)* that calls the *cyclotronFrequency Time Step Restriction*.

in (string vector, required)

Mass Density (*nodalArray*, at least 1 component, required) The mass density of the plasma. The component of the data structure that contains the mass density is specified with the parameter *massIndex* (see below).

14.1.3 Example

The following block demonstrates *cyclotronFrequency* used in combination with *timeStepRestrictionUpdater (1d, 2d, 3d)* and *plasmaFrequency (1d, 2d, 3d)* to compute the time-step restriction in a plasma:

```
<Updater twofluidTimeStepRestrictions>
  kind = timeStepRestrictionUpdater1d
  in = [q]
  restrictions = [wpe, wce]
  onGrid = domain
  courantCondition = 1.0

  <TimeStepRestriction wpe>
      kind = plasmaFrequency1d
      cfl = 1.0
      speciesCharge = ELECTRON_CHARGE
      speciesMass = ELECTRON_MASS
      epsilon0 = 1.0
      massDensityIndex = 0
  </TimeStepRestriction>

  <TimeStepRestriction wce>
      kind = cyclotronFrequency1d
      speciesCharge = ELECTRON_CHARGE
      speciesMass = ELECTRON_MASS
      magneticFieldIndexes = [23, 24, 25]
      massDensityIndex = 0
  </TimeStepRestriction>

</Updater>
```

14.2 frequency (1d, 2d, 3d)

Computes the minimum time step suggested by an array of frequencies.

14.2.1 Parameters

components (int, required) Number of components in the input array. Each of the values in the array will be used to compute a time step restriction.

14.2.2 Parent Updater Data

The following data structures should be specified to the *timeStepRestrictionUpdater (1d, 2d, 3d)* that calls the *frequency Time Step Restriction*.

`in` (string vector, required)

`Reaction Frequency` (*nodalArray*, N components, required) An set of reaction frequencies to compute the restriction from

14.2.3 Example

The following block demonstrates *frequency* used in combination with *timeStepRestrictionUpdater (1d, 2d, 3d)* to compute a time-step restriction for a set of reactions:

```
<Updater timestepRestriction>
 kind = timeStepRestrictionUpdater2d
  in = [reactionFreq]
  onGrid = domain
  restrictions = [reaction]
  courantCondition = CFLR

  <TimeStepRestriction reaction>
    kind = frequency2d
    components = 1
  </TimeStepRestriction>
</Updater>
```

14.3 hyperbolic (1d, 2d, 3d)

Computes the minimum time step and fastest wave speed based on the courant condition for a specified *Hyperbolic Equations*.

14.3.1 Parameters

`model` (string, required) The *Hyperbolic Equations* used. Available options are:

eulerEqn

Defines the equations of inviscid compressible hydrodynamics:

$$\frac{\partial \rho}{\partial t} + \nabla \cdot [\rho\, \mathbf{u}] = 0$$
$$\frac{\partial \rho\, \mathbf{u}}{\partial t} + \nabla \cdot \left[\rho\, \mathbf{u}\, \mathbf{u}^T + \mathbb{I} P\right] = 0$$
$$\frac{\partial E}{\partial t} + \nabla \cdot [(E + P)\, \mathbf{u}] = 0$$

Here, $\mathbb{I}$ is the identity matrix, $P = \rho\epsilon(\gamma - 1)$ is the pressure of an ideal gas, ϵ is the specific internal energy and γ is the adiabatic index (ratio of specific heats).

Parameters

`gasGamma` (float) Specifies the adiabatic index (ratio of specific heats), γ. Defaults to 5/3.

`basementPressure` (float, optional) The minimum pressure allowed. Pressures below this value will be replaced with this value for primitive state, eigensystem and flux computations. Defaults to zero.

`basementDensity` (float, optional) The minimum density allowed. Densities below this value will be replaced with this value for primitive state, eigensystem and flux computations. Defaults to zero.

Parent Updater Data

`in` (string vector, required)

`Vector of Conserved Quantities` (*nodalArray*, 5-components, required) The vector of conserved quantities, $\mathbf{q}$ has 5 entries:

0. ρ: mass density
1. $\rho\, u_{\hat{\mathbf{i}}} = \rho\mathbf{u}\cdot\hat{\mathbf{i}}$: momentum density in the $\hat{\mathbf{i}}$ direction
2. $\rho\, u_{\hat{\mathbf{j}}} = \rho\mathbf{u}\cdot\hat{\mathbf{j}}$: momentum density in the $\hat{\mathbf{j}}$ direction
3. $\rho\, u_{\hat{\mathbf{k}}} = \rho\mathbf{u}\cdot\hat{\mathbf{k}}$: momentum density in the $\hat{\mathbf{k}}$ direction
4. $E = \frac{P}{\gamma-1} + \frac{1}{2}\rho|\mathbf{u}|^2$: total energy density

`out` (string vector, required) For the eulerEqn, one of four output variables are computed, depending on whether the equation is combined with an updater capable of computing fluxes (*classicMusclUpdater (1d, 2d, 3d)*), primitive variables (*computePrimitiveState(1d, 2d, 3d)*), the time step associated with the CFL condition (*timeStepRestrictionUpdater (1d, 2d, 3d)*) or the fastest wave speed in the grid (*timeStepRestrictionUpdater (1d, 2d, 3d)*).

`Vector of Fluxes` (*nodalArray*, 5-components) When combined with an updater that computes $\nabla\cdot\mathcal{F}(\mathbf{w})$ (e.g. *classicMusclUpdater (1d, 2d, 3d)*), the equation system returns:

0. $\nabla\cdot\mathcal{F}(\rho)$: mass flux
1. $\nabla\cdot\mathcal{F}\left(\rho\, u_{\hat{\mathbf{i}}}\right)$: $\hat{\mathbf{i}}$ momentum flux
2. $\nabla\cdot\mathcal{F}\left(\rho\, u_{\hat{\mathbf{j}}}\right)$: $\hat{\mathbf{j}}$ momentum flux
3. $\nabla\cdot\mathcal{F}\left(\rho\, u_{\hat{\mathbf{k}}}\right)$: $\hat{\mathbf{k}}$ momentum flux
4. $\nabla\cdot\mathcal{F}(E)$: total energy flux

`Vector of Primitive States` (*nodalArray*, 5-components) When combined with an updater that computes $\mathbf{w} = \mathbf{w}(\mathbf{q})$ (e.g. *computePrimitiveState(1d, 2d, 3d)*), the equation systen returns:

0. ρ: mass density
1. $u_{\hat{\mathbf{i}}} = \mathbf{u}\cdot\hat{\mathbf{i}}$: velocity in the $\hat{\mathbf{i}}$ direction
2. $u_{\hat{\mathbf{j}}} = \mathbf{u}\cdot\hat{\mathbf{j}}$: velocity in the $\hat{\mathbf{j}}$ direction
3. $u_{\hat{\mathbf{k}}} = \mathbf{u}\cdot\hat{\mathbf{k}}$: velocity in the $\hat{\mathbf{k}}$ direction
4. $P = \rho\epsilon(\gamma-1)$: ideal gas pressure

`Time Step` (*dynVector*, 1-component) When combined with the kind=hyperblic, model=eulerEqn *timeStepRestrictionUpdater (1d, 2d, 3d)*, and storeTimeStep is true, the equation system returns the time step consistent with the CFL condition across the entire simulation domain.

`Fastest Wave Speed` (*dynVector*, 1-component) When combined with the kind=hyperbolic, model=eulerEqn *timeStepRestrictionUpdater (1d, 2d, 3d)*, and storeWaveSpeed is true, the equation system returns the fastest wave speed across the entire simulation domain, c_{fast}.

Example

The following block demonstrates the eulerEqn used in combination with *classicMusclUpdater (1d, 2d, 3d)* to compute $\nabla \cdot \mathcal{F}(\mathbf{w})$:

```
<Updater hyper>
  kind=classicMuscl1d
  onGrid=domain
  timeIntegrationScheme=none
  numericalFlux=roeFlux
  limiter=[muscl]
  variableForm=primitive
  in=[q]
  out=[qnew]
  cfl=0.3
  equations=[euler]

  <Equation euler>
    kind=eulerEqn
    gasGamma=1.4
    basementDensity = 1.0e-5
    basementPressure = 1.0e-6
  </Equation>

</Updater>
```

realGasEqn

Real gas using a real gas equation of state. Requires the computation of specific heat and temperature and assignment of zero point energy outside of the equation. Assumes single temperature. The equations are solved in conservative form.

$$\frac{\partial}{\partial t}\begin{pmatrix} \rho \\ \rho\, u_x \\ \rho\, u_y \\ \rho\, u_z \\ e \end{pmatrix} + \nabla \cdot \begin{pmatrix} \rho\, u_x & \rho\, u_y & \rho\, u_z \\ \rho\, u_x^2 + P & \rho\, u_x\, u_y & \rho\, u_x\, u_z \\ \rho\, u_y\, u_x & \rho\, u_y\, u_y + P & \rho\, u_y\, u_z \\ \rho\, u_z\, u_x & \rho\, u_z\, u_y & \rho\, u_z\, u_z + P \\ u_x\,(e+P) & u_y\,(e+P) & u_z\,(e+P) \end{pmatrix} = 0$$

The energy is given by

$$e = \frac{1}{2}\rho\left(u_x^2 + u_y^2 + u_z^2\right) + \sum_i n_i\left(Cv_i T + e_{0\,i}\right) \tag{14.-2}$$

Parameters

`numSpecies` (float) The number of species modeled in the real gas system.

`basementPressure` (float) The minimum pressure allowed. Defaults to 0.

basementDensity (float) The minimum density allowed. Defaults to 0.

> **Note:** basementPressure and basementDensity are only used if correct=true

correct (boolean) Tells whether or not densities or pressures should be corrected when the fall below basement pressures or basement densities. When set to true pressure=max(basementPressure, pressure) and density = max(basementDensity, density). Defaults to false.

> **Note:** Setting correctNans or correct to true can lead to energy conservation errors

Parent Updater Data

in (string vector, required)

Vector of conserved quantities

0. ρ mass density
1. $\rho\, u_x$ x momentum density
2. $\rho\, u_y$ y momentum density
3. $\rho\, u_z$ z momentum density
4. e energy density

2nd variable (3n+1) 3n+1 auxiliary variables with n the number of species

0. variables 0-(n-1). n_i species number density
1. variables n-(2n-1). Cv_i species specific heat at constant volume
2. variables n-(3n-1). $e_{0\,i}$ species zero point energy density
3. variables 3n. T Temperature in Kelvin

Example

An example *realGas* equation block is given below

```
<Equation realGas>
  kind = realGasEqn
  numSpecies = 7
</Equation>
```

realGasEosEqn

Gas dynamics with a general equation of state. The equations are solved in conservative form.

$$\frac{\partial}{\partial t}\begin{pmatrix}\rho\\ \rho\,u_x\\ \rho\,u_y\\ \rho\,u_z\\ e\end{pmatrix}+\nabla\cdot\begin{pmatrix}\rho\,u_x & \rho\,u_y & \rho\,u_z\\ \rho\,u_x^2+P & \rho\,u_x\,u_y & \rho\,u_x\,u_z\\ \rho\,u_y\,u_x & \rho\,u_y\,u_y+P & \rho\,u_y\,u_z\\ \rho\,u_z\,u_x & \rho\,u_z\,u_y & \rho\,u_z\,u_z+P\\ u_x\,(e+P) & u_y\,(e+P) & u_z\,(e+P)\end{pmatrix}=0$$

Parameters

`basementPressure` (float) The minimum pressure allowed. Default is 0.

`basementDensity` (float) The minimum density allowed. Default is 0.

> **Note:** basementPressure and basementDensity are only used if correct=true

`correct` (boolean) Tells whether or not densities or pressures should be corrected when the fall below basement pressures or basement densities. When set to true pressure=max(basementPressure, pressure) and density = max(basementDensity, density)

Parent Updater Data

`in` (string vector, required)

`Vector of conserved quantities` (5 components)

0. ρ mass density
1. $\rho\, u_x$ x momentum density
2. $\rho\, u_y$ y momentum density
3. $\rho\, u_z$ z momentum density
4. e energy density

`fluid pressure` (1 component)

0. P total fluid pressure (not magnetic pressure included)

`gas dynamic sound speed` (1 component)

0. a estimate of the fluid sound speed

Example

An example *realGasEos* equation block is given below:

```
<Equation realGasEos>
  kind = realGasEosEqn
</Equation>
```

tenMomentEqn

Ideal compressible 10 moment fluid equations. The equations are solved in conservative form.

$$\frac{\partial}{\partial t}\begin{pmatrix} \rho \\ \rho\, u_x \\ \rho\, u_y \\ \rho\, u_z \\ \rho\, u_x^2 + P_{x\,x} \\ \rho\, u_x\, u_y + P_{x\,y} \\ \rho\, u_x\, u_z + P_{x\,z} \\ \rho\, u_y^2 + P_{y\,y} \\ \rho\, u_y\, u_z + P_{y\,z} \\ \rho\, u_z^2 + P_{z\,z} \end{pmatrix} + \nabla \cdot\ P = 0$$

where P is defined as

$$\begin{pmatrix} \rho\, u_x & \rho\, u_y & \rho\, u_z \\ \rho\, u_x^2 + P_{x\,x} & \rho\, u_x\, u_y + P_{x\,y} & \rho\, u_x\, u_z + P_{x\,z} \\ \rho\, u_y\, u_x + P_{x\,y} & \rho\, u_y\, u_y + P_{y\,y} & \rho\, u_y\, u_z + P_{y\,z} \\ \rho\, u_z\, u_x + P_{x\,z} & \rho\, u_z\, u_y + P_{y\,z} & \rho\, u_z\, u_z + P_{z\,z} \\ \rho\, u_x^3 + 3u_x P_{x\,x} & \rho\, u_y u_x^2 + u_x P_{y\,y} + 2u_x P_{x\,y} & \rho\, u_z u_x^2 + u_z P_{x\,x} + 2u_x P_{x\,z} \\ \rho\, u_x^2 u_y + 2\, u_x\, P_{x\,y} + u_y P_{x\,x} & 0 & 0 \\ \rho\, u_x^2 u_z + 2\, u_x\, P_{x\,z} + u_z P_{x\,x} & 0 & 0 \\ \rho\, u_x u_y^2 + u_x P_{y\,y} + 2u_y P_{x\,y} & \rho\, u_y^3 + 3u_y P_{y\,y} & 0 \\ \rho\, u_x u_y u_z + u_x P_{y\,z} + u_y P_{x\,z} + u_z P_{x\,y} & 0 & 0 \\ \rho\, u_x u_z^2 + u_x P_{z\,z} + 2u_z P_{x\,z} & 0 & \rho\, u_z^3 + 3u_z P_{z\,z} \end{pmatrix}$$

Parameters

`basementPressure` (float) The minimum pressure allowed. Defaults to 0.

`basementDensity` (float) The minimum density allowed. Defaults to 0.

Parent Updater Data

`in` (string vector, required)

1st variable

0. ρ mass density
1. $\rho\, u_x$ x momentum density
2. $\rho\, u_y$ y momentum density
3. $\rho\, u_z$ z momentum density
4. $\rho\, u_x^2 + P_{x\,x}$ xx energy density
5. $\rho\, u_x\, u_y + P_{x\,y}$ xy energy density
6. $\rho\, u_x\, u_z + P_{x\,z}$ xz energy density
7. $\rho\, u_y^2 + P_{y\,y}$ yy energy density
8. $\rho\, u_y\, u_z + P_{y\,z}$ yz energy density
9. $\rho\, u_z^2 + P_{z\,z}$ zz energy density

Example

An example *tenMoment* equation block is given below:

```
<Equation tenMoment>
  kind = tenMomentEqn
</Equation>
```

multiSpeciesSingleVelocityEqn

This equation represents continuity equations for n species. The species continuity equation is given by

$$\frac{\partial n_i}{\partial t} + \nabla_j \left(n_i \, u_j \right) = 0 \tag{14.-4}$$

Parameters

`basementNumberDensity` (float) The minimum species number density allowed

`basementDensity` (float) The minimum auxiliary variable mass density allowed. Defaults to 0.

`numberOfSpecies` (integer) The number of species that have continuity equations.

`useParentEigenvalues` (boolean) When set to true the eigenvalues of the parent system are used in computing dissipation in fluxes such as the localLaxFlux as well as time step restrictions. When set to false, the eigenvalue is simply u normal to the direction of interest.

Sub-Blocks

`Equation` (block) Defines the parent equation type of the system. The parent equation could be eulerEqn or idealMhdEqn for example. The first 4 components must be density, followed by the 3 components of momentum. This equation is used to compute the advection velocity and if useParentEigenvalues=true then the eigenvalues of this system are used to compute the level of dissipation in the flux functions.

Parent Updater Data

`in` (string vector, required)

`Species densities` Entries 1-N where N is the number of species

0. variables 0-(N-1) n_i number density of species i

`Vector of conserved quantities` Entries are determined by the Equation sub-block and only the first 4 entries are used in this equation. Entries 1-N where N the number variables in the parent equation

0. ρ species density
1. $\rho \, u_x$ species x momentum
2. $\rho \, u_y$ species y momentum
3. $\rho \, u_z$ species z momentum
4. all components beyond 3 are ignored.

Example

An example *multiSpeciesSingleVelocity* equation block is given below

```
<Equation speciesContinuity>
  kind = multiSpeciesSingleVelocityEqn
  useParentEigenvalues = true
  inputVariables = [qSpecies, q]
  numberOfSpecies = NSPECIES

  <Equation realGas>
    kind = realGasEqn
    inputVariables = [q, realGasVariables]
    numSpecies = NSPECIES
  </Equation>

</Equation>
```

mhdDednerEqn

Defines the equations of ideal compressible magnetohydrodynamics with divergence cleaning:

$$\begin{aligned}
\frac{\partial \rho}{\partial t} + \nabla \cdot [\rho\, \mathbf{u}] &= 0 \\
\frac{\partial \rho\, \mathbf{u}}{\partial t} + \nabla \cdot \left[\rho\, \mathbf{u}\, \mathbf{u}^T - \mathbf{b}\, \mathbf{b}^T + \mathbb{I}\left(P + \tfrac{1}{2}\left|\mathbf{b}\right|^2\right)\right] &= 0 \\
\frac{\partial E}{\partial t} + \nabla \cdot \left[(E + P)\, \mathbf{u} + \mathbf{e} \times \mathbf{b}\right] &= 0 \\
\frac{\partial \mathbf{b}^{\text{plasma}}}{\partial t} + \nabla \times \mathbf{e} + \nabla \psi &= 0 \\
\frac{\partial \psi}{\partial t} + \nabla \cdot \left[c_{\text{fast}}^2 \mathbf{b}\right] &= 0
\end{aligned}$$

Here, $\mathbb{I}$ is the identity matrix, $P = \rho\epsilon(\gamma - 1)$ is the pressure of an ideal gas, ϵ is the specific internal energy and γ is the adiabatic index (ratio of specific heats). The quantity c_{fast} corresponds to the fastest wave speed over the entire simulation domain; divergence errors are advected out of the domain with this speed.

The electromagnetic fields are defined as:

$$\begin{aligned}
\mathbf{b} &= \mathbf{b}^{\text{plasma}} + \mathbf{b}^{\text{external}} = \mu_0^{-1/2}\left(\mathbf{B}^{\text{plasma}} + \mathbf{B}^{\text{external}}\right) \\
\mathbf{e} &= -\mathbf{u} \times \mathbf{b} + \mathbf{e}^{\text{external}} = \mu_0^{-1/2}\left(-\mathbf{u} \times \mathbf{B} + \mathbf{E}^{\text{external}}\right)
\end{aligned}$$

Here, $\mathbf{b}^{\text{plasma}}$ is the magnetic field induced in the plasma by the inductive electric field, $\mathbf{e}$, while $\mathbf{e}^{\text{external}}$ and $\mathbf{b}^{\text{external}}$ are electromagnetic fields computed "externally" to the ideal magnetohydrodynamic equations.

Parameters

`basementPressure` (float, optional) The minimum pressure allowed. Pressures below this value will be replaced with this value for primitive state, eigensystem and flux computations. Defaults to zero.

`basementDensity` (float, optional) The minimum density allowed. Densities below this value will be replaced with this value for primitive state, eigensystem and flux computations. Defaults to zero.

`gasGamma` (float, optional) Specifies the adiabatic index (ratio of specific heats), γ. Defaults to 5/3.

`mu0` (float, optional) Optional value for the constant μ_0. Defaults to $4\pi \times 10^{-7}$.

`externalEfield` (string, optional) Specifies the name of the data structure containing the externally computed electric field, $\mathbf{e}^{\text{external}}$.

`externalBfield` (string, optional) Specifies the name of the data structure containing the externally computed magnetic field, $\mathbf{b}^{\text{external}}$.

Parent Updater Data

`in` (string vector, required)

`Vector of Conserved Quantities` (*nodalArray*, 9-components, required) The vector of conserved quantities, $\mathbf{q}$ has 9 entries:

0. ρ: mass density
1. $\rho\, u_{\hat{\mathbf{i}}} = \rho\mathbf{u}\cdot\hat{\mathbf{i}}$: momentum density in the $\hat{\mathbf{i}}$ direction
2. $\rho\, u_{\hat{\mathbf{j}}} = \rho\mathbf{u}\cdot\hat{\mathbf{j}}$: momentum density in the $\hat{\mathbf{j}}$ direction
3. $\rho\, u_{\hat{\mathbf{k}}} = \rho\mathbf{u}\cdot\hat{\mathbf{k}}$: momentum density in the $\hat{\mathbf{k}}$ direction
4. $E = \frac{P}{\gamma-1} + \frac{1}{2}\rho|\mathbf{u}|^2 + \frac{1}{2}|\mathbf{b}|^2$: total energy density
5. $b_{\hat{\mathbf{i}}} = \mathbf{b}\cdot\hat{\mathbf{i}} = \mu_0^{-1/2}\mathbf{B}\cdot\hat{\mathbf{i}}$: magnetic field normalized by permeability of free-space in the $\hat{\mathbf{i}}$ direction
6. $b_{\hat{\mathbf{j}}} = \mathbf{b}\cdot\hat{\mathbf{j}} = \mu_0^{-1/2}\mathbf{B}\cdot\hat{\mathbf{j}}$: magnetic field normalized by permeability of free-space in the $\hat{\mathbf{j}}$ direction
7. $b_{\hat{\mathbf{k}}} = \mathbf{b}\cdot\hat{\mathbf{k}} = \mu_0^{-1/2}\mathbf{B}\cdot\hat{\mathbf{k}}$: magnetic field normalized by permeability of free-space in the $\hat{\mathbf{k}}$ direction
8. ψ: correction potential

`Fastest Wave Speed` (*dynVector*, 1-component, required) The fastest wave speed across the entire simulation domain, c_{fast}. Can be computed using *hyperbolic (1d, 2d, 3d)* (see below).

`Externally Computed Electric Field` (*nodalArray*, 3-components, optional) Additional terms in the generalized Ohm's law, $\mathbf{E}^{\text{external}}$, computed "externally" to the ideal magnetohydrodynamic system. The data structure containing $\mathbf{e}^{\text{external}}$ is specified by the "externalEField" option described below.

0. $e^{\text{external}}_{\hat{\mathbf{i}}} = \mathbf{e}^{\text{external}}\cdot\hat{\mathbf{i}} = \mu_0^{-1/2}\mathbf{E}^{\text{external}}\cdot\hat{\mathbf{i}}$: "externally" computed electric field normalized by permeability of free-space in the $\hat{\mathbf{i}}$ direction.
1. $e^{\text{external}}_{\hat{\mathbf{j}}} = \mathbf{e}^{\text{external}}\cdot\hat{\mathbf{j}} = \mu_0^{-1/2}\mathbf{E}^{\text{external}}\cdot\hat{\mathbf{j}}$:"externally" computed electric field normalized by permeability of free-space in the $\hat{\mathbf{j}}$ direction
2. $e^{\text{external}}_{\hat{\mathbf{k}}} = \mathbf{e}^{\text{external}}\cdot\hat{\mathbf{k}} = \mu_0^{-1/2}\mathbf{E}^{\text{external}}\cdot\hat{\mathbf{k}}$: "externally" computed electric field normalized by permeability of free-space in the $\hat{\mathbf{k}}$ direction

`Externally Computed Magnetic Field` (*nodalArray*, 3-components, optional) Additional contribution to the magnetic field, $\mathbf{b}^{\text{external}}$, which is not evolved by the induction equation, but does contribute to the Lorentz force and the work done on the plasma. The data structure containing $\mathbf{b}^{\text{external}}$ is specified by the "externalBField" option described below.

0. $b^{\text{external}}_{\hat{\mathbf{i}}} = \mathbf{b}^{\text{external}}\cdot\hat{\mathbf{i}} = \mu_0^{-1/2}\mathbf{B}^{\text{external}}\cdot\hat{\mathbf{i}}$: magnetic field normalized by permeability of free-space in the $\hat{\mathbf{i}}$ direction

1. $b^{\text{external}}_{\hat{\mathbf{j}}} = \mathbf{b}^{\text{external}} \cdot \hat{\mathbf{j}} = \mu_0^{-1/2}\mathbf{B}^{\text{external}} \cdot \hat{\mathbf{j}}$: magnetic field normalized by permeability of free-space in the $\hat{\mathbf{j}}$ direction
2. $b^{\text{external}}_{\hat{\mathbf{k}}} = \mathbf{b}^{\text{external}} \cdot \hat{\mathbf{k}} = \mu_0^{-1/2}\mathbf{B}^{\text{external}} \cdot \hat{\mathbf{k}}$: magnetic field normalized by permeability of free-space in the $\hat{\mathbf{k}}$ direction

`out` (string vector, required) For the mhdDednerEqn, one of four output variables are computed, depending on whether the equation is combined with an updater capable of computing fluxes (*classicMusclUpdater (1d, 2d, 3d)*), primitive variables (*computePrimitiveState(1d, 2d, 3d)*), the time step associated with the CFL condition (*timeStepRestrictionUpdater (1d, 2d, 3d)*) or the fastest wave speed in the grid (*hyperbolic (1d, 2d, 3d)*).

`Vector of Fluxes` (*nodalArray*, 9-components) When combined with an updater that computes $\nabla \cdot \mathcal{F}(\mathbf{w})$ (e.g. *classicMusclUpdater (1d, 2d, 3d)*), the equation system returns:

0. $\nabla \cdot \mathcal{F}(\rho)$: mass flux
1. $\nabla \cdot \mathcal{F}\left(\rho\, u_{\hat{\mathbf{i}}}\right)$: $\hat{\mathbf{i}}$ momentum flux
2. $\nabla \cdot \mathcal{F}\left(\rho\, u_{\hat{\mathbf{j}}}\right)$: $\hat{\mathbf{j}}$ momentum flux
3. $\nabla \cdot \mathcal{F}\left(\rho\, u_{\hat{\mathbf{k}}}\right)$: $\hat{\mathbf{k}}$ momentum flux
4. $\nabla \cdot \mathcal{F}(E)$: total energy flux
5. $\nabla \cdot \mathcal{F}\left(b_{\hat{\mathbf{i}}}\right)$: $\hat{\mathbf{i}}$ magnetic field flux
6. $\nabla \cdot \mathcal{F}\left(b_{\hat{\mathbf{j}}}\right)$: $\hat{\mathbf{j}}$ magnetic field flux
7. $\nabla \cdot \mathcal{F}\left(b_{\hat{\mathbf{k}}}\right)$: $\hat{\mathbf{k}}$ magnetic field flux
8. $\nabla \cdot \mathcal{F}(\psi)$: correction potential flux

`Vector of Primitive States` (*nodalArray*, 9-components) When combined with an updater that computes $\mathbf{w} = \mathbf{w}(\mathbf{q})$ (e.g. *computePrimitiveState(1d, 2d, 3d)*), the equation systen returns:

0. ρ: mass density
1. $u_{\hat{\mathbf{i}}} = \mathbf{u} \cdot \hat{\mathbf{i}}$: velocity in the $\hat{\mathbf{i}}$ direction
2. $u_{\hat{\mathbf{j}}} = \mathbf{u} \cdot \hat{\mathbf{j}}$: velocity in the $\hat{\mathbf{j}}$ direction
3. $u_{\hat{\mathbf{k}}} = \mathbf{u} \cdot \hat{\mathbf{k}}$: velocity in the $\hat{\mathbf{k}}$ direction
4. $P = \rho\epsilon(\gamma - 1)$: ideal gas pressure
5. $b_{\hat{\mathbf{i}}} = \mathbf{b} \cdot \hat{\mathbf{i}} = \mu_0^{-1/2}\mathbf{B} \cdot \hat{\mathbf{i}}$: magnetic field normalized by permeability of free-space in the $\hat{\mathbf{i}}$ direction
6. $b_{\hat{\mathbf{j}}} = \mathbf{b} \cdot \hat{\mathbf{j}} = \mu_0^{-1/2}\mathbf{B} \cdot \hat{\mathbf{j}}$: magnetic field normalized by permeability of free-space in the $\hat{\mathbf{j}}$ direction
7. $b_{\hat{\mathbf{k}}} = \mathbf{b} \cdot \hat{\mathbf{k}} = \mu_0^{-1/2}\mathbf{B} \cdot \hat{\mathbf{k}}$: magnetic field normalized by permeability of free-space in the $\hat{\mathbf{k}}$ direction
8. ψ: correction potential

`Time Step` (*dynVector*, 1-component) When combined with *timeStepRestrictionUpdater (1d, 2d, 3d)*, the equation system returns the time step consisten with the CFL condition across the entire simulation domain.

`Fastest Wave Speed` (*dynVector*, 1-component) When combined with *hyperbolic (1d, 2d, 3d)*, the equation system returns the fastest wave speed across the entire simulation domain, c_{fast}.

Examples

The following block demonstrates the mhdDednerEqn used in combination with *classicMusclUpdater (1d, 2d, 3d)* to compute $\nabla \cdot \mathcal{F}(\mathbf{w})$ with an externally supplied magnetic field:

```
<Updater hyper>
  kind=classicMuscl1d
  onGrid=domain

  # input nodal component arrays
  in=[q   backgroundB]

  # output nodal component array
  out=[qnew]

  # input dynVector containing fastest wave speed
  waveSpeeds=[waveSpeed]

  # the numerical flux to use
  numericalFlux= hlldFlux

  # CFL number to use
  cfl=0.3
  # Form of variables to limit
  variableForm= primitive

  # Limiter; one per input nodal component array
  limiter=[minmod   minmod]

  # list of equations to solve
  equations=[mhd]

  <Equation mhd>
    kind=mhdDednerEqn
    gasGamma=1.4
    externalBfield="backgroundB"
  </Equation>

</Updater>
```

The following block demonstrates the mhdDednerEqn used in combination with *timeStepRestrictionUpdater (1d, 2d, 3d)* and *hyperbolic (1d, 2d, 3d)* to compute c_{fast} with an externally supplied magnetic field:

```
<Updater getWaveSpeed>
  kind=timeStepRestrictionUpdater1d
  onGrid=domain

  # input nodal component arrays
  in=[q   backgroundB]

  # output dynVector containing fastest wave speed
  waveSpeeds=[waveSpeed]

  # list of equations to compute fastest wave speed for
  restrictions=[idealMhd]
```

```
  # courant condition to apply to the timestep
  courantCondition=1.0

  <TimeStepRestriction idealMhd>
    kind=hyperbolic1d
    model=mhdDednerEqn
    gasGamma= 1.4
    externalBfield=True
    includeInTimeStep=False
  </TimeStepRestriction>
</Updater>
```

mhdDednerEosEqn

Defines the equations of ideal compressible magnetohydrodynamics with and arbitrary equation of state (EOS) and divergence cleaning:

$$\begin{aligned}
\frac{\partial \rho}{\partial t} + \nabla \cdot [\rho\, \mathbf{u}] &= 0 \\
\frac{\partial \rho\, \mathbf{u}}{\partial t} + \nabla \cdot \left[\rho\, \mathbf{u}\, \mathbf{u}^T - \mathbf{b}\, \mathbf{b}^T + \mathbb{I}\left(P + \tfrac{1}{2}\,|\mathbf{b}|^2\right)\right] &= 0 \\
\frac{\partial E}{\partial t} + \nabla \cdot [(E + P)\, \mathbf{u} + \mathbf{e} \times \mathbf{b}] &= 0 \\
\frac{\partial \mathbf{b}^{\text{plasma}}}{\partial t} + \nabla \times \mathbf{e} + \nabla \psi &= 0 \\
\frac{\partial \psi}{\partial t} + \nabla \cdot \left[c_{\text{fast}}^2 \mathbf{b}\right] &= 0
\end{aligned}$$

Here, $\mathbb{I}$ is the identity matrix and P is the pressure as specified by an external EOS. Updaters that compute all the data required from an EOS are found in *vanDerWaalsComputeVariables*, *sesameComputeVariables* and *propaceosComputeVariables*. The quantity c_{fast} corresponds to the fastest wave speed over the entire simulation domain; divergence errors are advected out of the domain with this speed.

The electromagnetic fields are defined as:

$$\begin{aligned}
\mathbf{b} &= \mathbf{b}^{\text{plasma}} + \mathbf{b}^{\text{external}} = \mu_0^{-1/2} \left(\mathbf{B}^{\text{plasma}} + \mathbf{B}^{\text{external}}\right) \\
\mathbf{e} &= -\mathbf{u} \times \mathbf{b} + \mathbf{e}^{\text{external}} = \mu_0^{-1/2} \left(-\mathbf{u} \times \mathbf{B} + \mathbf{E}^{\text{external}}\right)
\end{aligned}$$

Here, $\mathbf{b}^{\text{plasma}}$ is the magnetic field induced in the plasma by the inductive electric field, $\mathbf{e}$, while $\mathbf{e}^{\text{external}}$ and $\mathbf{b}^{\text{external}}$ are electromagnetic fields computed "externally" to the ideal magnetohydrodynamic equations.

Parameters

`basementPressure` (float, optional) The minimum pressure allowed. Pressures below this value will be replaced with this value for primitive state, eigensystem and flux computations. Defaults to zero.

`basementDensity` (float, optional) The minimum density allowed. Densities below this value will be replaced with this value for primitive state, eigensystem and flux computations. Defaults to zero.

`mu0` (float, optional) Optional value for the constant μ_0. Defaults to $4\pi \times 10^{-7}$.

`externalEfield` (string, optional) Specifies the name of the data structure containing the externally computed electric field, $\mathbf{e}^{\text{external}}$.

`externalBfield` (string, optional) Specifies the name of the data structure containing the externally computed magnetic field, $\mathbf{b}^{\text{external}}$.

Parent Updater Data

`in` (string vector, required)

`Vector of Conserved Quantities` (*nodalArray*, 9-components, required) The vector of conserved quantities, $\mathbf{q}$ has 9 entries:

0. ρ: mass density
1. $\rho\, u_{\hat{\mathbf{i}}} = \rho\mathbf{u}\cdot\hat{\mathbf{i}}$: momentum density in the $\hat{\mathbf{i}}$ direction
2. $\rho\, u_{\hat{\mathbf{j}}} = \rho\mathbf{u}\cdot\hat{\mathbf{j}}$: momentum density in the $\hat{\mathbf{j}}$ direction
3. $\rho\, u_{\hat{\mathbf{k}}} = \rho\mathbf{u}\cdot\hat{\mathbf{k}}$: momentum density in the $\hat{\mathbf{k}}$ direction
4. $E = \rho\epsilon + \frac{1}{2}\rho|\mathbf{u}|^2 + \frac{1}{2}|\mathbf{b}|^2$: total energy density where ϵ is the specific internal energy
5. $b_{\hat{\mathbf{i}}} = \mathbf{b}\cdot\hat{\mathbf{i}} = \mu_0^{-1/2}\mathbf{B}\cdot\hat{\mathbf{i}}$: magnetic field normalized by permeability of free-space in the $\hat{\mathbf{i}}$ direction
6. $b_{\hat{\mathbf{j}}} = \mathbf{b}\cdot\hat{\mathbf{j}} = \mu_0^{-1/2}\mathbf{B}\cdot\hat{\mathbf{j}}$: magnetic field normalized by permeability of free-space in the $\hat{\mathbf{j}}$ direction
7. $b_{\hat{\mathbf{k}}} = \mathbf{b}\cdot\hat{\mathbf{k}} = \mu_0^{-1/2}\mathbf{B}\cdot\hat{\mathbf{k}}$: magnetic field normalized by permeability of free-space in the $\hat{\mathbf{k}}$ direction
8. ψ: correction potential

`Pressure` (*nodalArray*, 1-component, required) Value of the pressure as computed by the external EOS.

`Sound speed squared` (*nodalArray*, 1-component, required) Value of the sound speed squared as computed by the external EOS.

`internal energy` (*nodalArray*, 1-component, required) Value of the internal energy ($\rho\epsilon$) as computed by the external EOS.

`Fastest Wave Speed` (*dynVector*, 1-component, required) The fastest wave speed across the entire simulation domain, c_{fast}. Can be computed using *hyperbolic (1d, 2d, 3d)* (see below).

`Externally Computed Electric Field` (*nodalArray*, 3-components, optional) Additional terms in the generalized Ohm's law, $\mathbf{E}^{\text{external}}$, computed "externally" to the ideal magnetohydrodynamic system. The data structure containing $\mathbf{e}^{\text{external}}$ is specified by the "externalEField" option described below.

0. $e^{\text{external}}_{\hat{\mathbf{i}}} = \mathbf{e}^{\text{external}}\cdot\hat{\mathbf{i}} = \mu_0^{-1/2}\mathbf{E}^{\text{external}}\cdot\hat{\mathbf{i}}$: "externally" computed electric field normalized by permeability of free-space in the $\hat{\mathbf{i}}$ direction.
1. $e^{\text{external}}_{\hat{\mathbf{j}}} = \mathbf{e}^{\text{external}}\cdot\hat{\mathbf{j}} = \mu_0^{-1/2}\mathbf{E}^{\text{external}}\cdot\hat{\mathbf{j}}$:"externally" computed electric field normalized by permeability of free-space in the $\hat{\mathbf{j}}$ direction
2. $e^{\text{external}}_{\hat{\mathbf{k}}} = \mathbf{e}^{\text{external}}\cdot\hat{\mathbf{k}} = \mu_0^{-1/2}\mathbf{E}^{\text{external}}\cdot\hat{\mathbf{k}}$: "externally" computed electric field normalized by permeability of free-space in the $\hat{\mathbf{k}}$ direction

`Externally Computed Magnetic Field` (*nodalArray*, 3-components, optional) Additional contribution to the magnetic field, $\mathbf{b}^{\text{external}}$, which is not evolved by the induction equation, but does contribute to the Lorentz force and the work done on the plasma. The data structure containing $\mathbf{b}^{\text{external}}$ is specified by the "externalBField" option described below.

0. $b_{\hat{\mathbf{i}}}^{\text{external}} = \mathbf{b}^{\text{external}} \cdot \hat{\mathbf{i}} = \mu_0^{-1/2}\mathbf{B}^{\text{external}} \cdot \hat{\mathbf{i}}$: magnetic field normalized by permeability of free-space in the $\hat{\mathbf{i}}$ direction
1. $b_{\hat{\mathbf{j}}}^{\text{external}} = \mathbf{b}^{\text{external}} \cdot \hat{\mathbf{j}} = \mu_0^{-1/2}\mathbf{B}^{\text{external}} \cdot \hat{\mathbf{j}}$: magnetic field normalized by permeability of free-space in the $\hat{\mathbf{j}}$ direction
2. $b_{\hat{\mathbf{k}}}^{\text{external}} = \mathbf{b}^{\text{external}} \cdot \hat{\mathbf{k}} = \mu_0^{-1/2}\mathbf{B}^{\text{external}} \cdot \hat{\mathbf{k}}$: magnetic field normalized by permeability of free-space in the $\hat{\mathbf{k}}$ direction

`out` (string vector, required) For the mhdDednerEosEqn, one of four output variables are computed, depending on whether the equation is combined with an updater capable of computing fluxes (*classicMusclUpdater (1d, 2d, 3d)*), primitive variables (*computePrimitiveState(1d, 2d, 3d)*), the time step associated with the CFL condition (*timeStepRestrictionUpdater (1d, 2d, 3d)*) or the fastest wave speed in the grid (*hyperbolic (1d, 2d, 3d)*).

`Vector of Fluxes` (*nodalArray*, 9-components) When combined with an updater that computes $\nabla \cdot \mathcal{F}(\mathbf{w})$ (e.g. *classicMusclUpdater (1d, 2d, 3d)*), the equation system returns:

0. $\nabla \cdot \mathcal{F}(\rho)$: mass flux
1. $\nabla \cdot \mathcal{F}\left(\rho\, u_{\hat{\mathbf{i}}}\right)$: $\hat{\mathbf{i}}$ momentum flux
2. $\nabla \cdot \mathcal{F}\left(\rho\, u_{\hat{\mathbf{j}}}\right)$: $\hat{\mathbf{j}}$ momentum flux
3. $\nabla \cdot \mathcal{F}\left(\rho\, u_{\hat{\mathbf{k}}}\right)$: $\hat{\mathbf{k}}$ momentum flux
4. $\nabla \cdot \mathcal{F}(E)$: total energy flux
5. $\nabla \cdot \mathcal{F}\left(b_{\hat{\mathbf{i}}}\right)$: $\hat{\mathbf{i}}$ magnetic field flux
6. $\nabla \cdot \mathcal{F}\left(b_{\hat{\mathbf{j}}}\right)$: $\hat{\mathbf{j}}$ magnetic field flux
7. $\nabla \cdot \mathcal{F}\left(b_{\hat{\mathbf{k}}}\right)$: $\hat{\mathbf{k}}$ magnetic field flux
8. $\nabla \cdot \mathcal{F}(\psi)$: correction potential flux

`Vector of Primitive States` (*nodalArray*, 9-components) When combined with an updater that computes $\mathbf{w} = \mathbf{w}(\mathbf{q})$ (e.g. *computePrimitiveState(1d, 2d, 3d)*), the equation systen returns:

0. ρ: mass density
1. $u_{\hat{\mathbf{i}}} = \mathbf{u} \cdot \hat{\mathbf{i}}$: velocity in the $\hat{\mathbf{i}}$ direction
2. $u_{\hat{\mathbf{j}}} = \mathbf{u} \cdot \hat{\mathbf{j}}$: velocity in the $\hat{\mathbf{j}}$ direction
3. $u_{\hat{\mathbf{k}}} = \mathbf{u} \cdot \hat{\mathbf{k}}$: velocity in the $\hat{\mathbf{k}}$ direction
4. $P = \rho\epsilon(\gamma - 1)$: ideal gas pressure
5. $b_{\hat{\mathbf{i}}} = \mathbf{b} \cdot \hat{\mathbf{i}} = \mu_0^{-1/2}\mathbf{B} \cdot \hat{\mathbf{i}}$: magnetic field normalized by permeability of free-space in the $\hat{\mathbf{i}}$ direction
6. $b_{\hat{\mathbf{j}}} = \mathbf{b} \cdot \hat{\mathbf{j}} = \mu_0^{-1/2}\mathbf{B} \cdot \hat{\mathbf{j}}$: magnetic field normalized by permeability of free-space in the $\hat{\mathbf{j}}$ direction
7. $b_{\hat{\mathbf{k}}} = \mathbf{b} \cdot \hat{\mathbf{k}} = \mu_0^{-1/2}\mathbf{B} \cdot \hat{\mathbf{k}}$: magnetic field normalized by permeability of free-space in the $\hat{\mathbf{k}}$ direction
8. ψ: correction potential

`Time Step` (*dynVector*, 1-component) When combined with *timeStepRestrictionUpdater (1d, 2d, 3d)*, the equation system returns the time step consisten with the CFL condition across the entire simulation domain.

`Fastest Wave Speed` (*dynVector*, 1-component) When combined with *hyperbolic (1d, 2d, 3d)*, the equation system returns the fastest wave speed across the entire simulation domain, c_{fast}.

Examples

The following block demonstrates the mhdDednerEosEqn used in combination with *classicMusclUpdater (1d, 2d, 3d)* to compute $\nabla \cdot \mathcal{F}(\mathbf{w})$:

```
<Updater hyper>
  kind = classicMuscl2d
  onGrid = domain

  # input nodal component arrays
  in=[q, pressure, soundSqr, intEnergy]

  # output nodal component arrays
  out = [qNew]

  # input dynVector containing fastest wave speed
  waveSpeeds = [waveSpeed]

  # the numerical flux to use
  numericalFlux = hlldFlux

  # CFL number to use
  cfl = 0.5

  # determines solve is conservative or primitive
  variableForm = conservative

  # Limiter; one per input nodal component array
  limiter=[muscl, muscl, muscl, muscl]

  # list of equations to solve
  equations = [mhd]

  <Equation mhd>
    kind=mhdDednerEosEqn
    mu0=1.0
  </Equation>

</Updater>
```

The following block demonstrates the mhdDednerEosEqn used in combination with *timeStepRestrictionUpdater (1d, 2d, 3d)* and *hyperbolic (1d, 2d, 3d)* to compute c_{fast} with an externally supplied magnetic field:

```
<Updater getWaveSpeed>
  kind=timeStepRestrictionUpdater2d
  onGrid=domain

  # input nodal component arrays
  in=[q, pressure, soundSqr, intEnergy]
```

```
  # output dynVector containing fastest wave speed
  waveSpeeds=[waveSpeed]

  # list of equations to compute fastest wave speed for
  restrictions=[idealMhd]

  # courant condition to apply to the timestep
  courantCondition=0.5

  <TimeStepRestriction idealMhd>
    kind=hyperbolic1d
    model=mhdDednerEosEqn
    mu0=1.0
  </TimeStepRestriction>
</Updater>
```

gasDynamicMhdDednerEqn

Defines the equations of inviscid fluid dynamics coupled to pre-Maxwell's equations in source term form with divergence cleaning:

$$\frac{\partial \rho}{\partial t} + \nabla \cdot [\rho \, \mathbf{u}] = 0$$

$$\frac{\partial (\rho \mathbf{u})}{\partial t} + \nabla \cdot \left[\rho \, \mathbf{u} \, \mathbf{u}^T + \mathbb{I} P\right] = \sum_{\text{species}} \left(q^{\text{species}} \mathbf{E} + \mathbf{J}^{\text{species}} \times \mathbf{B}\right)$$

$$\frac{\partial E}{\partial t} + \nabla \cdot [(E + P) \, \mathbf{u}] = \sum_{\text{species}} \mathbf{J}^{\text{species}} \cdot \mathbf{E}^{\text{species}}$$

$$\frac{\partial \mathbf{B}^{\text{plasma}}}{\partial t} + \nabla \times \mathbf{E} + \nabla \psi = 0$$

$$\frac{\partial \psi}{\partial t} + \nabla \cdot \left[c_{\text{fast}}^2 \mathbf{b}\right] = 0$$

Here, q^{species} is the species charge density, $\mathbf{J}^{\text{species}}$ is the species current density, $\mathbb{I}$ is the identity matrix, $P = \rho\epsilon(\gamma - 1)$ is the pressure of an ideal gas, ϵ is the specific internal energy and γ is the adiabatic index (ratio of specific heats). The quantity c_{fast} corresponds to the fastest wave speed over the entire simulation domain; divergence errors are advected out of the domain with this speed.

In order to integrate these equations, USim casts them into flux-conservative form using the following standard identities (note that the use of these identities does not require an assumption of quasi-neutrality):

$$\sum_{\text{species}} \left(q^{\text{species}} \mathbf{E} + \mathbf{J}^{\text{species}} \times \mathbf{B}\right) = -\frac{\partial c^{-2} \mathbf{S}^{\text{EM}}}{\partial t} + \nabla \cdot \mathcal{T}^{\text{EM}}$$

$$\sum_{\text{species}} \mathbf{J}^{\text{species}} \cdot \mathbf{E} = -\frac{\partial E^{\text{EM}}}{\partial t} - \nabla \cdot \mathbf{S}^{\text{EM}}$$

Here, $\mathcal{T}^{\text{EM}}$ is the electromagnetic stress tensor and $\mathbf{S}^{\text{EM}}$ is the electromagnetic energy (Poynting) flux vector, which are defined as:

$$\mathcal{T}^{\text{EM}} = \frac{1}{\mu_0}\left(\frac{\mathbf{E}\mathbf{E}^T}{c^2} + \mathbf{B}\mathbf{B}^T\right) + \mathbb{I} E_{\text{EM}} = \frac{\mathbf{e}\mathbf{e}^T}{c^2} + \mathbf{b}\mathbf{b}^T + \mathbb{I} E_{\text{EM}}$$

$$\mathbf{S}^{\text{EM}} = \mu_0^{-1} \mathbf{E} \times \mathbf{B} = \mathbf{e} \times \mathbf{b}$$

$$E^{\text{EM}} = \frac{1}{2\mu_0}\left(\frac{|\mathbf{E}|^2}{c^2} + |\mathbf{B}|^2\right) = \frac{1}{2}\left(\frac{|\mathbf{e}|^2}{c^2} + |\mathbf{b}|^2\right)$$

Here, E^{EM} is the electromagnetic energy density and the electromagnetic fields are defined as:

$$\mathbf{b} = \mathbf{b}^{\mathrm{plasma}} + \mathbf{b}^{\mathrm{external}} = \mu_0^{-1/2}\left(\mathbf{B}^{\mathrm{plasma}} + \mathbf{B}^{\mathrm{external}}\right)$$
$$\mathbf{e} = -\mathbf{u}\times\mathbf{b} + \mathbf{e}^{\mathrm{external}} = \mu_0^{-1/2}\left(-\mathbf{u}\times\mathbf{B} + \mathbf{E}^{\mathrm{external}}\right)$$

Here, $\mathbf{b}^{\mathrm{plasma}}$ is the magnetic field induced in the plasma by the inductive electric field, $\mathbf{e}$, while $\mathbf{e}^{\mathrm{external}}$ and $\mathbf{b}^{\mathrm{external}}$ are electromagnetic fields computed "externally" to the pre-Maxwell equations.

With these identitifications, the gasDynamicMhdDednerEqn takes the form:

$$\frac{\partial \rho}{\partial t} + \nabla\cdot\left[\rho\,\mathbf{u}\right] = 0$$
$$\frac{\partial\left(\rho\mathbf{u} + c^{-2}\mathbf{S}^{\mathrm{EM}}\right)}{\partial t} + \nabla\cdot\left[\rho\,\mathbf{u}\,\mathbf{u}^T + \mathbb{I}P - \mathcal{T}^{\mathrm{EM}}\right] = 0$$
$$\frac{\partial\left(E + E^{\mathrm{EM}}\right)}{\partial t} + \nabla\cdot\left[\left(E+P\right)\mathbf{u} + \mathbf{S}^{\mathrm{EM}}\right] = 0$$
$$\frac{\partial \mathbf{b}^{\mathrm{plasma}}}{\partial t} + \nabla\times\mathbf{e} + \nabla\psi = 0$$
$$\frac{\partial \psi}{\partial t} + \nabla\cdot\left[c_{\mathrm{fast}}^2\,\mathbf{b}\right] = 0$$

This flux-conservative formulation is implemented in USim.

Parameters

`lightSpeed` (float, optional) The speed of light in m/s. Defaults to 2.99792458e8.

`basementPressure` (float, optional) The minimum pressure allowed. Pressures below this value will be replaced with this value for primitive state, eigensystem and flux computations. Defaults to zero.

`basementDensity` (float, optional) The minimum density allowed. Densities below this value will be replaced with this value for primitive state, eigensystem and flux computations. Defaults to zero.

`gasGamma` (float, optional) Specifies the adiabatic index (ratio of specific heats), γ. Defaults to 5/3.

`externalEfield` (string, optional) Specifies the name of the data structure containing the externally computed electric field, $\mathbf{e}^{\mathrm{external}}$.

`externalBfield` (string, optional) Specifies the name of the data structure containing the externally computed magnetic field, $\mathbf{b}^{\mathrm{external}}$.

Parent Updater Data

`in` (string vector, required)

`Vector of Conserved Quantities` (*nodalArray*, 9-components, required) The vector of conserved quantities, $\mathbf{q}$ has 9 entries:

0. ρ: mass density
1. $\rho\,u_{\hat{\mathbf{i}}} + c^{-2}S_{\hat{\mathbf{i}}}^{\mathrm{EM}} = \left(\rho\mathbf{u} + c^{-2}\mathbf{S}^{\mathrm{EM}}\right)\cdot\hat{\mathbf{i}}$: total momentum density in the $\hat{\mathbf{i}}$ direction
2. $\rho\,u_{\hat{\mathbf{j}}} + c^{-2}S_{\hat{\mathbf{j}}}^{\mathrm{EM}} = \left(\rho\mathbf{u} + c^{-2}\mathbf{S}^{\mathrm{EM}}\right)\cdot\hat{\mathbf{j}}$: total momentum density in the $\hat{\mathbf{j}}$ direction
3. $\rho\,u_{\hat{\mathbf{k}}} + c^{-2}S_{\hat{\mathbf{k}}}^{\mathrm{EM}} = \left(\rho\mathbf{u} + c^{-2}\mathbf{S}^{\mathrm{EM}}\right)\cdot\hat{\mathbf{k}}$: total momentum density in the $\hat{\mathbf{k}}$ direction
4. $E + E^{\mathrm{EM}} = \frac{P}{\gamma-1} + \frac{1}{2}\rho|\mathbf{u}|^2 + E^{\mathrm{EM}}$: total energy density

5. $b_{\hat{\mathbf{i}}} = \mathbf{b} \cdot \hat{\mathbf{i}} = \mu_0^{-1/2} \mathbf{B} \cdot \hat{\mathbf{i}}$: magnetic field normalized by permeability of free-space in the $\hat{\mathbf{i}}$ direction
6. $b_{\hat{\mathbf{j}}} = \mathbf{b} \cdot \hat{\mathbf{j}} = \mu_0^{-1/2} \mathbf{B} \cdot \hat{\mathbf{j}}$: magnetic field normalized by permeability of free-space in the $\hat{\mathbf{j}}$ direction
7. $b_{\hat{\mathbf{k}}} = \mathbf{b} \cdot \hat{\mathbf{k}} = \mu_0^{-1/2} \mathbf{B} \cdot \hat{\mathbf{k}}$: magnetic field normalized by permeability of free-space in the $\hat{\mathbf{k}}$ direction
8. ψ: correction potential

`Fastest Wave Speed` (***dynVector***, **1-component, required**) The fastest wave speed across the entire simulation domain, c_{fast}. Can be computed using *hyperbolic (1d, 2d, 3d)* (see below).

`Externally Computed Electric Field` (***nodalArray***, **3-components, optional**) Additional terms in the generalized Ohm's law, $\mathbf{E}^{\text{external}}$, computed "externally" to the ideal magnetohydrodynamic system. The data structure containing $\mathbf{e}^{\text{external}}$ is specified by the "externalEField" option described below.

0. $e^{\text{external}}_{\hat{\mathbf{i}}} = \mathbf{e}^{\text{external}} \cdot \hat{\mathbf{i}} = \mu_0^{-1/2} \mathbf{E}^{\text{external}} \cdot \hat{\mathbf{i}}$: "externally" computed electric field normalized by permeability of free-space in the $\hat{\mathbf{i}}$ direction.
1. $e^{\text{external}}_{\hat{\mathbf{j}}} = \mathbf{e}^{\text{external}} \cdot \hat{\mathbf{j}} = \mu_0^{-1/2} \mathbf{E}^{\text{external}} \cdot \hat{\mathbf{j}}$:"externally" computed electric field normalized by permeability of free-space in the $\hat{\mathbf{j}}$ direction
2. $e^{\text{external}}_{\hat{\mathbf{k}}} = \mathbf{e}^{\text{external}} \cdot \hat{\mathbf{k}} = \mu_0^{-1/2} \mathbf{E}^{\text{external}} \cdot \hat{\mathbf{k}}$: "externally" computed electric field normalized by permeability of free-space in the $\hat{\mathbf{k}}$ direction

`Externally Computed Magnetic Field` (***nodalArray***, **3-components, optional**) Additional contribution to the magnetic field, $\mathbf{b}^{\text{external}}$, which is not evolved by the induction equation, but does contribute to the Lorentz force and the work done on the plasma. The data structure containing $\mathbf{b}^{\text{external}}$ is specified by the "externalBField" option described below.

0. $b^{\text{external}}_{\hat{\mathbf{i}}} = \mathbf{b}^{\text{external}} \cdot \hat{\mathbf{i}} = \mu_0^{-1/2} \mathbf{B}^{\text{external}} \cdot \hat{\mathbf{i}}$: magnetic field normalized by permeability of free-space in the $\hat{\mathbf{i}}$ direction
1. $b^{\text{external}}_{\hat{\mathbf{j}}} = \mathbf{b}^{\text{external}} \cdot \hat{\mathbf{j}} = \mu_0^{-1/2} \mathbf{B}^{\text{external}} \cdot \hat{\mathbf{j}}$: magnetic field normalized by permeability of free-space in the $\hat{\mathbf{j}}$ direction
2. $b^{\text{external}}_{\hat{\mathbf{k}}} = \mathbf{b}^{\text{external}} \cdot \hat{\mathbf{k}} = \mu_0^{-1/2} \mathbf{B}^{\text{external}} \cdot \hat{\mathbf{k}}$: magnetic field normalized by permeability of free-space in the $\hat{\mathbf{k}}$ direction

`out` **(string vector, required)** For the gasDynamicMhdDednerEqn, one of four output variables are computed, depending on whether the equation is combined with an updater capable of computing fluxes (*classicMusclUpdater (1d, 2d, 3d)*), primitive variables (*computePrimitiveState(1d, 2d, 3d)*), the time step associated with the CFL condition (*timeStepRestrictionUpdater (1d, 2d, 3d)*) or the fastest wave speed in the grid (*hyperbolic (1d, 2d, 3d)*).

`Vector of Fluxes` (***nodalArray***, **9-components**) When combined with an updater that computes $\nabla \cdot \mathcal{F}(\mathbf{w})$ (e.g. *classicMusclUpdater (1d, 2d, 3d)*), the equation system returns:

0. $\nabla \cdot \mathcal{F}(\rho)$: mass flux
1. $\nabla \cdot \mathcal{F}\left(\rho\, u_{\hat{\mathbf{i}}} + c^{-2} S^{\text{EM}}_{\hat{\mathbf{i}}}\right)$: $\hat{\mathbf{i}}$ momentum flux
2. $\nabla \cdot \mathcal{F}\left(\rho\, u_{\hat{\mathbf{j}}} + c^{-2} S^{\text{EM}}_{\hat{\mathbf{j}}}\right)$: $\hat{\mathbf{j}}$ momentum flux
3. $\nabla \cdot \mathcal{F}\left(\rho\, u_{\hat{\mathbf{k}}} + c^{-2} S^{\text{EM}}_{\hat{\mathbf{k}}}\right)$: $\hat{\mathbf{k}}$ momentum flux

4. $\nabla \cdot \mathcal{F}(E)$: total energy flux
5. $\nabla \cdot \mathcal{F}\left(b_{\hat{\mathbf{i}}}\right)$: $\hat{\mathbf{i}}$ magnetic field flux
6. $\nabla \cdot \mathcal{F}\left(b_{\hat{\mathbf{j}}}\right)$: $\hat{\mathbf{j}}$ magnetic field flux
7. $\nabla \cdot \mathcal{F}\left(b_{\hat{\mathbf{k}}}\right)$: $\hat{\mathbf{k}}$ magnetic field flux
8. $\nabla \cdot \mathcal{F}(\psi)$: correction potential flux

`Vector of Primitive States` (*nodalArray*, 9-components) When combined with an updater that computes $\mathbf{w} = \mathbf{w}(\mathbf{q})$ (e.g. *computePrimitiveState(1d, 2d, 3d)*), the equation systen returns:

0. ρ: mass density
1. $u_{\hat{\mathbf{i}}} = \mathbf{u} \cdot \hat{\mathbf{i}}$: velocity in the $\hat{\mathbf{i}}$ direction
2. $u_{\hat{\mathbf{j}}} = \mathbf{u} \cdot \hat{\mathbf{j}}$: velocity in the $\hat{\mathbf{j}}$ direction
3. $u_{\hat{\mathbf{k}}} = \mathbf{u} \cdot \hat{\mathbf{k}}$: velocity in the $\hat{\mathbf{k}}$ direction
4. $P = \rho\epsilon(\gamma - 1)$: ideal gas pressure
5. $b_{\hat{\mathbf{i}}} = \mathbf{b} \cdot \hat{\mathbf{i}} = \mu_0^{-1/2}\mathbf{B} \cdot \hat{\mathbf{i}}$: magnetic field normalized by permeability of free-space in the $\hat{\mathbf{i}}$ direction
6. $b_{\hat{\mathbf{j}}} = \mathbf{b} \cdot \hat{\mathbf{j}} = \mu_0^{-1/2}\mathbf{B} \cdot \hat{\mathbf{j}}$: magnetic field normalized by permeability of free-space in the $\hat{\mathbf{j}}$ direction
7. $b_{\hat{\mathbf{k}}} = \mathbf{b} \cdot \hat{\mathbf{k}} = \mu_0^{-1/2}\mathbf{B} \cdot \hat{\mathbf{k}}$: magnetic field normalized by permeability of free-space in the $\hat{\mathbf{k}}$ direction
8. ψ: correction potential

`Time Step` (*dynVector*, 1-component) When combined with *timeStepRestrictionUpdater (1d, 2d, 3d)*, the equation system returns the time step consisten with the CFL condition across the entire simulation domain.

`Fastest Wave Speed` (*dynVector*, 1-component) When combined with *hyperbolic (1d, 2d, 3d)*, the equation system returns the fastest wave speed across the entire simulation domain, c_{fast}.

Examples

The following block demonstrates the mhdDednerEqn used in combination with *classicMusclUpdater (1d, 2d, 3d)* to compute $\nabla \cdot \mathcal{F}(\mathbf{w})$ with an externally supplied magnetic field:

```
<Updater hyper>
  kind=classicMuscl1d
  onGrid=domain

  # input nodal component arrays
  in=[q   backgroundB]

  # output nodal component array
  out=[qnew]

  # input dynVector containing fastest wave speed
  waveSpeeds=[waveSpeed]

  # the numerical flux to use
```

```
    numericalFlux= hlldFlux

    # CFL number to use
    cfl=0.3
    # Form of variables to limit
    variableForm= primitive

    # Limiter; one per input nodal component array
    limiter=[minmod   minmod]

    # list of equations to solve
    equations=[mhd]

    <Equation mhd>
      kind=gasDynamicMhdDednerEqn
      gasGamma=1.4
      externalBfield="backgroundB"
    </Equation>

  </Updater>
```

The following block demonstrates the gasDynamicMhdDednerEqn used in combination with *timeStepRestrictionUpdater (1d, 2d, 3d)* and *hyperbolic (1d, 2d, 3d)* to compute c_{fast} with an externally supplied magnetic field:

```
  <Updater getWaveSpeed>
    kind=timeStepRestrictionUpdater1d
    onGrid=domain

    # input nodal component arrays
    in=[q   backgroundB]

    # output dynVector containing fastest wave speed
    waveSpeeds=[waveSpeed]

    # list of equations to compute fastest wave speed for
    restrictions=[idealMhd]

    # courant condition to apply to the timestep
    courantCondition=1.0

    <TimeStepRestriction idealMhd>
      kind=hyperbolic1d
      model=gasDynamicMhdDednerEqn
      gasGamma= 1.4
      externalBfield=True
      includeInTimeStep=False
    </TimeStepRestriction>
  </Updater>
```

simpleTwoTemperatureMhdDednerEqn

Defines the equations of ideal compressible magnetohydrodynamics with divergence cleaning and an electron entropy equation:

$$\begin{aligned}
\frac{\partial \rho}{\partial t} + \nabla \cdot [\rho \, \mathbf{u}] &= 0 \\
\frac{\partial \rho \, \mathbf{u}}{\partial t} + \nabla \cdot \left[\rho \, \mathbf{u} \, \mathbf{u}^T - \mathbf{b} \, \mathbf{b}^T + \mathbb{I} \left(P_{\text{tot}} + \tfrac{1}{2} \left|\mathbf{b}\right|^2\right)\right] &= 0 \\
\frac{\partial E}{\partial t} + \nabla \cdot [(E + P) \, \mathbf{u} + \mathbf{e} \times \mathbf{b}] &= 0 \\
\frac{\partial \mathbf{b}^{\text{plasma}}}{\partial t} + \nabla \times \mathbf{e} + \nabla \psi &= 0 \\
\frac{\partial \psi}{\partial t} + \nabla \cdot \left[c_{\text{fast}}^2 \mathbf{b}\right] &= 0 \\
\frac{\partial S_{\text{electron}}}{\partial t} + \nabla \cdot [S_{\text{electron}} \, \mathbf{u}] &= 0
\end{aligned}$$

Here, $\mathbb{I}$ is the identity matrix, $P_{\text{tot}} = P_{\text{ion}} + P_{\text{electron}} = \rho_{\text{ion}} \epsilon_{\text{ion}} (\gamma_{\text{ion}} - 1) + \rho_{\text{electron}} \epsilon_{\text{electron}} (\gamma_{\text{electron}} - 1)$ is the total plasma pressure, $\epsilon_{\text{ion,electron}}$ is the specific internal energy of ions and electrons and $\gamma_{\text{ion,electron}}$ is the adiabatic index (ratio of specific heats) for the ions and electrons. The quantity c_{fast} corresponds to the fastest wave speed over the entire simulation domain; divergence errors are advected out of the domain with this speed.

In order to track the electron temperature, USim evolves the electron entropy, defined as:

$$S_{\text{electron}} = P_{\text{electron}} n_{\text{electron}}^{-(\gamma_{\text{electron}}+1)}; \;\; n_{\text{electron}} = \frac{\rho}{m_{\text{electron}} + \frac{m_{\text{ion}}}{Z}}$$

Here, n_{electron} is the electron number density, m_{electron} is the electron mass, m_{ion} is the ion mass and Z is the ion charge state. with the fluid velocity, $\mathbf{u}$. In order to advect the electron entropy with the electron velocity, refer to *twoTemperatureMhdDednerEqn*. The method provided by *simpleTwoTemperatureMhdDednerEqn* is generally more robust and has lower computational cost than that provided by *twoTemperatureMhdDednerEqn*. If, for example, heating of electrons by (for example) magnetic dissipation is required, then this can be accomplished by adding source terms of the electron entropy equation, see, e.g. *mhdSrc*.

The electromagnetic fields are defined as:

$$\begin{aligned}
\mathbf{b} &= \mathbf{b}^{\text{plasma}} + \mathbf{b}^{\text{external}} = \mu_0^{-1/2} \left(\mathbf{B}^{\text{plasma}} + \mathbf{B}^{\text{external}}\right) \\
\mathbf{e} &= -\mathbf{u} \times \mathbf{b} + \mathbf{e}^{\text{external}} = \mu_0^{-1/2} \left(-\mathbf{u} \times \mathbf{B} + \mathbf{E}^{\text{external}}\right)
\end{aligned}$$

Here, $\mathbf{b}^{\text{plasma}}$ is the magnetic field induced in the plasma by the inductive electric field, $\mathbf{e}$, while $\mathbf{e}^{\text{external}}$ and $\mathbf{b}^{\text{external}}$ are electromagnetic fields computed "externally" to the ideal magnetohydrodynamic equations.

Parameters

`basementPressure` (float, optional) The minimum pressure allowed. Pressures below this value will be replaced with this value for primitive state, eigensystem and flux computations. Defaults to zero.

`basementDensity` (float, optional) The minimum density allowed. Densities below this value will be replaced with this value for primitive state, eigensystem and flux computations. Defaults to zero.

`gasGamma` (float, optional) Specifies the adiabatic index (ratio of specific heats) for the total pressure, γ. Defaults to 5/3.

`electronGamma` (float, optional) Specifies the adiabatic index (ratio of specific heats) for the electrons, γ_{electron}. Defaults to 5/3.

`electronMass` (float, optional) Specifies the electron mass, m_{electron}. Defaults to $(1836)^{-1}$.

`ionMass` (float, optional) Specifies the ion mass, m_{ion}. Defaults to 1.

`chargeState` (float, optional) Specifies the charge on an ion, Z. Defaults to 1.

`currentVector` (string, required) Specifies the name of the data structure containing the total (ion + electron) plasma current, $\mathbf{J}^{\text{plasma}}$.

`externalEfield` (string, optional) Specifies the name of the data structure containing the externally computed electric field, $\mathbf{e}^{\text{external}}$.

`externalBfield` (string, optional) Specifies the name of the data structure containing the externally computed magnetic field, $\mathbf{b}^{\text{external}}$.

Parent Updater Data

`in` (string vector, required)

`Vector of Conserved Quantities` (*nodalArray*, 10-components, required) The vector of conserved quantities, $\mathbf{q}$ has 10 entries:

0. ρ: mass density
1. $\rho\, u_{\hat{\mathbf{i}}} = \rho\mathbf{u}\cdot\hat{\mathbf{i}}$: momentum density in the $\hat{\mathbf{i}}$ direction
2. $\rho\, u_{\hat{\mathbf{j}}} = \rho\mathbf{u}\cdot\hat{\mathbf{j}}$: momentum density in the $\hat{\mathbf{j}}$ direction
3. $\rho\, u_{\hat{\mathbf{k}}} = \rho\mathbf{u}\cdot\hat{\mathbf{k}}$: momentum density in the $\hat{\mathbf{k}}$ direction
4. $E = \frac{P}{\gamma-1} + \frac{1}{2}\rho|\mathbf{u}|^2 + \frac{1}{2}|\mathbf{b}|^2$: total energy density
5. $b_{\hat{\mathbf{i}}} = \mathbf{b}\cdot\hat{\mathbf{i}} = \mu_0^{-1/2}\mathbf{B}\cdot\hat{\mathbf{i}}$: magnetic field normalized by permeability of free-space in the $\hat{\mathbf{i}}$ direction
6. $b_{\hat{\mathbf{j}}} = \mathbf{b}\cdot\hat{\mathbf{j}} = \mu_0^{-1/2}\mathbf{B}\cdot\hat{\mathbf{j}}$: magnetic field normalized by permeability of free-space in the $\hat{\mathbf{j}}$ direction
7. $b_{\hat{\mathbf{k}}} = \mathbf{b}\cdot\hat{\mathbf{k}} = \mu_0^{-1/2}\mathbf{B}\cdot\hat{\mathbf{k}}$: magnetic field normalized by permeability of free-space in the $\hat{\mathbf{k}}$ direction
8. ψ: correction potential
9. S_{electron}: electron entropy

`Fastest Wave Speed` (*dynVector*, 1-component, required) The fastest wave speed across the entire simulation domain, c_{fast}. Can be computed using *hyperbolic (1d, 2d, 3d)* (see below).

`Externally Computed Electric Field` (*nodalArray*, 3-components, optional) Additional terms in the generalized Ohm's law, $\mathbf{E}^{\text{external}}$, computed "externally" to the ideal magnetohydrodynamic system. The data structure containing $\mathbf{e}^{\text{external}}$ is specified by the "externalEField" option described below.

0. $e^{\text{external}}_{\hat{\mathbf{i}}} = \mathbf{e}^{\text{external}}\cdot\hat{\mathbf{i}} = \mu_0^{-1/2}\mathbf{E}^{\text{external}}\cdot\hat{\mathbf{i}}$: "externally" computed electric field normalized by permeability of free-space in the $\hat{\mathbf{i}}$ direction.

1. $e^{\text{external}}_{\hat{\mathbf{j}}} = \mathbf{e}^{\text{external}} \cdot \hat{\mathbf{j}} = \mu_0^{-1/2} \mathbf{E}^{\text{external}} \cdot \hat{\mathbf{j}}$:"externally" computed electric field normalized by permeability of free-space in the $\hat{\mathbf{j}}$ direction
2. $e^{\text{external}}_{\hat{\mathbf{k}}} = \mathbf{e}^{\text{external}} \cdot \hat{\mathbf{k}} = \mu_0^{-1/2} \mathbf{E}^{\text{external}} \cdot \hat{\mathbf{k}}$: "externally" computed electric field normalized by permeability of free-space in the $\hat{\mathbf{k}}$ direction

`Externally Computed Magnetic Field` (*nodalArray*, **3-components, optional**)
Additional contribution to the magnetic field, $\mathbf{b}^{\text{external}}$, which is not evolved by the induction equation, but does contribute to the Lorentz force and the work done on the plasma. The data structure containing $\mathbf{b}^{\text{external}}$ is specified by the "externalBField" option described below.

0. $b^{\text{external}}_{\hat{\mathbf{i}}} = \mathbf{b}^{\text{external}} \cdot \hat{\mathbf{i}} = \mu_0^{-1/2} \mathbf{B}^{\text{external}} \cdot \hat{\mathbf{i}}$: magnetic field normalized by permeability of free-space in the $\hat{\mathbf{i}}$ direction
1. $b^{\text{external}}_{\hat{\mathbf{j}}} = \mathbf{b}^{\text{external}} \cdot \hat{\mathbf{j}} = \mu_0^{-1/2} \mathbf{B}^{\text{external}} \cdot \hat{\mathbf{j}}$: magnetic field normalized by permeability of free-space in the $\hat{\mathbf{j}}$ direction
2. $b^{\text{external}}_{\hat{\mathbf{k}}} = \mathbf{b}^{\text{external}} \cdot \hat{\mathbf{k}} = \mu_0^{-1/2} \mathbf{B}^{\text{external}} \cdot \hat{\mathbf{k}}$: magnetic field normalized by permeability of free-space in the $\hat{\mathbf{k}}$ direction

`out` **(string vector, required)** For the mhdDednerEqn, one of four output variables are computed, depending on whether the equation is combined with an updater capable of computing fluxes (*classicMusclUpdater (1d, 2d, 3d)*), primitive variables (*computePrimitiveState(1d, 2d, 3d)*), the time step associated with the CFL condition (*timeStepRestrictionUpdater (1d, 2d, 3d)*) or the fastest wave speed in the grid (*hyperbolic (1d, 2d, 3d)*).

`Vector of Fluxes` (*nodalArray*, **9-components**) When combined with an updater that computes $\nabla \cdot \mathcal{F}(\mathbf{w})$ (e.g. *classicMusclUpdater (1d, 2d, 3d)*), the equation system returns:

0. $\nabla \cdot \mathcal{F}(\rho)$: mass flux
1. $\nabla \cdot \mathcal{F}\left(\rho\, u_{\hat{\mathbf{i}}}\right)$: $\hat{\mathbf{i}}$ momentum flux
2. $\nabla \cdot \mathcal{F}\left(\rho\, u_{\hat{\mathbf{j}}}\right)$: $\hat{\mathbf{j}}$ momentum flux
3. $\nabla \cdot \mathcal{F}\left(\rho\, u_{\hat{\mathbf{k}}}\right)$: $\hat{\mathbf{k}}$ momentum flux
4. $\nabla \cdot \mathcal{F}(E)$: total energy flux
5. $\nabla \cdot \mathcal{F}\left(b_{\hat{\mathbf{i}}}\right)$: $\hat{\mathbf{i}}$ magnetic field flux
6. $\nabla \cdot \mathcal{F}\left(b_{\hat{\mathbf{j}}}\right)$: $\hat{\mathbf{j}}$ magnetic field flux
7. $\nabla \cdot \mathcal{F}\left(b_{\hat{\mathbf{k}}}\right)$: $\hat{\mathbf{k}}$ magnetic field flux
8. $\nabla \cdot \mathcal{F}(\psi)$: correction potential flux
9. $\nabla \cdot \mathcal{F}(S_{\text{electron}})$: electron entropy flux

`Vector of Primitive States` (*nodalArray*, **9-components**) When combined with an updater that computes $\mathbf{w} = \mathbf{w}(\mathbf{q})$ (e.g. *computePrimitiveState(1d, 2d, 3d)*), the equation systen returns:

0. ρ: mass density
1. $u_{\hat{\mathbf{i}}} = \mathbf{u} \cdot \hat{\mathbf{i}}$: velocity in the $\hat{\mathbf{i}}$ direction
2. $u_{\hat{\mathbf{j}}} = \mathbf{u} \cdot \hat{\mathbf{j}}$: velocity in the $\hat{\mathbf{j}}$ direction
3. $u_{\hat{\mathbf{k}}} = \mathbf{u} \cdot \hat{\mathbf{k}}$: velocity in the $\hat{\mathbf{k}}$ direction

4. $P = \rho\epsilon(\gamma - 1)$: ideal gas pressure
5. $b_{\hat{\mathbf{i}}} = \mathbf{b} \cdot \hat{\mathbf{i}} = \mu_0^{-1/2}\mathbf{B} \cdot \hat{\mathbf{i}}$: magnetic field normalized by permeability of free-space in the $\hat{\mathbf{i}}$ direction
6. $b_{\hat{\mathbf{j}}} = \mathbf{b} \cdot \hat{\mathbf{j}} = \mu_0^{-1/2}\mathbf{B} \cdot \hat{\mathbf{j}}$: magnetic field normalized by permeability of free-space in the $\hat{\mathbf{j}}$ direction
7. $b_{\hat{\mathbf{k}}} = \mathbf{b} \cdot \hat{\mathbf{k}} = \mu_0^{-1/2}\mathbf{B} \cdot \hat{\mathbf{k}}$: magnetic field normalized by permeability of free-space in the $\hat{\mathbf{k}}$ direction
8. ψ: correction potential
9. P_{electron}: electron pressure

`Time Step` (*dynVector*, 1-component) When combined with *timeStepRestrictionUpdater (1d, 2d, 3d)*, the equation system returns the time step consisten with the CFL condition across the entire simulation domain.

`Fastest Wave Speed` (*dynVector*, 1-component) When combined with *hyperbolic (1d, 2d, 3d)*, the equation system returns the fastest wave speed across the entire simulation domain, c_{fast}.

Examples

The following block demonstrates the simpleTwoTemperatureMhdDednerEqn used in combination with *classicMusclUpdater (1d, 2d, 3d)* to compute $\nabla \cdot \mathcal{F}(\mathbf{w})$

```
 <Updater hyper>
   kind = classicMuscl1d
   onGrid = domain

# input data-structures
   in = [q,electricField]
# output data-structures
   out = [qnew]
# the time integration scheme, rk1 for first order runge-kutta
   timeIntegrationScheme = none
# the numerical flux to use
   numericalFlux = roeFlux
# CFL number to use
   cfl = 0.4
# Form of variables to limit
   variableForm = primitive
# Limiter to use
   limiter = [muscl,muscl]

   waveSpeeds = [waveSpeed]

# list of equations to solve
   equations = [mhd]

   <Equation mhd>
     kind = simpleTwoTemperatureMhdDednerEqn
     gasGamma = GAS_GAMMA
     electronGamma = $ELECTRON_GAMMA$
     basementDensity = $BASEMENT_DENSITY$
     basementPressure = $BASEMENT_PRESSURE$
     externalEfield = "electricField"
   </Equation>
```

```
 </Updater>
```

The following block demonstrates the simpleTwoTemperatureMhdDednerEqn used in combination with *computePrimitiveState(1d, 2d, 3d)* to compute $\mathbf{w}\left(\mathbf{q}\right)$

```
 <Updater computePrimitiveState>
   kind = computePrimitiveState1d

   onGrid = domain
# input data-structures
   in = [q,electricField]

# ouput data-structures
   out = [w]

   <Equation mhd>
     kind = simpleTwoTemperatureMhdDednerEqn
     gasGamma = GAS_GAMMA
     electronGamma = $ELECTRON_GAMMA$
     basementDensity = $BASEMENT_DENSITY$
     basementPressure = $BASEMENT_PRESSURE$
     externalEfield = "electricField"
   </Equation>

 </Updater>
```

The following block demonstrates the simpleTwoTemperatureMhdDednerEqn used in combination with *timeStepRestrictionUpdater (1d, 2d, 3d)*, *hyperbolic (1d, 2d, 3d)* and *quadratic (1d, 2d, 3d)* to compute dt_{min}, dt_{diff} and c_{fast} for resistive two-temperature MHD:

```
<Updater getHypDT>
 kind = timeStepRestrictionUpdater1d
  in = [q,electricField]
 onGrid = domain
 waveSpeeds = [waveSpeed]
 timeSteps = [diffDT]
 restrictions = [idealMhd]
 courantCondition = CFL

 <TimeStepRestriction idealMhd>
   kind = hyperbolic1d
   cfl = CFL
   model = simpleTwoTemperatureMhdDednerEqn
   gasGamma = GAS_GAMMA
   electronGamma = $ELECTRON_GAMMA$
   correctNans = true
   correct = true
   correctNans = true
   basementDensity = $BASEMENT_DENSITY$
   basementPressure = $BASEMENT_PRESSURE$
   externalEfield = "electricField"
   storeTimeStep = False
 </TimeStepRestriction>

</Updater>
```

twoTemperatureMhdDednerEqn

Defines the equations of ideal compressible magnetohydrodynamics with divergence cleaning and an electron entropy equation:

$$\begin{aligned}
\frac{\partial \rho}{\partial t} + \nabla \cdot [\rho \, \mathbf{u}] &= 0 \\
\frac{\partial \rho \, \mathbf{u}}{\partial t} + \nabla \cdot \left[\rho \, \mathbf{u} \, \mathbf{u}^T - \mathbf{b} \, \mathbf{b}^T + \mathbb{I} \left(P_{\text{tot}} + \tfrac{1}{2} \left|\mathbf{b}\right|^2\right)\right] &= 0 \\
\frac{\partial E}{\partial t} + \nabla \cdot [(E + P) \, \mathbf{u} + \mathbf{e} \times \mathbf{b}] &= 0 \\
\frac{\partial \mathbf{b}^{\text{plasma}}}{\partial t} + \nabla \times \mathbf{e} + \nabla \psi &= 0 \\
\frac{\partial \psi}{\partial t} + \nabla \cdot \left[c_{\text{fast}}^2 \mathbf{b}\right] &= 0 \\
\frac{\partial S_{\text{electron}}}{\partial t} + \nabla \cdot [S_{\text{electron}} \, \mathbf{u}_{\text{electron}}] &= 0
\end{aligned}$$

Here, $\mathbb{I}$ is the identity matrix, $P_{\text{tot}} = P_{\text{ion}} + P_{\text{electron}} = \rho_{\text{ion}} \epsilon_{\text{ion}} (\gamma_{\text{ion}} - 1) + \rho_{\text{electron}} \epsilon_{\text{electron}} (\gamma_{\text{electron}} - 1)$ is the total plasma pressure, $\epsilon_{\text{ion,electron}}$ is the specific internal energy of ions and electrons and $\gamma_{\text{ion,electron}}$ is the adiabatic index (ratio of specific heats) for the ions and electrons. The quantity c_{fast} corresponds to the fastest wave speed over the entire simulation domain; divergence errors are advected out of the domain with this speed.

In order to track the electron temperature, USim evolves the electron entropy, defined as:

$$S_{\text{electron}} = P_{\text{electron}} n_{\text{electron}}^{-(\gamma_{\text{electron}}+1)}; \;\; n_{\text{electron}} = \frac{\rho}{m_{\text{electron}} + \frac{m_{\text{ion}}}{Z}}$$

Here, n_{electron} is the electron number density, m_{electron} is the electron mass, m_{ion} is the ion mass and Z is the ion charge state. The electron entropy is advected by the electron velocity, $\mathbf{u}_{\text{electron}}$, computed as:

$$\mathbf{u}_{\text{electron}} = -\frac{\mathbf{J}^{\text{plasma}} - q Z m_{\text{ion}}^{-1} \rho \mathbf{u}}{q n_{\text{electron}}}; \;\; \mathbf{J}^{\text{plasma}} = \mu_0^{-1/2} \nabla \times \mathbf{b}^{\text{plasma}} = \mu_0^{-1} \nabla \times \mathbf{B}^{\text{plasma}}$$

Here, $\mathbf{J}^{\text{plasma}}$ is the total (ion+electron) plasma current and q is the fundamental change ($-q$ is the charge on an electron). As defined above, the electron entropy is advected with the electron density. If, for example, heating of electrons by (for example) magnetic dissipation is required, then this can be accomplished by adding source terms of the electron entropy equation, see, e.g. *mhdSrc*.

The electromagnetic fields are defined as:

$$\begin{aligned}
\mathbf{b} &= \mathbf{b}^{\text{plasma}} + \mathbf{b}^{\text{external}} = \mu_0^{-1/2} \left(\mathbf{B}^{\text{plasma}} + \mathbf{B}^{\text{external}}\right) \\
\mathbf{e} &= -\mathbf{u} \times \mathbf{b} + \mathbf{e}^{\text{external}} = \mu_0^{-1/2} \left(-\mathbf{u} \times \mathbf{B} + \mathbf{E}^{\text{external}}\right)
\end{aligned}$$

Here, $\mathbf{b}^{\text{plasma}}$ is the magnetic field induced in the plasma by the inductive electric field, $\mathbf{e}$, while $\mathbf{e}^{\text{external}}$ and $\mathbf{b}^{\text{external}}$ are electromagnetic fields computed "externally" to the ideal magnetohydrodynamic equations.

Parameters

`basementPressure` (float, optional) The minimum pressure allowed. Pressures below this value will be replaced with this value for primitive state, eigensystem and flux computations. Defaults to zero.

`basementDensity` (float, optional) The minimum density allowed. Densities below this value will be replaced with this value for primitive state, eigensystem and flux computations. Defaults to zero.

`gasGamma` (float, optional) Specifies the adiabatic index (ratio of specific heats) for the total pressure, γ. Defaults to 5/3.

`electronGamma` (float, optional) Specifies the adiabatic index (ratio of specific heats) for the electrons, γ_{electron}. Defaults to 5/3.

`electronMass` (float, optional) Specifies the electron mass, m_{electron}. Defaults to $(1836)^{-1}$.

`ionMass` (float, optional) Specifies the ion mass, m_{ion}. Defaults to 1.

`chargeState` (float, optional) Specifies the charge on an ion, Z. Defaults to 1.

`currentVector` (string, required) Specifies the name of the data structure containing the total (ion + electron) plasma current, $\mathbf{J}^{\text{plasma}}$.

`externalEfield` (string, optional) Specifies the name of the data structure containing the externally computed electric field, $\mathbf{e}^{\text{external}}$.

`externalBfield` (string, optional) Specifies the name of the data structure containing the externally computed magnetic field, $\mathbf{b}^{\text{external}}$.

Parent Updater Data

`in` (string vector, required)

`Vector of Conserved Quantities` (*nodalArray*, 10-components, required) The vector of conserved quantities, $\mathbf{q}$ has 10 entries:

0. ρ: mass density
1. $\rho\, u_{\hat{\mathbf{i}}} = \rho\mathbf{u}\cdot\hat{\mathbf{i}}$: momentum density in the $\hat{\mathbf{i}}$ direction
2. $\rho\, u_{\hat{\mathbf{j}}} = \rho\mathbf{u}\cdot\hat{\mathbf{j}}$: momentum density in the $\hat{\mathbf{j}}$ direction
3. $\rho\, u_{\hat{\mathbf{k}}} = \rho\mathbf{u}\cdot\hat{\mathbf{k}}$: momentum density in the $\hat{\mathbf{k}}$ direction
4. $E = \frac{P}{\gamma-1} + \frac{1}{2}\rho|\mathbf{u}|^2 + \frac{1}{2}|\mathbf{b}|^2$: total energy density
5. $b_{\hat{\mathbf{i}}} = \mathbf{b}\cdot\hat{\mathbf{i}} = \mu_0^{-1/2}\mathbf{B}\cdot\hat{\mathbf{i}}$: magnetic field normalized by permeability of free-space in the $\hat{\mathbf{i}}$ direction
6. $b_{\hat{\mathbf{j}}} = \mathbf{b}\cdot\hat{\mathbf{j}} = \mu_0^{-1/2}\mathbf{B}\cdot\hat{\mathbf{j}}$: magnetic field normalized by permeability of free-space in the $\hat{\mathbf{j}}$ direction
7. $b_{\hat{\mathbf{k}}} = \mathbf{b}\cdot\hat{\mathbf{k}} = \mu_0^{-1/2}\mathbf{B}\cdot\hat{\mathbf{k}}$: magnetic field normalized by permeability of free-space in the $\hat{\mathbf{k}}$ direction
8. ψ: correction potential
9. S_{electron}: electron entropy

`Current Density` (*nodalArray*, 3-components, required) The total (ion and electron) current in the plasma, typically calculated from from pre-Maxwell form of Ampere's law, $\mathbf{J}^{\text{plasma}} = \mu_0^{1/2}\nabla\times\mathbf{b}^{\text{plasma}}$, which can be computed through, e.g. *vector (1d, 2d, 3d)*. The data structure containing $\mathbf{J}^{\text{plasma}}$ is specified by the "currentVector" option described below.

0. $J_{\hat{\mathbf{i}}}^{\text{plasma}} = \mathbf{J}^{\text{plasma}}\cdot\hat{\mathbf{i}}$: total (ion and electron) current in the plasma in the $\hat{\mathbf{i}}$ direction.
1. $J_{\hat{\mathbf{j}}}^{\text{plasma}} = \mathbf{J}^{\text{plasma}}\cdot\hat{\mathbf{j}}$: total (ion and electron) current in the plasma in the $\hat{\mathbf{j}}$ direction

2. $J^{\text{plasma}}_{\hat{\mathbf{k}}} = \mathbf{J}^{\text{plasma}} \cdot \hat{\mathbf{k}}$: total (ion and electron) current in the plasma in the $\hat{\mathbf{k}}$ direction

`Fastest Wave Speed` **(*dynVector*, 1-component, required)** The fastest wave speed across the entire simulation domain, c_{fast}. Can be computed using *hyperbolic (1d, 2d, 3d)* (see below).

`Externally Computed Electric Field` **(*nodalArray*, 3-components, optional)** Additional terms in the generalized Ohm's law, $\mathbf{E}^{\text{external}}$, computed "externally" to the ideal magnetohydrodynamic system. The data structure containing $\mathbf{e}^{\text{external}}$ is specified by the "externalEField" option described below.

0. $e^{\text{external}}_{\hat{\mathbf{i}}} = \mathbf{e}^{\text{external}} \cdot \hat{\mathbf{i}} = \mu_0^{-1/2}\mathbf{E}^{\text{external}} \cdot \hat{\mathbf{i}}$: "externally" computed electric field normalized by permeability of free-space in the $\hat{\mathbf{i}}$ direction.
1. $e^{\text{external}}_{\hat{\mathbf{j}}} = \mathbf{e}^{\text{external}} \cdot \hat{\mathbf{j}} = \mu_0^{-1/2}\mathbf{E}^{\text{external}} \cdot \hat{\mathbf{j}}$:"externally" computed electric field normalized by permeability of free-space in the $\hat{\mathbf{j}}$ direction
2. $e^{\text{external}}_{\hat{\mathbf{k}}} = \mathbf{e}^{\text{external}} \cdot \hat{\mathbf{k}} = \mu_0^{-1/2}\mathbf{E}^{\text{external}} \cdot \hat{\mathbf{k}}$: "externally" computed electric field normalized by permeability of free-space in the $\hat{\mathbf{k}}$ direction

`Externally Computed Magnetic Field` **(*nodalArray*, 3-components, optional)** Additional contribution to the magnetic field, $\mathbf{b}^{\text{external}}$, which is not evolved by the induction equation, but does contribute to the Lorentz force and the work done on the plasma. The data structure containing $\mathbf{b}^{\text{external}}$ is specified by the "externalBField" option described below.

0. $b^{\text{external}}_{\hat{\mathbf{i}}} = \mathbf{b}^{\text{external}} \cdot \hat{\mathbf{i}} = \mu_0^{-1/2}\mathbf{B}^{\text{external}} \cdot \hat{\mathbf{i}}$: magnetic field normalized by permeability of free-space in the $\hat{\mathbf{i}}$ direction
1. $b^{\text{external}}_{\hat{\mathbf{j}}} = \mathbf{b}^{\text{external}} \cdot \hat{\mathbf{j}} = \mu_0^{-1/2}\mathbf{B}^{\text{external}} \cdot \hat{\mathbf{j}}$: magnetic field normalized by permeability of free-space in the $\hat{\mathbf{j}}$ direction
2. $b^{\text{external}}_{\hat{\mathbf{k}}} = \mathbf{b}^{\text{external}} \cdot \hat{\mathbf{k}} = \mu_0^{-1/2}\mathbf{B}^{\text{external}} \cdot \hat{\mathbf{k}}$: magnetic field normalized by permeability of free-space in the $\hat{\mathbf{k}}$ direction

`out` **(string vector, required)** For the mhdDednerEqn, one of four output variables are computed, depending on whether the equation is combined with an updater capable of computing fluxes (*classicMusclUpdater (1d, 2d, 3d)*), primitive variables (*computePrimitiveState(1d, 2d, 3d)*), the time step associated with the CFL condition (*timeStepRestrictionUpdater (1d, 2d, 3d)*) or the fastest wave speed in the grid (*hyperbolic (1d, 2d, 3d)*).

`Vector of Fluxes` **(*nodalArray*, 9-components)** When combined with an updater that computes $\nabla \cdot \mathcal{F}(\mathbf{w})$ (e.g. *classicMusclUpdater (1d, 2d, 3d)*), the equation system returns:

0. $\nabla \cdot \mathcal{F}(\rho)$: mass flux
1. $\nabla \cdot \mathcal{F}\left(\rho\, u_{\hat{\mathbf{i}}}\right)$: $\hat{\mathbf{i}}$ momentum flux
2. $\nabla \cdot \mathcal{F}\left(\rho\, u_{\hat{\mathbf{j}}}\right)$: $\hat{\mathbf{j}}$ momentum flux
3. $\nabla \cdot \mathcal{F}\left(\rho\, u_{\hat{\mathbf{k}}}\right)$: $\hat{\mathbf{k}}$ momentum flux
4. $\nabla \cdot \mathcal{F}(E)$: total energy flux
5. $\nabla \cdot \mathcal{F}\left(b_{\hat{\mathbf{i}}}\right)$: $\hat{\mathbf{i}}$ magnetic field flux
6. $\nabla \cdot \mathcal{F}\left(b_{\hat{\mathbf{j}}}\right)$: $\hat{\mathbf{j}}$ magnetic field flux
7. $\nabla \cdot \mathcal{F}\left(b_{\hat{\mathbf{k}}}\right)$: $\hat{\mathbf{k}}$ magnetic field flux
8. $\nabla \cdot \mathcal{F}(\psi)$: correction potential flux

9. $\nabla \cdot \mathcal{F}\left(S_{\text{electron}}\right)$: electron entropy flux

`Vector of Primitive States` (*nodalArray*, 9-components) When combined with an updater that computes $\mathbf{w} = \mathbf{w}(\mathbf{q})$ (e.g. *computePrimitiveState(1d, 2d, 3d)*), the equation systen returns:

0. ρ: mass density
1. $u_{\hat{\mathbf{i}}} = \mathbf{u} \cdot \hat{\mathbf{i}}$: velocity in the $\hat{\mathbf{i}}$ direction
2. $u_{\hat{\mathbf{j}}} = \mathbf{u} \cdot \hat{\mathbf{j}}$: velocity in the $\hat{\mathbf{j}}$ direction
3. $u_{\hat{\mathbf{k}}} = \mathbf{u} \cdot \hat{\mathbf{k}}$: velocity in the $\hat{\mathbf{k}}$ direction
4. $P = \rho\epsilon(\gamma - 1)$: ideal gas pressure
5. $b_{\hat{\mathbf{i}}} = \mathbf{b} \cdot \hat{\mathbf{i}} = \mu_0^{-1/2}\mathbf{B} \cdot \hat{\mathbf{i}}$: magnetic field normalized by permeability of free-space in the $\hat{\mathbf{i}}$ direction
6. $b_{\hat{\mathbf{j}}} = \mathbf{b} \cdot \hat{\mathbf{j}} = \mu_0^{-1/2}\mathbf{B} \cdot \hat{\mathbf{j}}$: magnetic field normalized by permeability of free-space in the $\hat{\mathbf{j}}$ direction
7. $b_{\hat{\mathbf{k}}} = \mathbf{b} \cdot \hat{\mathbf{k}} = \mu_0^{-1/2}\mathbf{B} \cdot \hat{\mathbf{k}}$: magnetic field normalized by permeability of free-space in the $\hat{\mathbf{k}}$ direction
8. ψ: correction potential
9. P_{electron}: electron pressure

`Time Step` (*dynVector*, 1-component) When combined with *timeStepRestrictionUpdater (1d, 2d, 3d)*, the equation system returns the time step consisten with the CFL condition across the entire simulation domain.

`Fastest Wave Speed` (*dynVector*, 1-component) When combined with *hyperbolic (1d, 2d, 3d)*, the equation system returns the fastest wave speed across the entire simulation domain, c_{fast}.

Examples

The following block demonstrates the twoTemperatureMhdDednerEqn used in combination with *classicMusclUpdater (1d, 2d, 3d)* to compute $\nabla \cdot \mathcal{F}\left(\mathbf{w}\right)$, including resistive effects

```
 <Updater hyper>
   kind = classicMuscl1d
   onGrid = domain

# input data-structures
   in = [q,electricField,current,chargeState,resistivity]
# output data-structures
   out = [qnew]
# the time integration scheme, rk1 for first order runge-kutta
   timeIntegrationScheme = none
# the numerical flux to use
   numericalFlux = roeFlux
# CFL number to use
   cfl = 0.4
# Form of variables to limit
   variableForm = primitive
# Limiter to use
   limiter = [muscl,muscl,muscl,muscl,muscl]

   waveSpeeds = [waveSpeed]
```

```
# list of equations to solve
   equations = [mhd]

# list of sources to add
   source = [mhdSource]

   <Equation mhd>
     kind = twoTemperatureMhdDednerEqn
     gasGamma = GAS_GAMMA
     electronGamma = $ELECTRON_GAMMA$
     basementDensity = $BASEMENT_DENSITY$
     basementPressure = $BASEMENT_PRESSURE$
     externalEfield = "electricField"
     currentVector = "current"
   </Equation>

   <Source mhdSource>
     kind = mhdSrc
     model = twoTemperatureMhdDednerEqn
     externalEfield = true
     inputVariables = [q, electricField,current,chargeState,resistivity]
     ionMass = ION_MASS
     fundamentalCharge = FUNDAMENTAL_CHARGE
   </Source>

 </Updater>
```

The following block demonstrates the twoTemperatureMhdDednerEqn used in combination with *computePrimitiveState(1d, 2d, 3d)* to compute $\mathbf{w}\left(\mathbf{q}\right)$

```
 <Updater computePrimitiveState>
   kind = computePrimitiveState1d

   onGrid = domain
# input data-structures
   in = [q,electricField,current,chargeState,resistivity]

# ouput data-structures
   out = [w]

   <Equation mhd>
     kind = twoTemperatureMhdDednerEqn
     gasGamma = GAS_GAMMA
     electronGamma = $ELECTRON_GAMMA$
     basementDensity = $BASEMENT_DENSITY$
     basementPressure = $BASEMENT_PRESSURE$
     externalEfield = "electricField"
     currentVector = "current"
   </Equation>

 </Updater>
```

The following block demonstrates the twoTemperatureMhdDednerEqn used in combination with *timeStepRestrictionUpdater (1d, 2d, 3d)*, *hyperbolic (1d, 2d, 3d)* and *quadratic (1d, 2d, 3d)* to compute $dt_{\min}$, dt_{diff} and c_{fast} for resistive two-temperature MHD:

```
 <Updater getHypDT>
  kind = timeStepRestrictionUpdater1d
```

```
    in = [q,electricField,current,chargeState,resistivity]
  onGrid = domain
  waveSpeeds = [waveSpeed]
  timeSteps = [diffDT]
  restrictions = [idealMhd,quadratic]
  courantCondition = CFL

  <TimeStepRestriction idealMhd>
    kind = hyperbolic1d
    cfl = CFL
    model = twoTemperatureMhdDednerEqn
    gasGamma = GAS_GAMMA
    electronGamma = $ELECTRON_GAMMA$
    correctNans = true
    correct = true
    correctNans = true
    basementDensity = $BASEMENT_DENSITY$
    basementPressure = $BASEMENT_PRESSURE$
    externalEfield = "electricField"
    currentVector = "current"
    storeTimeStep = False
  </TimeStepRestriction>

  <TimeStepRestriction quadratic>
    kind = quadratic1d
    in = [resistivity]
    cfl = CFL
  </TimeStepRestriction>
</Updater>
```

maxwellDednerEqn

Fluxes and eigensystem for Maxwell's equations in vacuum with divergence cleaning.

$$\frac{\partial \mathbf{E}}{\partial t} + c^2 \nabla \times \mathbf{B} + \nabla \Phi = 0$$
$$\frac{\partial \mathbf{B}}{\partial t} - \nabla \times \mathbf{E} + \nabla \psi = 0$$
$$\frac{\partial \Phi}{\partial t} + \nabla \cdot \left[c_{\text{fast}}^2 \mathbf{E} \right] = 0$$
$$\frac{\partial \psi}{\partial t} + \nabla \cdot \left[c_{\text{fast}}^2 \mathbf{B} \right] = 0$$

Coupling of Maxwell's equations to a plasma is accomplished using *current*.

Parameters

`mu0` (float, optional) Permeability of free space. Default value is 1.256e-06.

`epsilon0` (float, optional) Permittivity of free space. Default value is 8.854e-12.

`cfl` (float, optional) CFL number. Default value is 1.0.

Parent Updater Data

`in` (string vector, required)

`Vector of Conserved Quantities` (*nodalArray*, 8-components, required) The vector of conserved quantities, $\mathbf{q}$ has 8 entries:

0. $E_{\hat{\mathbf{i}}} = \mathbf{E} \cdot \hat{\mathbf{i}}$: electric field in the $\hat{\mathbf{i}}$ direction.
1. $E_{\hat{\mathbf{j}}} = \mathbf{E} \cdot \hat{\mathbf{j}}$: electric field in the $\hat{\mathbf{j}}$ direction
2. $E_{\hat{\mathbf{k}}} = \mathbf{E} \cdot \hat{\mathbf{k}}$: electric field in the $\hat{\mathbf{k}}$ direction
3. $B_{\hat{\mathbf{i}}} = \mathbf{B} \cdot \hat{\mathbf{i}}$: magnetic field in the $\hat{\mathbf{i}}$ direction
4. $B_{\hat{\mathbf{j}}} = \mathbf{B} \cdot \hat{\mathbf{j}}$: magnetic field in the $\hat{\mathbf{j}}$ direction
5. $B_{\hat{\mathbf{k}}} = \mathbf{B} \cdot \hat{\mathbf{k}}$: magnetic field in the $\hat{\mathbf{k}}$ direction
6. Φ electric field correction potential
7. Ψ magnetic field correction potential

`Fastest Wave Speed` (*dynVector*, 1-component, required) The fastest wave speed across the entire simulation domain, c_{fast}. Can be computed using *hyperbolic (1d, 2d, 3d)* (see below).

`out` (string vector, required) For the maxwellDednerEqn, one of three output variables are computed, depending on whether the equation is combined with an updater capable of computing fluxes (*classicMusclUpdater (1d, 2d, 3d)*), the time step associated with the CFL condition (*timeStepRestrictionUpdater (1d, 2d, 3d)*) or the fastest wave speed in the grid (*hyperbolic (1d, 2d, 3d)*).

`Vector of Fluxes` (*nodalArray*, 9-components) When combined with an updater that computes $\nabla \cdot \mathcal{F}(\mathbf{w})$ (e.g. *classicMusclUpdater (1d, 2d, 3d)*), the equation system returns:

0. $\nabla \cdot \mathcal{F}\left(E_{\hat{\mathbf{i}}}\right)$: $\hat{\mathbf{i}}$ electric field flux
1. $\nabla \cdot \mathcal{F}\left(E_{\hat{\mathbf{j}}}\right)$: $\hat{\mathbf{i}}$ electric field flux
2. $\nabla \cdot \mathcal{F}\left(E_{\hat{\mathbf{k}}}\right)$: $\hat{\mathbf{j}}$ electric field flux
3. $\nabla \cdot \mathcal{F}\left(B_{\hat{\mathbf{i}}}\right)$: $\hat{\mathbf{i}}$ magnetic field flux
4. $\nabla \cdot \mathcal{F}\left(B_{\hat{\mathbf{j}}}\right)$: $\hat{\mathbf{i}}$ magnetic field flux
5. $\nabla \cdot \mathcal{F}\left(B_{\hat{\mathbf{k}}}\right)$: $\hat{\mathbf{j}}$ magnetic field flux
6. $\nabla \cdot \mathcal{F}(\psi)$: electric correction potential flux
7. $\nabla \cdot \mathcal{F}(\psi)$: magnetic correction potential flux

`Time Step` (*dynVector*, 1-component) When combined with *timeStepRestrictionUpdater (1d, 2d, 3d)*, the equation system returns the time step consisten with the CFL condition across the entire simulation domain.

`Fastest Wave Speed` (*dynVector*, 1-component) When combined with *hyperbolic (1d, 2d, 3d)*, the equation system returns the fastest wave speed across the entire simulation domain, c_{fast}.

Example

The following block demonstrates the maxwellDednerEqn used in combination with *classicMusclUpdater (1d, 2d, 3d)* to compute $\nabla \cdot \mathcal{F}(\mathbf{w})$:

```
<Updater hyper>
  kind=classicMuscl2d
  onGrid=domain
```

```
  timeIntegrationScheme=none
  numericalFlux=hlleFlux
  limiter=[none]
  variableForm=conservative
  preservePositivity=false
  in=[q]
  out=[qNew]
  waveSpeeds=[waveSpeed]
  cfl=0.4
  equations=[maxwell]

  <Equation maxwell>
    kind=maxwellDednerEqn
    epsilon0=1.0
    mu0=1.0
    cfl=0.4
  </Equation>

</Updater>
```

The following block demonstrates the maxwellDednerEqn used in combination with *timeStepRestrictionUpdater (1d, 2d, 3d)* and *hyperbolic (1d, 2d, 3d)* to compute c_{fast}:

```
<Updater getWaveSpeed>
  kind=timeStepRestrictionUpdater2d
  in=[q]
  waveSpeeds=[waveSpeed]
  onGrid=domain
  restrictions=[hyperbolic]
  cfl=0.4
  courantCondition=0.4

  <TimeStepRestriction hyperbolic>
    kind=hyperbolic2d
    model=maxwellEqn
    cfl=0.4
    c0=1.0
    gamma=0.0
    chi=0.0
    includeInTimeStep=False
  </TimeStepRestriction>

</Updater>
```

gasDynamicMaxwellDednerEqn

Defines the equations of inviscid fluid dynamics coupled to Maxwell's equations in source term form with divergence cleaning:

$$\begin{aligned}
\frac{\partial \rho}{\partial t} + \nabla \cdot [\rho\, \mathbf{u}] &= 0 \\
\frac{\partial (\rho \mathbf{u})}{\partial t} + \nabla \cdot \left[\rho\, \mathbf{u}\, \mathbf{u}^T + \mathbb{I} P\right] &= \sum_{\text{species}} \left(q^{\text{species}} \mathbf{E} + \mathbf{J}^{\text{species}} \times \mathbf{B}\right) \\
\frac{\partial E}{\partial t} + \nabla \cdot [(E + P)\, \mathbf{u}] &= \sum_{\text{species}} \mathbf{J}^{\text{species}} \cdot \mathbf{E} \\
\frac{\partial \mathbf{B}^{\text{plasma}}}{\partial t} + \nabla \times \mathbf{E} + \nabla \Psi &= 0 \\
\frac{\partial \mathbf{E}^{\text{plasma}}}{\partial t} - c^2 \nabla \times \mathbf{B} + \nabla \Phi &= -\epsilon_0^{-1} \sum_{\text{species}} \mathbf{J}^{\text{species}} \\
\frac{\partial \Psi}{\partial t} + \nabla \cdot \left[c_{\text{fast}}^2 \mathbf{B}^{\text{plasma}}\right] &= 0 \\
\frac{\partial \Phi}{\partial t} + \nabla \cdot \left[c_{\text{fast}}^2 \mathbf{E}^{\text{plasma}}\right] &= \sum_{\text{species}} q^{\text{species}}
\end{aligned}$$

Here, q^{species} is the species charge density, $\mathbf{J}^{\text{species}}$ is the species current density, $\mathbb{I}$ is the identity matrix, $P = \rho\epsilon(\gamma - 1)$ is the pressure of an ideal gas, ϵ is the specific internal energy and γ is the adiabatic index (ratio of specific heats). The quantity c_{fast} corresponds to the fastest wave speed over the entire simulation domain; divergence errors are advected out of the domain with this speed.

In order to integrate these equations, USim casts them into flux-conservative form using the following standard identities (note that the use of these identities does not require an assumption of quasi-neutrality):

$$\begin{aligned}
\sum_{\text{species}} \left(q^{\text{species}} \mathbf{E} + \mathbf{J}^{\text{species}} \times \mathbf{B}\right) &= -\frac{\partial c^{-2} \mathbf{S}^{\text{EM}}}{\partial t} + \nabla \cdot \mathcal{T}^{\text{EM}} \\
\sum_{\text{species}} \mathbf{J}^{\text{species}} \cdot \mathbf{E} &= -\frac{\partial E^{\text{EM}}}{\partial t} - \nabla \cdot \mathbf{S}^{\text{EM}}
\end{aligned}$$

Here, $\mathcal{T}^{\text{EM}}$ is the electromagnetic stress tensor and $\mathbf{S}^{\text{EM}}$ is the electromagnetic energy (Poynting) flux vector, which are defined as:

$$\begin{aligned}
\mathcal{T}^{\text{EM}} = \frac{1}{\mu_0}\left(\frac{\mathbf{E}\mathbf{E}^T}{c^2} + \mathbf{B}\mathbf{B}^T\right) + \mathbb{I} E_{\text{EM}} &= \frac{\mathbf{e}\mathbf{e}^T}{c^2} + \mathbf{b}\mathbf{b}^T + \mathbb{I} E_{\text{EM}} \\
\mathbf{S}^{\text{EM}} = \mu_0^{-1} \mathbf{E} \times \mathbf{B} &= \mathbf{e} \times \mathbf{b} \\
E^{\text{EM}} = \frac{1}{2\mu_0}\left(\frac{|\mathbf{E}|^2}{c^2} + |\mathbf{B}|^2\right) &= \frac{1}{2}\left(\frac{|\mathbf{e}|^2}{c^2} + |\mathbf{b}|^2\right)
\end{aligned}$$

Here, E^{EM} is the electromagnetic energy density and the electromagnetic fields are defined as:

$$\begin{aligned}
\mathbf{b} = \mathbf{b}^{\text{plasma}} + \mathbf{b}^{\text{external}} &= \mu_0^{-1/2}\left(\mathbf{B}^{\text{plasma}} + \mathbf{B}^{\text{external}}\right) = \mu_0^{-1/2} \mathbf{B} \\
\mathbf{e} = \mathbf{e}^{\text{plasma}} + \mathbf{e}^{\text{external}} &= \mu_0^{-1/2}\left(\mathbf{E}^{\text{plasma}} + \mathbf{E}^{\text{external}}\right) = \mu_0^{-1/2} \mathbf{E}
\end{aligned}$$

Here, $\mathbf{b}^{\text{plasma}}$ is the magnetic field induced in the plasma, $\mathbf{e}^{\text{plasma}}$ is the electric field associated with net charge in the plasma, while $\mathbf{e}^{\text{external}}$ and $\mathbf{b}^{\text{external}}$ are electromagnetic fields computed "externally"

to Maxwell's equations inside the plasma. With these identitifications, the fluid part of the gasDynamicMaxwellDednerEqn takes the form:

$$\frac{\partial \rho}{\partial t} + \nabla \cdot [\rho \, \mathbf{u}] = 0$$
$$\frac{\partial \left(\rho \mathbf{u} + c^{-2}\mathbf{S}^{\mathrm{EM}}\right)}{\partial t} + \nabla \cdot \left[\rho \, \mathbf{u} \, \mathbf{u}^T + \mathbb{I}P - \mathcal{T}^{\mathrm{EM}}\right] = 0$$
$$\frac{\partial \left(E + E^{\mathrm{EM}}\right)}{\partial t} + \nabla \cdot \left[(E+P)\,\mathbf{u} + \mathbf{S}^{\mathrm{EM}}\right] = 0$$

The electromagnetic part of the system is solved in USim as:

$$\frac{\partial \mathbf{b}^{\mathrm{plasma}}}{\partial t} - \nabla \times \mathbf{e} + \nabla \psi = 0$$
$$\frac{\partial \mathbf{e}^{\mathrm{plasma}}}{\partial t} + f^2 c_{\mathrm{fast}}^2 \nabla \times \mathbf{b} + \nabla \phi = -f^2 c_{\mathrm{fast}}^2 \mu_0^{1/2} \sum_{\mathrm{species}} \mathbf{J}^{\mathrm{species}}$$
$$\frac{\partial \psi}{\partial t} + \nabla \cdot \left[c_{\mathrm{fast}}^2 \mathbf{b}^{\mathrm{plasma}}\right] = 0$$
$$\frac{\partial \phi}{\partial t} + \nabla \cdot \left[c_{\mathrm{fast}}^2 \mathbf{e}^{\mathrm{plasma}}\right] = \mu_0^{-1/2} \sum_{\mathrm{species}} q^{\mathrm{species}}$$

Here, we have written $c^2 = f^2 c_{\mathrm{fast}}^2 = (\epsilon_0 \mu_0)^{-1}$, where f is a dimensionless number that defines the ratio of the speed of light to the fatest wave in the mesh and we have further defined $\psi = \mu_0^{-1/2}\Psi$ and $\Phi = \mu_0^{-1/2}\Phi$.

In order to close the electromagnetic part of the equations, a model for the current density and charge is required. An example of such a model that is provided with USim is *mhdSrc*. However, the user is also free to construct their own closure that returns:

$$\mu_0^{-1/2} \sum_{\mathrm{species}} q^{\mathrm{species}}; \;\; \mu_0^{1/2} \sum_{\mathrm{species}} \mathbf{J}^{\mathrm{species}}$$

Parameters

`lightSpeed` (float, optional) The speed of light in m/s. Used to specify the speed of light in the fluid momentum and energy equations. Defaults to 2.99792458e8.

`lightSpeedFactor` (float, optional) Dimensionless number, used to specify the ratio of the speed of light to the fastest wave speed in the grid. Defaults to 1.0e3.

`basementPressure` (float, optional) The minimum pressure allowed. Pressures below this value will be replaced with this value for primitive state, eigensystem and flux computations. Defaults to zero.

`basementDensity` (float, optional) The minimum density allowed. Densities below this value will be replaced with this value for primitive state, eigensystem and flux computations. Defaults to zero.

`gasGamma` (float, optional) Specifies the adiabatic index (ratio of specific heats), γ. Defaults to 5/3.

`externalEfield` (string, optional) Specifies the name of the data structure containing the externally computed electric field, $\mathbf{e}^{\mathrm{external}}$.

`externalBfield` (string, optional) Specifies the name of the data structure containing the externally computed magnetic field, $\mathbf{b}^{\mathrm{external}}$.

Parent Updater Data

`in` (string vector, required)

`Vector of Conserved Quantities` (*nodalArray*, **12-components, required**) The vector of conserved quantities, $\mathbf{q}$ has 9 entries:

0. ρ: mass density
1. $\rho\, u_{\hat{\mathbf{i}}} + c^{-2} S^{\mathrm{EM}}_{\hat{\mathbf{i}}} = \left(\rho\mathbf{u} + c^{-2}\mathbf{S}^{\mathrm{EM}}\right)\cdot\hat{\mathbf{i}}$: total momentum density in the $\hat{\mathbf{i}}$ direction
2. $\rho\, u_{\hat{\mathbf{j}}} + c^{-2} S^{\mathrm{EM}}_{\hat{\mathbf{j}}} = \left(\rho\mathbf{u} + c^{-2}\mathbf{S}^{\mathrm{EM}}\right)\cdot\hat{\mathbf{j}}$: total momentum density in the $\hat{\mathbf{j}}$ direction
3. $\rho\, u_{\hat{\mathbf{k}}} + c^{-2} S^{\mathrm{EM}}_{\hat{\mathbf{k}}} = \left(\rho\mathbf{u} + c^{-2}\mathbf{S}^{\mathrm{EM}}\right)\cdot\hat{\mathbf{k}}$: total momentum density in the $\hat{\mathbf{k}}$ direction
4. $E + E^{\mathrm{EM}} = \frac{P}{\gamma-1} + \frac{1}{2}\rho|\mathbf{u}|^2 + E^{\mathrm{EM}}$: total energy density
5. $b_{\hat{\mathbf{i}}} = \mathbf{b}\cdot\hat{\mathbf{i}} = \mu_0^{-1/2}\mathbf{B}\cdot\hat{\mathbf{i}}$: magnetic field normalized by permeability of free-space in the $\hat{\mathbf{i}}$ direction
6. $b_{\hat{\mathbf{j}}} = \mathbf{b}\cdot\hat{\mathbf{j}} = \mu_0^{-1/2}\mathbf{B}\cdot\hat{\mathbf{j}}$: magnetic field normalized by permeability of free-space in the $\hat{\mathbf{j}}$ direction
7. $b_{\hat{\mathbf{k}}} = \mathbf{b}\cdot\hat{\mathbf{k}} = \mu_0^{-1/2}\mathbf{B}\cdot\hat{\mathbf{k}}$: magnetic field normalized by permeability of free-space in the $\hat{\mathbf{k}}$ direction
8. $e_{\hat{\mathbf{i}}} = \mathbf{e}\cdot\hat{\mathbf{i}} = \mu_0^{-1/2}\mathbf{E}\cdot\hat{\mathbf{i}}$: electric field normalized by permeability of free-space in the $\hat{\mathbf{i}}$ direction
9. $e_{\hat{\mathbf{j}}} = \mathbf{e}\cdot\hat{\mathbf{j}} = \mu_0^{-1/2}\mathbf{E}\cdot\hat{\mathbf{j}}$: electric field normalized by permeability of free-space in the $\hat{\mathbf{j}}$ direction
10. $e_{\hat{\mathbf{k}}} = \mathbf{e}\cdot\hat{\mathbf{k}} = \mu_0^{-1/2}\mathbf{E}\cdot\hat{\mathbf{k}}$: electric field normalized by permeability of free-space in the $\hat{\mathbf{k}}$ direction
11. ψ: magnetic field correction potential
12. ϕ: electric field correction potential

`Fastest Wave Speed` (*dynVector*, **1-component, required**) The fastest wave speed across the entire simulation domain, c_{fast}. Can be computed using *hyperbolic (1d, 2d, 3d)* (see below).

`Externally Computed Electric Field` (*nodalArray*, **3-components, optional**) Additional contribution to the electric field, $\mathbf{e}^{\mathrm{external}}$, which is not evolved by Ampere's equation, but does contribution to the induction equation, the Lorentz force and the work done on the plasma. The data structure containing $\mathbf{e}^{\mathrm{external}}$ is specified by the "externalEField" option described below.

0. $e^{\mathrm{external}}_{\hat{\mathbf{i}}} = \mathbf{e}^{\mathrm{external}}\cdot\hat{\mathbf{i}} = \mu_0^{-1/2}\mathbf{E}^{\mathrm{external}}\cdot\hat{\mathbf{i}}$: "externally" computed electric field normalized by permeability of free-space in the $\hat{\mathbf{i}}$ direction.
1. $e^{\mathrm{external}}_{\hat{\mathbf{j}}} = \mathbf{e}^{\mathrm{external}}\cdot\hat{\mathbf{j}} = \mu_0^{-1/2}\mathbf{E}^{\mathrm{external}}\cdot\hat{\mathbf{j}}$:"externally" computed electric field normalized by permeability of free-space in the $\hat{\mathbf{j}}$ direction
2. $e^{\mathrm{external}}_{\hat{\mathbf{k}}} = \mathbf{e}^{\mathrm{external}}\cdot\hat{\mathbf{k}} = \mu_0^{-1/2}\mathbf{E}^{\mathrm{external}}\cdot\hat{\mathbf{k}}$: "externally" computed electric field normalized by permeability of free-space in the $\hat{\mathbf{k}}$ direction

`Externally Computed Magnetic Field` (*nodalArray*, **3-components, optional**) Additional contribution to the magnetic field, $\mathbf{b}^{\mathrm{external}}$, which is not evolved by the induction equation, but does contribute to Ampere's equation, the Lorentz force and the work done on the plasma. The data structure containing $\mathbf{b}^{\mathrm{external}}$ is specified by the "externalBField" option described below.

0. $b^{\text{external}}_{\hat{\mathbf{i}}} = \mathbf{b}^{\text{external}} \cdot \hat{\mathbf{i}} = \mu_0^{-1/2} \mathbf{B}^{\text{external}} \cdot \hat{\mathbf{i}}$: magnetic field normalized by permeability of free-space in the $\hat{\mathbf{i}}$ direction
1. $b^{\text{external}}_{\hat{\mathbf{j}}} = \mathbf{b}^{\text{external}} \cdot \hat{\mathbf{j}} = \mu_0^{-1/2} \mathbf{B}^{\text{external}} \cdot \hat{\mathbf{j}}$: magnetic field normalized by permeability of free-space in the $\hat{\mathbf{j}}$ direction
2. $b^{\text{external}}_{\hat{\mathbf{k}}} = \mathbf{b}^{\text{external}} \cdot \hat{\mathbf{k}} = \mu_0^{-1/2} \mathbf{B}^{\text{external}} \cdot \hat{\mathbf{k}}$: magnetic field normalized by permeability of free-space in the $\hat{\mathbf{k}}$ direction

`out` (string vector, required) For the gasDynamicMhdDednerEqn, one of four output variables are computed, depending on whether the equation is combined with an updater capable of computing fluxes (*classicMusclUpdater (1d, 2d, 3d)*), primitive variables (*computePrimitiveState(1d, 2d, 3d)*), the time step associated with the CFL condition (*timeStepRestrictionUpdater (1d, 2d, 3d)*) or the fastest wave speed in the grid (*hyperbolic (1d, 2d, 3d)*).

`Vector of Fluxes` (*nodalArray*, 12-components) When combined with an updater that computes $\nabla \cdot \mathcal{F}(\mathbf{w})$ (e.g. *classicMusclUpdater (1d, 2d, 3d)*), the equation system returns:

0. $\nabla \cdot \mathcal{F}(\rho)$: mass flux
1. $\nabla \cdot \mathcal{F}\left(\rho\, u_{\hat{\mathbf{i}}} + c^{-2} S^{\text{EM}}_{\hat{\mathbf{i}}}\right)$: $\hat{\mathbf{i}}$ momentum flux
2. $\nabla \cdot \mathcal{F}\left(\rho\, u_{\hat{\mathbf{j}}} + c^{-2} S^{\text{EM}}_{\hat{\mathbf{j}}}\right)$: $\hat{\mathbf{j}}$ momentum flux
3. $\nabla \cdot \mathcal{F}\left(\rho\, u_{\hat{\mathbf{k}}} + c^{-2} S^{\text{EM}}_{\hat{\mathbf{k}}}\right)$: $\hat{\mathbf{k}}$ momentum flux
4. $\nabla \cdot \mathcal{F}(E)$: total energy flux
5. $\nabla \cdot \mathcal{F}\left(b_{\hat{\mathbf{i}}}\right)$: $\hat{\mathbf{i}}$ magnetic field flux
6. $\nabla \cdot \mathcal{F}\left(b_{\hat{\mathbf{j}}}\right)$: $\hat{\mathbf{j}}$ magnetic field flux
7. $\nabla \cdot \mathcal{F}\left(b_{\hat{\mathbf{k}}}\right)$: $\hat{\mathbf{k}}$ magnetic field flux
8. $\nabla \cdot \mathcal{F}\left(e_{\hat{\mathbf{i}}}\right)$: $\hat{\mathbf{i}}$ electric field flux
9. $\nabla \cdot \mathcal{F}\left(e_{\hat{\mathbf{j}}}\right)$: $\hat{\mathbf{j}}$ electric field flux
10. $\nabla \cdot \mathcal{F}\left(e_{\hat{\mathbf{k}}}\right)$: $\hat{\mathbf{k}}$ electric field flux
11. $\nabla \cdot \mathcal{F}(\psi)$: magnetic correction potential flux
12. $\nabla \cdot \mathcal{F}(\phi)$: electric correction potential flux

`Vector of Primitive States` (*nodalArray*, 9-components) When combined with an updater that computes $\mathbf{w} = \mathbf{w}(\mathbf{q})$ (e.g. *computePrimitiveState(1d, 2d, 3d)*), the equation systen returns:

0. ρ: mass density
1. $u_{\hat{\mathbf{i}}} = \mathbf{u} \cdot \hat{\mathbf{i}}$: velocity in the $\hat{\mathbf{i}}$ direction
2. $u_{\hat{\mathbf{j}}} = \mathbf{u} \cdot \hat{\mathbf{j}}$: velocity in the $\hat{\mathbf{j}}$ direction
3. $u_{\hat{\mathbf{k}}} = \mathbf{u} \cdot \hat{\mathbf{k}}$: velocity in the $\hat{\mathbf{k}}$ direction
4. $P = \rho\epsilon(\gamma - 1)$: ideal gas pressure
5. $b_{\hat{\mathbf{i}}} = \mathbf{b} \cdot \hat{\mathbf{i}} = \mu_0^{-1/2} \mathbf{B} \cdot \hat{\mathbf{i}}$: magnetic field normalized by permeability of free-space in the $\hat{\mathbf{i}}$ direction

6. $b_{\hat{\mathbf{j}}} = \mathbf{b} \cdot \hat{\mathbf{j}} = \mu_0^{-1/2} \mathbf{B} \cdot \hat{\mathbf{j}}$: magnetic field normalized by permeability of free-space in the $\hat{\mathbf{j}}$ direction
7. $b_{\hat{\mathbf{k}}} = \mathbf{b} \cdot \hat{\mathbf{k}} = \mu_0^{-1/2} \mathbf{B} \cdot \hat{\mathbf{k}}$: magnetic field normalized by permeability of free-space in the $\hat{\mathbf{k}}$ direction
8. $e_{\hat{\mathbf{i}}} = \mathbf{e} \cdot \hat{\mathbf{i}} = \mu_0^{-1/2} \mathbf{E} \cdot \hat{\mathbf{i}}$: electric field normalized by permeability of free-space in the $\hat{\mathbf{i}}$ direction
9. $e_{\hat{\mathbf{j}}} = \mathbf{e} \cdot \hat{\mathbf{j}} = \mu_0^{-1/2} \mathbf{E} \cdot \hat{\mathbf{j}}$: electric field normalized by permeability of free-space in the $\hat{\mathbf{j}}$ direction
10. $e_{\hat{\mathbf{k}}} = \mathbf{e} \cdot \hat{\mathbf{k}} = \mu_0^{-1/2} \mathbf{E} \cdot \hat{\mathbf{k}}$: electric field normalized by permeability of free-space in the $\hat{\mathbf{k}}$ direction
11. ψ: magnetic field correction potential
12. ϕ: electric field correction potential

`Time Step` (*dynVector*, 1-component) When combined with *timeStepRestrictionUpdater (1d, 2d, 3d)*, the equation system returns the time step consisten with the CFL condition across the entire simulation domain.

`Fastest Wave Speed` (*dynVector*, 1-component) When combined with *hyperbolic (1d, 2d, 3d)*, the equation system returns the fastest wave speed across the entire simulation domain, c_{fast}.

Examples

The following block demonstrates the gasDynamicMaxwellDednerEqn used in combination with *classicMusclUpdater (1d, 2d, 3d)* to compute $\nabla \cdot \mathcal{F}(\mathbf{w})$ with an externally supplied magnetic field:

```
<Updater hyper>
  kind=classicMuscl1d
  onGrid=domain

  # input nodal component arrays
  in=[q   backgroundB]

  # output nodal component array
  out=[qnew]

  # input dynVector containing fastest wave speed
  waveSpeeds=[waveSpeed]

  # the numerical flux to use
  numericalFlux= hlldFlux

  # CFL number to use
  cfl=0.3
  # Form of variables to limit
  variableForm= primitive

  # Limiter; one per input nodal component array
  limiter=[minmod   minmod]

  # list of equations to solve
  equations=[mhd]

  <Equation mhd>
    kind = gasDynamicMaxwellDednerEqn
    gasGamma = GAS_GAMMA
```

```
    lightSpeedFactor  =LIGHT_SPEED_FACTOR
    externalBfield = EXTERNAL_FIELD
    basementPressure = BASEMENT_PRESSURE
    basementDensity = BASEMENT_DENSITY
  </Equation>

  <Source mhdSrc>
    kind = gasDynamicMhdDednerSrc
    scalarConductivity = $1.0/OHMIC_RESISTIVITY$
  </Source>

  <Source mhdClean>
    kind = mhdSrc
    model = mhdDednerEqn
    momentumEnergySource = 1
    inputVariables = [q,divB,gradPsi]
  </Source>

</Updater>
```

The following block demonstrates the gasDynamicMaxwellDednerEqn used in combination with *computePrimitiveState(1d, 2d, 3d)* to compute $\mathbf{w}$:

```
<Updater computePrimitiveState>
  kind = computePrimitiveState$NDIM$d

  onGrid = domain
  # input array
  in = [q]

  # ouput data-structures
  out = [w]

  <Equation fluid>
    kind = gasDynamicMaxwellDednerEqn
    gasGamma = GAS_GAMMA
    lightSpeedFactor  =LIGHT_SPEED_FACTOR
    externalBfield = EXTERNAL_FIELD
    basementPressure = BASEMENT_PRESSURE
    basementDensity = BASEMENT_DENSITY
  </Equation>

</Updater>
```

The following block demonstrates the gasDynamicMhdDednerEqn used in combination with *timeStepRestrictionUpdater (1d, 2d, 3d)* and *hyperbolic (1d, 2d, 3d)* to compute c_{fast} with an externally supplied magnetic field:

```
<Updater getWaveSpeed>
  kind=timeStepRestrictionUpdater1d
  onGrid=domain

  # input nodal component arrays
  in=[q   backgroundB]

  # output dynVector containing fastest wave speed
  waveSpeeds=[waveSpeed]

  # list of equations to compute fastest wave speed for
```

```
  restrictions=[idealMhd]

  # courant condition to apply to the timestep
  courantCondition=1.0

 <TimeStepRestriction idealMhd>
   kind = hyperbolic1d
   model = gasDynamicMaxwellDednerEqn
    gasGamma = GAS_GAMMA
    lightSpeedFactor  =LIGHT_SPEED_FACTOR
    externalBfield = EXTERNAL_FIELD
    basementPressure = BASEMENT_PRESSURE
    basementDensity = BASEMENT_DENSITY
 </TimeStepRestriction>
</Updater>
```

twoFluidEqn

Two fluid equations written as total mass density, momentum density, total charge density total current density and ion and electron energy. The two-fluid equations can also be written as two separate sets of euler equations, however, this form has the advantage that numerical diffusion is applied to the total charge density so that quasi-neutrality is enforced numerically.

$$\frac{\partial}{\partial t}\begin{pmatrix} \rho \\ \rho\, u_x \\ \rho\, u_y \\ \rho\, u_z \\ \rho_c \\ j_x \\ j_y \\ j_z \\ e_i \\ e_e \end{pmatrix} + \nabla \cdot P = 0$$

where P is defined as

$$\begin{pmatrix} \rho_i\, u_{x\,i} + \rho_i\, u_{x\,e} & \rho_i\, u_{y\,i} + \rho_e\, u_{y\,e} & \rho_i\, u_{z\,i} + \rho_e\, u_{z\,e} \\ \rho_i\, u_{x\,i}^2 + P_i + \rho_e\, u_{x\,e}^2 + P_e & \rho_i\, u_{x\,i}\, u_{y\,i} + \rho_e\, u_{x\,e}\, u_{y\,e} & \rho_i\, u_{x\,i}\, u_{z\,i} + \rho_e\, u_{x\,e}\, u_{z\,e} \\ \rho_i\, u_{y\,i}\, u_{x\,i} + \rho_e\, u_{y\,e}\, u_{x\,e} & \rho_i\, u_{y\,i}\, u_{y\,i} + P_i + \rho_e\, u_{y\,e}\, u_{y\,e} + P_e & \rho_i\, u_{y\,i}\, u_{z\,i} + \rho_e\, u_{y\,e}\, u_{z\,e} \\ \rho_i\, u_{z\,i}\, u_{x\,i} + \rho_e\, u_{z\,e}\, u_{x\,e} & \rho_i\, u_{z\,i}\, u_{y\,i} + \rho_e\, u_{z\,e}\, u_{y\,e} & \rho_i\, u_{z\,i}\, u_{z\,i} + P_i + \rho_e\, u_{z\,e}\, u_{z\,e} + P \\ \rho_i\, u_{x\,i} + \rho_i\, u_{x\,e} & \rho_i\, u_{y\,i} + \rho_e\, u_{y\,e} & \rho_i\, u_{z\,i} + \rho_e\, u_{z\,e} \\ r_i(\rho_i\, u_{x\,i}^2 + P_i) + r_e(\rho_e\, u_{x\,e}^2 + P_e) & r_i\rho_i\, u_{x\,i}\, u_{y\,i} + r_e\rho_e\, u_{x\,e}\, u_{y\,e} & r_i\rho_i\, u_{x\,i}\, u_{z\,i} + r_e\rho_e\, u_{x\,e}\, u_{z\,e} \\ r_i\rho_i\, u_{y\,i}\, u_{x\,i} + r_e\rho_e\, u_{y\,e}\, u_{x\,e} & r_i(\rho_i\, u_{y\,i}\, u_{y\,i} + P_i) + r_e(\rho_e\, u_{y\,e}\, u_{y\,e} + P_e) & r_i\rho_i\, u_{y\,i}\, u_{z\,i} + r_e\rho_e\, u_{y\,e}\, u_{z\,e} \\ r_i\rho_i\, u_{z\,i}\, u_{x\,i} + r_e\rho_e\, u_{z\,e}\, u_{x\,e} & r_i\rho_i\, u_{z\,i}\, u_{y\,i} + r_e\rho_e\, u_{z\,e}\, u_{y\,e} & r_i(\rho_i\, u_{z\,i}\, u_{z\,i} + P_i) + r_e(\rho_e\, u_{z\,e}\, u_{z\,e} \\ u_{x\,i}\,(e_i + P_i) & u_{y\,i}\,(e_i + P_i) & u_{z\,i}\,(e_i + P_i) \\ u_{x\,e}\,(e_e + P_e) & u_{y\,e}\,(e_e + P_e) & u_{z\,e}\,(e_e + P_e) \end{pmatrix}$$

With $r_i = q_i/m_i$ and $r_e = q_e/m_e$ where q_e is the electron charge, q_i is the ion charge, m_e is the electron mass and m_i is the ion mass. In addition the variables $(\rho_\alpha, u_{x\,\alpha}, u_{y\,\alpha}, u_{x\,\alpha}, e_\alpha, P_\alpha)$ are the species mass density, species x velocity, species y velocity, species z velocity, species total energy density, and species pressure respectively. In this case α represents the species, either e for electron or i for ion.

Parameters

`ionGamma` (float) Specific heat ratio for the ions

electronGamma (float) Specific heat ratio for the electrons. Defaults to 5/3

ionMass (float) ion mass

electronMass (float) electron mass

ionCharge (float) ion charge

electronCharge electron charge

basementPressure (float) The minimum pressure allowed. Defaults to 0.

basementDensity (float) The minimum density allowed for the ions. Defaults to 0. The electron basement density is determined by multiplying by the mass ratio, therefore $basementDensityElectrons = (me/mi)basementDensity$

Parent Updater Data

in (string vector, required)

1st Input Variable

0. ρ mass density
1. $\rho\, u_x$ x momentum density
2. $\rho\, u_y$ y momentum density
3. $\rho\, u_z$ z momentum density
4. ρ_c total charge density
5. j_x x current density
6. j_y y current density
7. j_z z current density
8. e_i ion energy density
9. e_e electron energy density

Example

An example *twoFluidEqn* equation block is given below:

```
<Equation twoFluid>
  kind = twoFluidEqn
  ionGamma = GAS_GAMMA
  electronGamma = GAS_GAMMA
  ionMass = ION_MASS
  electronMass = ELECTRON_MASS
  ionCharge = ION_CHARGE
  electronCharge = ELECTRON_CHARGE
  basementDensity = BASEMENT_DENSITY
  basementPressure = BASEMENT_PRESSURE
</Equation>
```

userDefinedEqn

Define an arbitrary hyperbolic system. Built in hyperbolic equations should be used when they are available as they are faster.

Parameters

indVars_inName For each input variable an "indVars" array must be defined. So if in = [E, B] then indVars_E and indVars_B must be defined. If indVars_E = ["Ex","Ey","Ez"] then operations are performed on "Ex","Ey" and "Ez" in the expression evaluator.

transform_inName For each variable there must be a vector that tells how the data is transformed upon rotation. For example, for an electric field E, the transform would be transform_E = [vector] so that USim knows the input data is a vector. If the input data is density, momentum, energy as in the euler equations then we would have transform_q = [scalar, vector, scalar] which assumes that momentum has 3 components. The previous example transforms the first variable as if it were a scalar, then the next 3 variables as if they were part of a tensor and then the last variable as if it were a scalar. Available options are *scalar*, *vector* and *tensor*. It is assumed that *vector* has 3 components even in 1D and 2D simulations. Also it's assumed that *tensor* has 6 components in the order Txx, Txy, Tx, Tyy, Tyz, Tzz and that the remaining components are symmetric so are redundant.

preExprs (string vector) Strings must be put in quotes. The preExprs is used to compute quantities based on indVars that can later be used in the *exprs* to evaluate the output. Available commands are defined by the muParser (http://muparser.sourceforge.net)

flux (string vector) Strings must be put in quotes. The strings are used to evaluate the flux in the x-direction. The fluxes in other directions are obtained through rotation of the input vector. Available command are defined by the muParser (http://muparser.sourceforge.net/)

eigenvalues (string vector) Strings must be put in quotes. The strings are evaluated and placed in the output array and are used to define the set of eigenvalues for the system. The eigenvalues are technically the eigenvalues in the x-direction and values in other directions are obtained through rotation. Available command are defined by the muParser (http://muparser.sourceforge.net/)

other (variable definition) In addition, an arbitrary number of constants can be defined that can then be used in evaluating expression in both *preExprs* and *flux* and *eigenvalues*.

Parent Updater Data

in (string vector) Input 1 to N are input nodalArray on which operations will be performed. Example in = [E, B]

out (string vector) output nodalArray where the result of the operation is stored

Example

```
<Updater hyper>
  kind = classicMuscl1d
  onGrid = domain

  in = [q]
  out = [qnew]
  timeIntegrationScheme = none
  numericalFlux = $RIEMANN_SOLVER$
  cfl = CFL
```

```
        variableForm = $VARIABLE_FORM$
        limiter = [$LIMITER$]

        equations = [euler]

        <Equation euler>
          kind = userDefinedEqn

          indVars_q = ["rho","mx","my","mz","en"]
          transform_q = [scalar, vector, scalar]

          gamma = GAS_GAMMA
          preExprs = ["p=(gamma-1.0)*(en-0.5*((mx*mx+my*my+mz*mz)/rho))"]
          flux = ["mx","(mx*mx/rho)+p","(mx*my/rho)","(mx*mz/rho)","(mx/rho)*(en+p)"]
          eigenvalues = ["sqrt(p*gamma/rho)+(mx/rho)"]

        </Equation>

      </Updater>
```

Parameters associated with the *Hyperbolic Equations* can be added to the time step restriction block.

14.3.2 Parent Updater Data

The following data structures should be specified to the *timeStepRestrictionUpdater (1d, 2d, 3d)* that calls the *hyperbolic Time Step Restriction*.

`in` (string vector, required) Input 1 to N are input *nodalArray*s which will be supplied to the equation. Defined by the choice of *Hyperbolic Equations*.

14.3.3 Example

The following block demonstrates *hyperbolic* used in combination with *timeStepRestrictionUpdater (1d, 2d, 3d)* to compute a wave-speed for *mhdDednerEqn*:

```
    <Updater getWaveSpeed>
     kind = timeStepRestrictionUpdater2d
     in = [q]
     waveSpeeds = [waveSpeed]
     onGrid = domain
     restrictions = [idealMhd]
     courantCondition = 1.0

     <TimeStepRestriction idealMhd>
       kind = hyperbolic2d
       model = mhdDednerEqn
       gasGamma = GAMMA
       mu0=MU0
       includeInTimeStep = False
     </TimeStepRestriction>
   </Updater>
```

14.4 plasmaFrequency (1d, 2d, 3d)

Computes the inverse plasma frequency

14.4.1 Parameters

`massDensityIndex` (integer, required) Gives the index of the primitive variable that stores mass density. This is used so that number density can be computed and then plasma frequency calculated.

`speciesCharge` (float, required) Charge of the species for which we are computing the plasma frequency.

`speciesMass` (float, required) Mass of the species for which we are computing the plasma frequency.

`epsilon0` (float, required) Value of permittivity

14.4.2 Parent Updater Data

The following data structures should be specified to the *timeStepRestrictionUpdater (1d, 2d, 3d)* that calls the *plasmaFrequency Time Step Restriction*.

`in` (string vector, required)

`Mass Density` (*nodalArray*, at least 1 component, required) The mass density of the plasma. The component of the data structure that contains the mass density is specified with the parameter *massDensityIndex* (see below).

14.4.3 Example

The following block demonstrates *plasmaFrequency* used in combination with *timeStepRestrictionUpdater (1d, 2d, 3d)* and *cyclotronFrequency (1d, 2d, 3d)* to compute the time-step restriction in a plasma:

```
<Updater twofluidTimeStepRestrictions>
  kind = timeStepRestrictionUpdater1d
  in = [q]
  restrictions = [wpe, wce]
  onGrid = domain
  courantCondition = 1.0

  <TimeStepRestriction wpe>
      kind = plasmaFrequency1d
      speciesCharge = ELECTRON_CHARGE
      speciesMass = ELECTRON_MASS
      epsilon0 = 1.0
      massDensityIndex = 0
  </TimeStepRestriction>

  <TimeStepRestriction wce>
      kind = cyclotronFrequency1d
      speciesCharge = ELECTRON_CHARGE
      speciesMass = ELECTRON_MASS
      magneticFieldIndexes = [23, 24, 25]
      massDensityIndex = 0
  </TimeStepRestriction>

</Updater>
```

14.5 positiveValue (1d, 2d, 3d)

Computes a time step given such that when a source time depenedent source is added to the value, the value in question does not become negative. An example of this is adding radiation loss to the energy of a fluid. If the time step is too large the fluid energy can become negative, however a smaller time step will allow the fluid to radiate energy until radiated energy is negligible and then the time step will rise again. The positiveValue restriction as defined the following way. S is the source term, E is the energy, B_v is the lowest value that we would like E to be, Δt is the time step, c is a *coefficient* which is generally used to alter the sign of the source term and α is a reduction factor which scales the time step. The time step is given below

The time step restriction is derived from simple considerations

$$\frac{\partial E}{\partial t} = S \tag{14.-29}$$

Which suggests that

$$E^{n+1} = E^n + \Delta t\, S > B_v \tag{14.-29}$$

So if we want to make sure we do not subtract all the energy off in one time step then Δt would be chosen as

$$\Delta t < -\frac{E^n - B_v}{S} \tag{14.-29}$$

We introduce the factor α and c to provide more flexibility in determining how the restriction should be applied and the relative sign of the source S. The restriction is then given by the following relation.

$$\Delta t = -\alpha \frac{E - B_v}{c\, S} \tag{14.-29}$$

In the case where $c\, S > 0$ no restriction is applied to Δt.

14.5.1 Parameters

`alpha` (float, required) If alpha is 1.0 then the time step that results will zero out the pressure in one time step. If alpha=0.25 then it will zero out the pressure in 4 time steps. alpha is similar to a cfl number where 1.0 is the maximum value that should be used.

`positiveIndex` (integer, required) Tell which component of the value we intend to keep positive should be used for comparison

`sourceIndex` (integer, required) Tell which component source vector should be used for comparison

`basementValue` (float, required) The value for which the positiveValue should not go lower than

`coefficient` (float, required) The restriction compares the positive value with the source value. If the source value is going to be added to the positive value then *coefficient=1.0*. If the source value is going to be subtracted from the source value then *coefficient=-1.0* is correct.

`numComponents` (integer, optional) Occasionally the user will want to apply the postitive value restriction to an array of variables such as densities in chemical reactions. *numComponents* allows you to specify the number of components that the restriction will be applied to. It's assumed that the values will range from *positiveIndex* to *positiveIndex*'+'*numComponents*-1 and *sourceIndex* to *sourceIndex*'+'*numComponents*-1 and that *coefficient* and *alpha* will be constant for all components.

14.5.2 Parent Updater Data

The following data structures should be specified to the *timeStepRestrictionUpdater (1d, 2d, 3d)* that calls the *positiveValue Time Step Restriction*.

in (string vector, required)

1st variable The first variable is the value we want to keep positive. An example would be energy or pressure.

2nd variable The second variable is the value that will be added to the first value when multiplied by Δt. An example of the second value could be energy loss rate due to radiation or mass loss rate due to chemical reactions.

14.5.3 Example

The following block demonstrates *positiveValue* used in combination with *timeStepRestrictionUpdater (1d, 2d, 3d)* to compute a time step that keeps the energy of the fluid (component 4 of input variable q) positive:

```
<Updater timeStepRestrictionEnerngy>
 kind = timeStepRestrictionUpdater2d
 in = [q, reactionEnergy]
 timeSteps = [diffDT4]
 onGrid = domain
 restrictions = [positiveSource]
 courantCondition = 1.0

 <TimeStepRestriction positiveSource>
   kind = positiveValue2d
   positiveIndex = 4
   sourceIndex = 0
   alpha = 0.01
   coefficient = 1.0
   basementValue = BASEMENT_ENERGY
   includeInTimeStep = false
 </TimeStepRestriction>

</Updater>
```

14.6 quadratic (1d, 2d, 3d)

Computes the minimum time step for systems that have 2nd derivatives such as the heat equation or Navier Stokes. The explicit time step for this type of system goes as the square of grid spacing.

14.6.1 Parameters

constant (float, optional) Multiply the input data by this value in computing the time-step restriction.

14.6.2 Parent Updater Data

The following data structures should be specified to the *timeStepRestrictionUpdater (1d, 2d, 3d)* that calls the *quadratic Time Step Restriction*.

in (string vector, required) `Diffusion coefficient` (*nodalArray*, 1 component, required) κ, where the diffusion operator is $\nabla \cdot \kappa \nabla \phi$.

14.6.3 Example

The following block demonstrates *quadratic* used in combination with *timeStepRestrictionUpdater (1d, 2d, 3d)* to compute the timestep associated with a thermal conductivity:

```
 <Updater timeStepRestriction2>
  kind = timeStepRestrictionUpdater2d
  in = [thermalCond]
  onGrid = domain
  restrictions = [quadratic]
  courantCondition = THERMAL_DIFF_TIMESTEP_FACTOR

  <TimeStepRestriction quadratic>
    kind = quadratic2d
  </TimeStepRestriction>
</Updater>
```

14.7 whistlerWave (1d, 2d, 3d)

Compute the time step restriction determined by the whistler wave and the grid resolution.

14.7.1 Parameters

`massIndex` (integer, optional) Gives the index of the *Vector of Conserved Quanties* that stores mass density. Defaults to 0.

`chargeStateIndex` (integer, optional) Gives the index of the *Vector of Conserved Quanties* that stores mass density. Defaults to 0.

`fundamentalCharge` (float, required) Proton charge

`speciesMass` (float, required) Mass of the species for which we are computing the plasma frequency.

`mu0` (float, required) Permeability of free space.

`bIndex` (integer vector, optional) Gives the indices of the *Vector of Conserved Quanties* that stores the magnetic field. Default: *bIndex=[5 6 7]*.

14.7.2 Parent Updater Data

The following data structures should be specified to the *timeStepRestrictionUpdater (1d, 2d, 3d)* that calls the *whistlerWave Time Step Restriction*.

`in` (string vector, required)

`Vector of Conserved Quanties` (*nodalArray*, at least 8 components, required) The vector of conserved quantities, $\mathbf{q}$ has at least 8 entries:

0. ρ: mass density
1. $\rho\, u_{\hat{\mathbf{i}}} = \rho\mathbf{u}\cdot\hat{\mathbf{i}}$: momentum density in the $\hat{\mathbf{i}}$ direction
2. $\rho\, u_{\hat{\mathbf{j}}} = \rho\mathbf{u}\cdot\hat{\mathbf{j}}$: momentum density in the $\hat{\mathbf{j}}$ direction
3. $\rho\, u_{\hat{\mathbf{k}}} = \rho\mathbf{u}\cdot\hat{\mathbf{k}}$: momentum density in the $\hat{\mathbf{k}}$ direction
4. $E = \frac{P}{\gamma-1} + \frac{1}{2}\rho|\mathbf{u}|^2 + \frac{1}{2}|\mathbf{b}|^2$: total energy density

5. $b_{\hat{\mathbf{i}}} = \mathbf{b} \cdot \hat{\mathbf{i}} = \mu_0^{-1/2} \mathbf{B} \cdot \hat{\mathbf{i}}$: magnetic field normalized by permeability of free-space in the $\hat{\mathbf{i}}$ direction
6. $b_{\hat{\mathbf{j}}} = \mathbf{b} \cdot \hat{\mathbf{j}} = \mu_0^{-1/2} \mathbf{B} \cdot \hat{\mathbf{j}}$: magnetic field normalized by permeability of free-space in the $\hat{\mathbf{j}}$ direction
7. $b_{\hat{\mathbf{k}}} = \mathbf{b} \cdot \hat{\mathbf{k}} = \mu_0^{-1/2} \mathbf{B} \cdot \hat{\mathbf{k}}$: magnetic field normalized by permeability of free-space in the $\hat{\mathbf{k}}$ direction

`Ion Charge State` (***nodalArray*, at least 1 component1, required**) The charge state of the ions in the plasma. If a data structure with more than 1 component is specified, then the component corresponding to the charge state can be supplied with *chargeStateIndex*.

14.7.3 Example

The following block demonstrates *whistlerWave* used in combination with *timeStepRestrictionUpdater (1d, 2d, 3d)* and *hyperbolic (1d, 2d, 3d)* to compute the fastest wave speed and the timestep restriction for Hall MHD:

```
<Updater timeStepRestriction>
  kind = timeStepRestrictionUpdater2d
  in = [q, Zavg]
  onGrid = domain
  waveSpeeds = [waveSpeed]
  restrictions = [idealMhd,whistler]
  courantCondition = CFL

   <TimeStepRestriction idealMhd>
     kind = hyperbolic2d
     model = mhdDednerEqn
     gasGamma = GAS_GAMMA
     mu0=MU0
   </TimeStepRestriction>

  <TimeStepRestriction whistler>
      kind = whistlerWave2d
      fundamentalCharge = CHARGE
      speciesMass = MI
      chargeStateIndex = 0
      mu0 = MU0
      bIndex = [5 6 7]
      massDensityIndex = 0
      gasGamma = GAS_GAMMA
  </TimeStepRestriction>
</Updater>
```

CHAPTER

FIFTEEN

MULTI-SPECIES DATA FILES

Multi-species data such as reaction rate constants, specific heats, atomic data etc can be supplied to USim using an ASCII text file. Each data file can contain the following options:

`REACTIONS` chemical reactions

`CP` specific heat at constant pressure

`EOF` standard energy of formation

`MOLECULARWEIGHT` molecular weight

`MOLECULARDIA` molecular diameter

`DOF` degrees of freedom of a gas molecule

Not all properties need to be included with every file. Data associated with each property is enclosed between lines labelled '<PROPERTY> START' and '<PROPERTY> END', where <PROPERTY> is replaced with one of the above list.

15.1 Multi-Species Chemical Reactions

An example multi-species reaction block that demonstrates a range of reaction types for mult-species chemistry is given below:

```
REACTIONS START
SPECIES N2 N O2 O NO
2 2 F 1.0E-8 0.0 N2 O2 NO NO
2 2 A 300.0 11000.0 6.43E-18  1.0E+0  3.16E+4 1.58E-8 1.0E+0 1.64E+5  N O2 NO O
2 2 E 300.0 12000.0 4.0E-9 0.0 0.0 2.0 5 0.0 0.0 0.0 0.0 0.0 N2 O NO N
REACTIONS END
```

This reactions block demonstrates chemical reactions involving 5 species: *N2 N O2 O NO*, denoted by the second line in the example: *SPECIES N2 N O2 O NO*. Note that each specie is delimited by a space. The next three lines of the example (lines 3 - 5) describe the chemical reactions involving these species, one chemical reaction per line. Data describing the chemical reaction is space delimited. All chemical reactions supported by USim use the following pattern **on one line** to describe the reaction properties

```
Num_LHS Num_RHS Type Parameters LHS_Species RHS_Species
```

where

- *Num_LHS* = Number of species on LHS
- *Num_RHS* = Number of species on RHS
- *Type* = Reaction Type

- *Parameters* = Reaction Parameters
- *LHS_Species* = LHS species list
- *RHS_Species* = RHS species list

In the example above, each reaction has

```
Num_LHS = 2
Num_RHS = 2
```

The third parameter, *Type = F,A,E* denotes the type of chemical reaction and determines what entries are necessary in *Parameters*. The final two entiries on the line list the species on the left-hand side and right-hand sides of the reaction respectively. The above example demonstrates the three types of chemical reactions that it is possible to include in a USim simulation, which are:

`Fixed rate reactions`; Type = *F* Fixed rate reactions are demonstrated on line 3 of the example. In this case, we are demonstrating the system:

Reaction: $N_2 + O_2 \rightleftharpoons 2NO$;

Foward Rate Constant: $k_f = 10^{-8}$;

Backward Rate Constant: $k_b = 0.0$

The data format for fixed rate reactions is as follows (**on one line**):

```
Num_LHS Num_RHS Type Forward_A Backward_A LHS_Species RHS_Species
```

In the example above these are set as:

```
Num_LHS = 2
Num_RHS = 2
Type = F
Forward_A = 1.0E-8
Backward_A = 0.0
LHS_Species = N2 O2
RHS_Species = NO NO
```

`Arrhenius-type Chemical Reactions`; Type = *A*; Arrhenius-type reactions are demonstrated on line 4 of the example. In this case, we are demonstrating the system:

Reaction: $N + O_2 \rightleftharpoons NO + O$;

Forward Rate Constant: $k_f = A\left(\frac{T}{298}\right)^n e^{\left(\frac{-E_a}{RT}\right)}$;

Backward Rate Constant: $k_b = A\left(\frac{T}{298}\right)^n e^{\left(\frac{-E_a}{RT}\right)}$

Arrhenius-type reactions are valid over a temperature range, $T_{min} < T < T_{max}$ which must be specified. The data format for Arrhenius-type reactions is as follows (**on one line**):

```
Num_LHS Num_RHS Type T_min T_max \
  Forward_A Forward_n Forward_Ea \
  Backward_A Backward_n Backward_Ez \
  LHS_Species RHS_Species
```

In the example above, these are set as:

```
Num_LHS = 2
Num_RHS = 2
Type = A
T_min = 300.0
T_max = 11000.0
```

```
    Forward_A = 6.43E-18
    Forward_n = 1.0E+0
    Forward_Ea = 3.16E+4
    Backward_A = 1.58E-8
    Backward_n = 1.0E+0
    Backward_Ea = 1.64E+5
    LHS_Species = N2 O
    RHS_Species = NO N
```

Arrhenius-type Chemical Reactions with Equilibriation; Type = *E* Arrhenius-type reactions with equilibriation are demonstrated on line 5 of the example. In this case, we are demonstrating the system:

$N_2 + O \rightleftharpoons NO + N$

Forward Rate Constant $k_f = A\left(\frac{T}{298}\right)^n e^{\left(\frac{-Ea}{RT}\right)}$

Equilibrium Rate Constant $k_e = Be^{\left(\sum_{i=1}^{m} c_i\left(\frac{10000}{T}\right)^i\right)}$

Backward Rate Constant $k_b = k_f/k_e$

This system of reactions are valid over a temperature range, $T_{min} < T < T_{max}$ which must be specified. USim automatically derives the backward rate constant based on the forward and equilibrium rate constants. The data format for Arrhenius-type reactions with equilibriation is as follows (**on one line**):

```
    Num_LHS Num_RHS Type T_min T_max \
       Forward_A Forward_n Forward_Ea \
       Equilib_B Equilib_m Equilib_c1 Equilib_c2 ... Equilib_cm \
       LHS_Species RHS_Species
```

Note that Equilib_m sets the number of entries Equilib_c1 .... Equilib_cm. In the example above, these are set as:

```
    Num_LHS = 2
    Num_RHS = 2
    Type = E
    T_min = 300.0
    T_max = 12000.0
    Forward_A = 4.0E-9
    Forward_n = 0.0E+0
    Forward_Ea = 0.0E0
    Equilib_B = 2.0
    Equilib_m = 5
    Equilib_c1 = 0.0
    Equilib_c2 = 0.0
    Equilib_c3 = 0.0
    Equilib_c4 = 0.0
    Equilib_c5 = 0.0
    LHS_Species = N O2
    RHS_Species = NO O
```

15.2 Multi-Species Specific Heat At Constant Pressure

An example multi-species specific heat at constant pressure block is given below:

```
CP START
SPECIES N2 N O2 O NO
1 100.0 500.0 5 28.98641 1.853978 -9.647459 16.63537 0.000117
```

```
1 298.0 6000.0 5 21.13 -0.388 0.04 0.02 -0.025
3 100.0 700.0 5 31.32 -20.23 57.86 -36.50 -0.0073 700.0 2000.0 5 30.0 8.77 -3.988 \
                          0.788 -0.7415 2000.0 6000.0 5 20.91 10.72 -2.02 0.14 9.2
1 298.0 6000.0 5 21.13 -0.388 0.04 0.02 -0.025
2 298.0 1200.0 5 23.83 12.58 -1.139 -1.497 0.214 1200.0 6000.0 5 35.99 0.95 -0.148 \
                                                                        0.0099 -3.0
CP END
```

This block demonstrates specific heat at constant pressure for 5 species: *N2 N O2 O NO*, denoted by the second line in the example: *SPECIES N2 N O2 O NO*. Note that each specie is delimited by a space. The specific heat data of each of the species is entered in a separate line in the same order as that of the species list. Data for each species should be entered on a single line, but has been formatted here for ease of viewing. USim species the specific heat at constant pressure using Shomate polynomials defined in multiple temperature ranges according to:

$C_p = a_0 + a_1 t + a_2 t^2 + a_3 t^3 + \frac{a_4}{t^2}$, where $t = T/1000$.

The polynomials are specified through the data format:

```
<Number-of-polynomials> {Polynomial1-parameters} {Polynomial2-parameters} .... \
                                                    {PolynomialN-parameters}
```

where each of the *N* <Number-of-polynomals> are specified through {PolynomialN-parameters}:

```
<Temperature-range-lower-limitN> <Temperature-range-upper-limitN> \
  <Number-coefficients-in-polynomialN> <polynomialN-coefficients>
```

In line 3 of the above example, C_{p,N_2} is specified using a single polynomial in the temperature range [100,500] K using

```
<Number-of-polynomials> = 1
<Temperature-range-lower-limit> = 100.0
<Temperature-range-upper-limit> = 500.0
<Number-coefficients-in-polynomial1> = 5
<polynomial1-coefficient1> = 28.98641
<polynomial1-coefficient2> = 1.853978
<polynomial1-coefficient3> = -9.647459
<polynomial1-coefficient4> = 16.63537
<polynomial1-coefficient5> = 0.000117
```

In line 7 of the above example, $C_{p,NO}$ is specified using two polynomials in the temperature ranges [298,1200] and (1200,6000] K using:

```
<Number-of-polynomials> = 2
<Temperature-range-lower-limit1> = 298.0
<Temperature-range-upper-limit1> = 1200.0
<Number-coefficients-in-polynomial1> = 5
<polynomial1-coefficient1> = 23.83
<polynomial1-coefficient2> = 12.58
<polynomial1-coefficient3> = -1.139
<polynomial1-coefficient4> = -1.497
<polynomial1-coefficient5> = 0.214
<Temperature-range-lower-limit2> = 1200.0
<Temperature-range-upper-limit2> = 6000.0
<Number-coefficients-in-polynomial2> = 5
<polynomial2-coefficient1> = 35.99
<polynomial2-coefficient2> = 0.95
<polynomial2-coefficient3> = -0.148
<polynomial2-coefficient4> = 0.0099
<polynomial1-coefficient5> = -3.0
```

15.3 Multi-Species Energy of Formation, Molecular Weight, Molecular Diameter and Degrees of Freedom

Each of the above mentioned properties are constant and hence share the same block format with the corresponding starting and ending lines. The example below is for molecular weight data. Block header 'MOLECULARWEIGHT START' is followed by list of species 'SPECIES N2 N O2 O NO'. Each of the species is delimited by space. The next line has the molecular weights entered in the same order as that of the species list. The block is closed with 'MOLECULARWEIGHT END'.

```
MOLECULARWEIGHT START
SPECIES N2 N O2 O NO
28.0 14.0 32.0 16.0 30.0
MOLECULARWEIGHT END
```

A

anisotropic diffusion, 58
Arrhenius-type Chemical Reactions, 265
Arrhenius-type Chemical Reactions with Equilibriation, 265

B

Backward, 93
bin, 31
binCells, 82
bodyFitted, 26
Boundary Condition, 207
boundaryEntityGenerator, 79
bremsPowerSrc, 180

C

characteristicCellLength, 79
Chemical Reactions, 265
classicMusclUpdater, 50
coilFieldEqn, 182
collisionFrequency, 188
combiner, 42
computeChargeError, 187
computePrimitiveState, 45
conductivityTensor, 191
copy, 207
Crank-Nicholson, 93
curl, 56
current, 183
curvilinear coordinates, 144, 148
cyclotronFrequency, 215

D

Data Files, 265
DataStruct, 31
DataStructAlias, 33
Degrees of Freedom, 269
diffusion, 58
Dirchlet, 207
divergence, 56
divergence cleaning, 104, 108, 112, 117, 122, 132, 224, 228, 232, 237, 242, 250
dynVector, 32
dynVectorOperator, 43

E

electron energy, 117, 122, 237, 242
Energy of Formation, 269
entityGenerator, 80
Equation, 97
equationUpdater, 45
Euler, 93
eulerBc, 208
eulerEqn, 97, 217
eulerSym, 144
explicit time integration, 71
exprHyperSrc, 153

F

fieldAtPoint, 83
first derivatives, 56
firstOrderMusclUpdater, 48
Fixed-rate chemical reactions, 265
frequency, 216
functionBc, 208

G

gasDynamicMaxwellDednerEqn, 132, 250
gasDynamicMhdDednerEqn, 112, 232
general equation of state, 100, 220
generalBc, 209
generalizedOhmsLaw, 67
gradient, 56
grid, 25

H

hydrodynamics, 97, 217
hyperbolic, 217
hyperbolicCleanSym, 188

I

idealGasComputeVariables, 167
idealGasVariables, 166
implicitMultiUpdater, 72

initialize, 39
intCombinedFields, 83
inviscid compressilbe hydrodynamics, 144
iterative solver, 72

J

Jacobian Free Newton Krylov, 72

K

kEpsilonOperator, 65
kOmegaOperator, 63

L

laminar viscosity, 60
laplacian, 58
linearCombiner, 41
lineIntegral, 84
localOdeIntegrator, 76
lorentzForce, 184

M

magnetohydrodynamics, 104, 108, 112, 117, 122, 132, 148, 224, 228, 232, 237, 242, 250
maxCombinedFields, 85
Maxwell's equations, 127, 129, 247
maxwellBc, 210
maxwellDednerEqn, 129, 247
maxwellEqn, 127
maxwellSym, 150
mhdBc, 210
mhdDednerEosEqn, 108, 228
mhdDednerEqn, 104, 224
mhdSrc, 154
mhdSym, 148
minDistanceToWall, 80
Molecular Diameter, 269
Molecular Weight, 269
momentumEnergyExchange, 192
multispecies, 103, 223
multiSpeciesSingleVelocityEqn, 103, 223
multiSpeciesSym, 151
multiUpdater, 71

N

nanChecker, 89
Navier-Stokes, 60
navierStokesViscousOperator, 60
Neumann, 207
NFluidSrc, 194
nodalArray, 32
ntCart, 25

O

operatorEntityGenerator, 81
ordinary differential equation', 76

P

paintEntity, 82
periodicCartBc, 211
plasmadynamics, 138, 256
plasmaFrequency, 260
positiveValue, 261
pressureDensityCorrector, 90
primitive state, 45
propaceosComputeVariables, 171
propaceosVariables, 169

Q

quadratic, 262

R

radiationAbsorption, 181
radiationEmission, 182
reactionTableRhs, 195
real gas, 99, 100, 219, 220
realGasEosEqn, 100, 220
realGasEqn, 99, 219
resistiveOperator, 69
Reynolds-Averaged Navier Stokes, 60
Runge Kutta, 93

S

second derivatives, 58
sesameComputeVariables, 175
sesameVariables, 173
simpleBc, 211
simpleTwoTemperatureMhdDednerEqn, 117, 237
Source, 143
Specific Heat At Constant Pressure, 267
stress, 60
sufaceEvapration, 212
Super Time Step, 93
surface ablation, 88
Surface Diagnostic, 87
surfaceIntegral, 86

T

temperatureAndHeatFlux, 89
temperatureRelaxation, 197
ten moment, 101, 222
tenMomentBc, 212
tenMomentEqn, 101, 222
tenMomentFluidSrc, 158
thermal conduction, 60
thirdOrderMusclUpdater, 54
time integration, 93
Time Step, 215

time step, 77
timeStepRestrictionUpdater, 77
transportCoeffSrc, 198
turbulent, 60
two fluid, 101, 222
twoFluidEqn, 138, 256
twoFluidSrc, 160
twoFluidSym, 152
twoTemperatureMhdDednerEqn, 122, 242

U

uniformCombiner, 41
unstructMusclUpdater, 52
unstructured, 28
Updater, 39
UpdateSequence, 37
UpdateStep, 35
userDefinedEqn, 139, 258

V

valueCorrector, 91
vanDerWaalsComputeVariables, 179
vanDerWaalsVariables, 177
vector, 56
vertexJetUpdater, 46
viscosity, 60
viscous, 60

W

wave speed, 77
whistlerWave, 263
wireFieldEqn, 186