USimInDepth

Release 3.0.1

Tech-X Corporation

May 03, 2018

CHAPTER

ONE

BASIC CONCEPTS

1.1 Pre File Syntax

The most basic elements of the USim simulation process, which are discussed in USimcomposer-intro and considered prerequisites for this section, are creating, running, and visualizing a run space. Here we will examine the basic concepts within a USim input file, which contains more detailed information than the Key Input Parameters view in the USimComposer *Setup* tab, and is by default not exposed to the user.

This section discusses the syntax used in pre files.

A pre file consists of:

- Comments
- Variables
- Top-level simulation parameters
- Parameters and vectors of parameters organized into input blocks
- Macros

1.1.1 Accessing the Input File

To access the input file in a run space, navigate to the *Setup* tab and click the View Input File button, which is circled in red in the below figure.

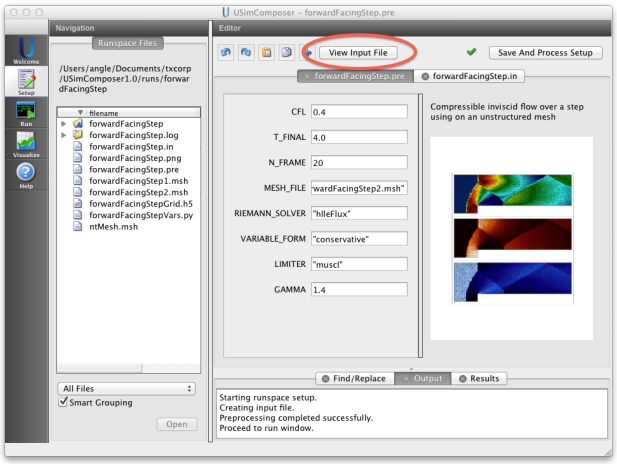
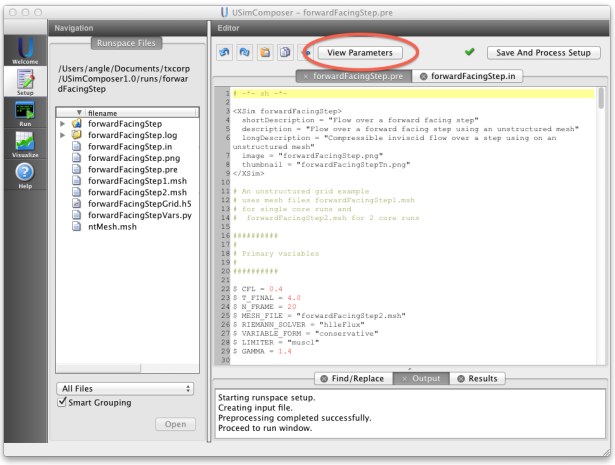


Fig. 1.1: Click the View Input File Button to change from the Parameters View to the Input File Editor.

This opens the Input File view as shown in the below:



Note that you can return to the Parameters view by clicking the Parameters View button, which is circled in red in the above figure.

The remainder of this section describes the basic elements of a USim input file. For a more detailed description of USim input files, see *Basic USim Simulations*.

1.1.2 Symbol Definition

In USim, symbols are defined by assignment, similar to many other programming languages. For example, to define a given symbol with an expression, the syntax is:

```
$SYMBOL = EXP
```

where `SYMBOL` is the name of the symbol and `EXP` is any valid expression.

The expression `EXP` is a valid expression. See *Expression Evaluation* for details.

The preprocessor will not try to substitute a symbol on the left hand side of an equal sign =. For example, the following code snippet:

```
$echarge = 1.6e-19
charge = echarge
```

results in:

```
charge = 1.6e-19
```

Comments

You can enter comments in either of two ways:

- Following a pound sign (#) either on a new line or a continuation of a current line
- Between the opening and closing comment tags <Comment> </Comment>

Note: Tech-X recommends that you always update your comments when you make changes to a pre file. The reasoning behind a change may become unclear if you do not provide comments that explain why you made the change. Pre files with old, out-of-date comments are difficult to work with.

Variables

Each line defining a variable begins with a dollar sign ($).

Parameters

Parameters can be integers, floating-point numbers, or text strings.

The format of the parameter value determines the type of parameter. For example:

- x = 10 indicates an integer
- x = 10.0 indicates a floating-point number
- x = ten indicates a text string

Some parameters accept any text string (within reason). Other parameters accept only a choice of text strings.

If USim can parse a value, such as `42`, as an integer, it will do so. If USim cannot parse the value as an integer, it will attempt to parse it as a floating-point number – for example, any of the following:

```
42.
3.14159
1.60217646e-19
```

If USim cannot parse the value as either an integer or a floating-point number, it will parse the value as a string of text, for example, either of the following:

```
4o. (4 and lowercase O) or
4O (4 and uppercase O).
```

Given these rules, use a decimal point to specify a floating point number. Any number without of decimal point will otherwise be interpreted as an integer.

If a parameter is specified twice, USim will use the second occurrence of the parameter in the input file produced from the pre file. The style recommendations in this user guide will help avoid multiple specifications of parameters.

Vectors of Parameters

Vectors of parameters are enclosed by brackets `[ ]` with white space used as separators. For example:

- `x = [10 10 10]` indicates a vector of integers
- `x = [10.  10.  10.]` indicates a vector of floats

1.1.3 Input Blocks

Input blocks are used to create simulation objects. The block is enclosed by opening and closing tags such as:

```
<Grid globalGrid>
  .
  .
  .
</Grid>
```

The tag determines:

- **object type:** indicated by an initial capital letter, for example, `Grid`
- **object name:** indicated by an initial lowercase letter, for example, `globalGrid`

You use the object name to refer to the object in other input blocks. For example, in the input block for a particle object, you may refer to the name of the electromagnetic field object.

Input blocks can be nested. For example, input blocks for boundary conditions are nested within the input block for an electromagnetic field.

1.1.4 Macros

Macros simplify input file construction through providing a mechanism for encapsulating commonly used input file snippets. A user can then put into the input file only the macro, and then it will be expanded into the full input file at the time of pre-processing the prefile.

Macros can have multiple uses including importing a group of parameters from a separate file, or simplifying an input block such as follows:

```
<macro  myFluid>

 equations = [euler]

 <Equation euler>
   kind = eulerEqn
   gasGamma = GAMMA
 </Equation>
</macro>
```

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

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

 myFluid

</Updater>
```

For more information about macros, see *Overview*

1.1.5 Scoping and Evaluation

Symbols in USim are *scoped*. This means that the effect of a symbol's definition is confined to the macro or block in which that symbol is defined. Whenever USim enters a macro or a new input file block, it enters a new scope.

In the case in which `SYMBOL` is defined in multiple scopes, USim ignores the previously defined `SYMBOL` for the duration of the current scope. In the case in which `SYMBOL` is defined more than once in the current scope, the new value overrides the previous value defined in the current scope.

This scope is closed once USim leaves the block or macro. That is, the symbol's definition no longer has an effect once USim has used the symbol's value in the macro or block where it was defined and then proceeded to a different block or macro. Scoping allows the next block or macro to be free to redefine the value of the symbol for its own purposes.

Global Variables

It is possible to declare a global variable in USim. This is done by first defining the variable, then declaring it global. For example:

```
<Block>
  $ X = 4
  $ global X
</Block>
```

Will cause the variable X to be equal to 4 outside of the Block. It is important to note that the variable must be defined, and declared global on seperate lines. For example $ global X = 4 will not define X as a global variable with value 4.

Expression Evaluation

USim evaluates expressions by interpreting them as Python expressions. Python expressions are composed of tokens. A token is a single element of an expression, such as a constant, identifier, or operation. The preprocessor breaks the expression string into individual tokens then performs recursive substitution on each token. Once a token is no longer found to be substitutable, the preprocessor tries to evaluate it as a Python expression. The result of this evaluation will then be used as the value of this token. All the token values are then concatenated and again evaluated as a Python expression. This result will then be assigned to the symbol.

Tokenizing, the act of breaking a string into tokens, is performed following the lexical rules of Python. This means that white spaces are used to delimit tokens, but are otherwise entirely ignored.

Note: A string within matched quotes is treated as a single token with the matching quotes removed.

The input files generated by USim are sensitive to white spaces; as a result, USim has to re-introduce white spaces in the translation process. By default, tokens are joined without any white spaces. However, if both tokens are of type string, then a white space is introduced. Also, tokens inside an array (delineated by `[` and `]`) are delimited by a white space.

See the Python documentation on the official Python website at http://www.python.org for more information about Python expressions.

Python Token Evaluator (txpp.py)

The Python preprocessor has the following features:

- It accepts a file, conventionally with suffix `.pre`, for processing.
- Lines in that file that start with the character `$` are processed by the preprocessor.
- Those lines are sent through the python interpreter to for evaluation
- The resulting values are replaced and written to a new file with suffix, `.in`

For example, suppose one has an input file, myfile.pre, containing,

```
$ LIGHTSPEED = 2.9979e8
$ LX = 1.e-6
$ NX = 20
$ DX = LX/NX
$ DT = DX/LIGHTSPEED
<Grid thegrid>
    numCells = [NX]
    lengths = [LX]
```

```
</Grid>
dt = DT
```

Pressing the Save and Validate button in USimComposer's *Setup* tab, or equivalently command line execution of:

```
<txpp.py directory>/txpp.py --prefile=myfile.prei
```

produces a file, myfile.in that contains:

```
#$ LIGHTSPEED = 2.9979e8
# -->   LIGHTSPEED      =       299790000.0
#$ LX = 1.e-6
# -->   LX      =       9.9999999999999995e-07
#$ NX = 20
# -->   NX      =       20
#$ DX = LX/NX
# -->   DX      =       4.9999999999999998e-08
#$ DT = DX/LIGHTSPEED
# -->   DT      =       1.6678341505720671e-16

<Grid thegrid>
  numCells=[20]
  lengths=[9.9999999999999995e-07]
</Grid>

dt=1.6678341505720671e-16
```

This mechanism facilitates modifying files to change systems size, resolution, or other parameters while keeping requisite mathematical relationships intact.

The preprocessor imports math, so one can include statements such as:

```
$ PI = math.pi
```

and then use the variable PI in the pre file. In addition, the replace occurs for commented lines as well, so the `myfile.pre` could have contained the line,

```
# dx = DX
```

and then `myfile.in` would have contained the line:

```
# dx = 4.9999999999999998e-08
```

This is useful for printing out intermediate values for, e.g., debugging.

The pre file can be made self executing by adding the stanza:

```
#!/bin/sh
$NTUTILSDIR/txpp.py --prefile=$0 $*
exit $?
```

to the top, where `NTUTILSDIR` is an environment variable that gives the directory of the preprocessor. The preprocessor then knows to skip lines up to exit before processing the file. In addition, the value of any variable named `NDIM` defined in the pre file can be modified on the command line with the directive `-ndim = 2`, for example, to have all occurrences of NDIM in the file replace by `2` instead of the value defined in the file. This enables writing only a single pre file for simulations of multiple dimensionalities when the differences in the file follow from the value of NDIM alone.

If a file fails to validate a brief explanation of what is wrong will be displayed in the *Output* tab under the *Editor* window. Common reasons for a file to fail to invalidate include

1. Using features not available to your USim module. i.e. an example under the USimHS templates will not validate if you are using a USimHEDP license.
2. A variable being declared as an integer instead of a float or vice versa. i.e. $ VAR = 6 instead of $ VAR = 6.0
3. A macro being called without it's parent first being imported.
4. A macro has been called with the wrong number of parameters.

Now that we have examined USim pre file syntax, we are ready to discuss the creation of key parameters in the *Setup* tab of USimComposer in *Key Parameters*.

1.2 Key Parameters

USim has the ability to create key parameters. These variables are visible in the *Editor* pane of the *Setup* tab in USimComposer, and they can be modified without the user having to sift through the input file (also called the pre file). They are useful when creating a base simulation that can be easily modified to simulate different phenomena within the same base simulation. This tutorial is for power users who wish to use key parameters within their own simulations and who are familiar with the USimcomposer-intro. As preparation for a discussion of key parameters, the user must be comfortable with accessing the input file, as discussed in *Pre File Syntax*.

The two main components of the key parameters feature are the XSim block and the XVar block. An example XSim block in a run space input file is boxed in red in the below figure, and an example XVar block is boxed in blue.

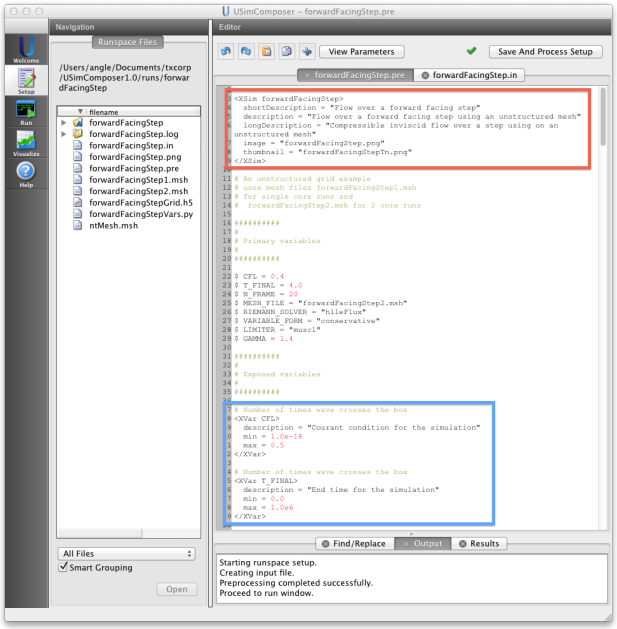


Fig. 1.2: Example XSim and XVar blocks in a run space input file

A description of the elements of these blocks and their effect on the Parameters view in the USimComposer *Setup* tab is given in the following sections.

1.2.1 XSim Block

Given below is a template XSim block that can be modified to fit any file:

```
<XSim simulationName>
  shortDescription = "Simulation Name"
  description = "Description of the simulation."
  longDescription = "Longer description of the simulation."
  image = "simulationName.png"
  thumbnail = "simulationNameTn.png"
</XSim>
```

Each line in this block is explained below:

1. **image** - The image parameter should give the name of a picture, located in the same directory as the .pre file, that will be given on the right hand side of the *Editor* pane in the *Setup* tab. Frequently, this image is used to illustrate key parameters such as dimensions of a physical structure. 400 by 500 pixels is a good image size.
2. **longDescription** - This text block will be visible above the image, and is generally used to give a description of what the simulation does, and what will happen when key parameters are modified.

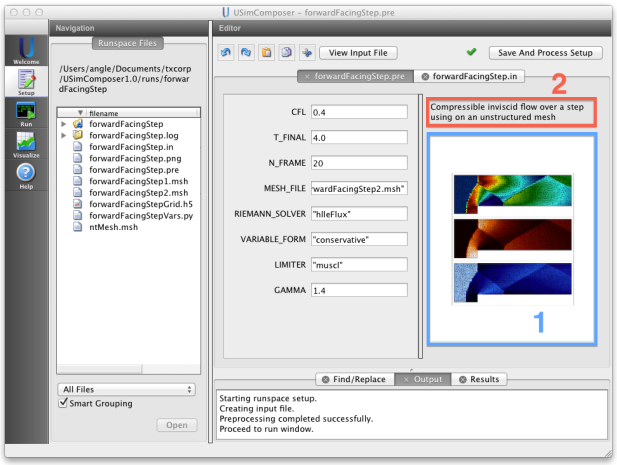


Fig. 1.3: Where image and longDescription appear in the Parameters View.

The three following parameters are only useful to very advanced users who are creating, and placing input files in the Examples directory of USimComposer. The examples directory can be found in [USimInstallDirectory]ContentsExamples.

3. **thumbnail** - This is the small image that is visible when you select an example file, located in the same directory as the .pre file. 250 x 250 pixels in a good image size.
4. **shortDescription** - This is the name that will be given to the example file.
5. **description** - This is the description given in the window on the right side in the examples window.
6. **analyzers[SCRIPT]** - This will cause USimComposer to load the analysis script specified by *SCRIPT*, located in the same directory as the .pre file, for use in the Analyze Tab.

1.2.2 XVar Block

Key parameters can be created in the input, or .pre, file of a USim simulation, and appear in the *Setup* tab in USimComposer as seen below boxed in red.

To create these parameters, the user must modify the .pre file and add XVar blocks, in the same way that the user must add an XSim block as described in the preceding section. It is the practice of Tech-X developers to first declare the

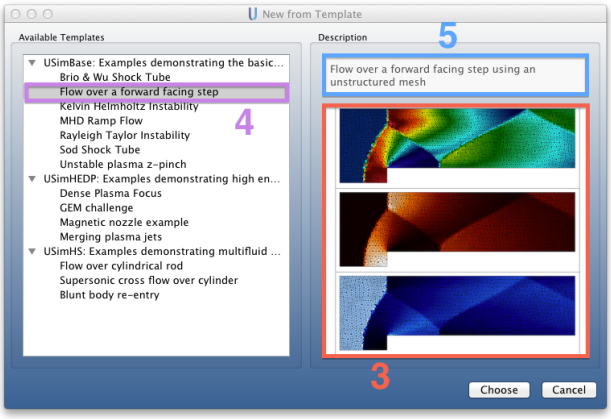


Fig. 1.4: Select An Example Window

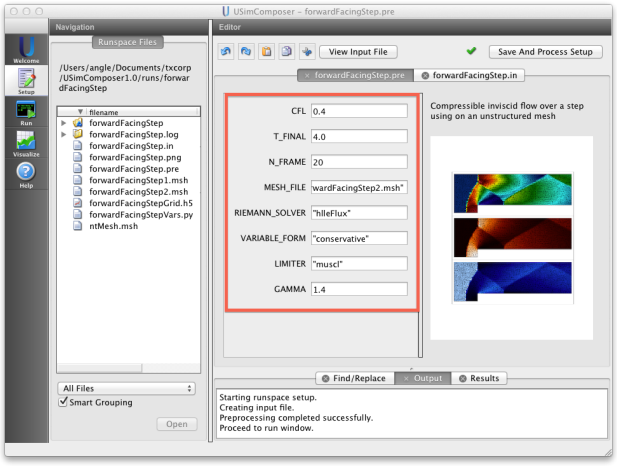


Fig. 1.5: Key parameters

primary variables with a default value, then give the XVar blocks for the primary variables below that. Given below is a template XVar block that can be modified to fit any file:

```
$ variableName = default value
<XVar variableName>
  description = "Description of the variable"
  min = minimum value
  max = maximum value
</XVar>
```

Each line in this block is explained below:

1. **variableName** - The very first line, above the XVar block, sets the default value of the variable.
2. **<XVar variableName>** - This line, which begins the XVar block, must exactly match the name of the variable given in the line preceding it.
2. **description** - This text should describe the variable and will appear when the cursor is placed over the variable name.
3. **min** - This is the minimum value for the variable and is optional. This can be very useful with certain simulation parameters such as cell size that can cause an instability if incorrectly specified.
4. **max** - This is the maximum value for the variable and is optional.

Note that the name of the key parameter will turn red if there is no value given for the parameter, or if the value is not greater than or equal to **min** and less than or equal to **max**, if they are specified.

CHAPTER

TWO

MACROS

2.1 Introduction

USim contains a number of pre-defined macros that are used throughout the example input files available through the USimComposer interface. The macros are used to help automate the process of setting up certain types of simulations. Input files can also be generated by external tools, one that we've found especially useful is Mako

2.2 Overview

2.2.1 Using Macros in Input Files

A macro is a mechanism to abstract complex input file sequences into (parameterized) tokens. In its simplest form, a macro provides a way to substitute a code snippet from an input file:

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

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

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

```
<macro  myFluid>

  equations = [euler]

  <Equation euler>
    kind = eulerEqn
    gasGamma = GAMMA
  </Equation>
</macro>
```

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

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

  myFluid
```

```
</Updater>
```

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

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

  equations = [euler]

  <Equation euler>
    kind = eulerEqn
    gasGamma = GAMMA
  </Equation>

</Updater>
```

Importing Local Macros

It is also possible to define a macro file, and provided that it is in the same directory as your input file, import it. This is useful when writing one custom macro that will be used over multiple simulations. The macro must have a .mac extension on it to be imported as a local macro. To extend the example above, say the macro myLaser is in the file Lasers.mac, the input file would look like this:

```
$ import fluidModels.mac

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

   myFluid

 </Updater>
```

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

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

  equations = [euler]

  <Equation euler>
    kind = eulerEqn
    gasGamma = GAMMA
  </Equation>

</Updater>
```

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

2.2.2 Macro Parameters

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

```
<macro finiteVolumeData(name, grid, components, write)>
  <DataStruct name>
    kind = nodalArray
    onGrid = grid
    numComponents = components
    numNodes = 1
    writeOut = write
  </DataStruct>
</macro>
```

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

```
finiteVolumeData(density, domain, 1, true)
```

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

```
<DataStruct name>
  kind = nodalArray
  onGrid = grid
  numComponents = components
  numNodes = 1
  writeOut = write
</DataStruct>
```

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

2.2.3 Macro Overloading

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

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

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

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

2.2.4 Defining Functions Using Macros

Macros can be particularly useful for defining complex mathematical expressions, such as defining functions in *expr* lists.

Consider a macro that should simplify the setup of a Gaussian. One could define the following macro:

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

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

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

will result in:

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

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

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

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

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

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

2.2.5 Importing Files

USim allows input files to be split into individual files, thus enabling macros to be encapsulated into separate libraries. For example, physical constant definitions or commonly-used geometry setups can be stored in files that can then be used by many USim simulations. Input files can be nested to arbitrary depth.

Files are imported via the import keyword:

```
$ import FILENAME
```

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

2.2.6 Conditionals

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

A conditional takes either the form:

```
$ if COND
      ...
$ endif
```

or

```
$ if COND
      ...
$ else
      ...
$ endif
```

Conditionals can be arbitrarily nested. All the tokens following the `if` token are interpreted following the expression evaluation procedure (see above) and if they evaluate to true, the text following the if statement is inserted into the output. If the conditional statement evaluates to false, the text after the `else` is inserted (if present). Note that `true` and `false` in preprocessor macros are evaluated by Python – in addition to evaluating conditional statements such as `x == 1`, other tokens can be evaluated. The most common use of this is using `0` for false and `1` for true. Empty strings are also evaluated to false. For more detailed information, consult the Python documentation.

Example Conditional Statements

```
$ if TYPE == "MHD"
$    numComponents = 9
$ else
$    numComponents = 5
$ endif
```

A conditional statement can also use Boolean operators:

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

2.2.7 Repetition

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

```
$ while COND
    .
    .
    .
$ endwhile
```

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

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

2.2.8 Recursion

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

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

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

2.2.9 Symbol Definition on the Command Line

txpp allows symbols to be defined on the command line. These definitions override any symbol definitions in the outer-most (global) scope. This allows you to set a default value inside an input file that can then be overridden on the command line if needed.

For example, if the following is in the outermost scope of the input file (outside of any blocks or macros):

```
$ X = 3
X
```

Then this will result in a line containing `3` in the output. However, if you were to invoke txpp via:

```
txpp.py -DX=4
```

then this will result in a line `4`.

However, if you were to define `X` inside a block (not in the global scope), such as:

```
<block foo>
    $ X = 3
    X
</block>
```

then `X` will always be `3`, no matter what value for `X` is specified on the command line.

2.2.10 Requires

When writing reusable macros, best practices compel macro authors to help ensure that the user can be prevented from making obvious mistakes. One such mechanism is the requires directive, which terminates translation if one or more symbols are not defined at the time. This allows users to write macros that depend on symbols that are not passed as

parameters. For example, the following code snippet will not be processed if the symbol `NDIM` has not been previously defined:

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

2.2.11 String Concatenation

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

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

However, we can get around this rule to get the desired output with the following:

```
<macro  concat(a, b)>
    $ tmp = 'a tmp b'
</macro>
```

Now when calling

```
concat(hello, world)
```

the result will be:

```
helloworld
```

The first line appends a single quote to `a` and stores the result in `tmp`. The next line then puts the token `a` together with the token `b`. As they are now no longer two strings, they will be concatenated without a space. The final evaluation of the resulting string then removes the quotes around it, resulting in the desired output.

Now that we have examined macros in an overview, we are ready for *Introduction*.

CHAPTER

THREE

BASIC USIM SIMULATIONS

The following tutorials can be worked through with a *USimBase* license and utilize *Macros* to perform simulations using USim.

3.1 Using USim to solve the Euler Equations

In this tutorial, we demonstrate the basic methods used by USim to solve the Euler equations. This will serve as a foundation for understanding how to run any simulation in USim.

Contents

3.1.1 The Euler Equations

This tutorial is based on the quickstart-shocktube example. The *Shock Tube* simulation is designed to set up a variety of tube simulations including those by Einfeldt, Sod, Liska & Wendroff, Brio & Wu, and Ryu & Jones. In this tutorial, we will look at the Sod Shock Tube based on the classic paper:

```
Sod, Gary A. "A survey of several finite difference methods for
systems of nonlinear hyperbolic conservation laws.";
Journal of Computational Physics 27.1 (1978): 1-31.
```

Note: It is important to note that while we will be following the quickstart-shocktube example, we will not reproduce the example file in its entirety. The shockTube example is designed to set up a variety of simulations, and by using "Flags" (*if* statements) in the shockTube.pre file, only certain parts of the file are used when simulating the Sod Shock Tube. Therefore, at the end of this tutorial, our input will not be an exact copy of the shockTube.pre file but will directly represent only the Sod Shock portion of it.

In this example we will look at the use of equations for inviscid compressible hydrodynamics (the *Euler* equations). It is appropriate to use these equations for transonic, supersonic and hypersonic flows (Mach numbers of 0.1 and

above) where compressibility effects are important and at high Reynolds numbers where the effects of viscosity and conductivity are relatively unimportant.

The Euler equations for an adiabatic gas can be written as a hyperbolic conservation law, which has the form:

$$\frac{\partial \mathbf{q}}{\partial t} + \nabla \cdot [\mathcal{F}(\mathbf{w})] = 0$$

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), and $\mathbf{w} = \mathbf{w}(\mathbf{q})$. Assuming an adiabatic equations of state, the Euler equations can be written in this form such that:

$$\begin{aligned} \frac{\partial \rho}{\partial t} + \nabla \cdot [\rho \, \mathbf{u}] &= 0 \\ \frac{\partial \rho \, \mathbf{u}}{\partial t} + \nabla \cdot \left[\rho \, \mathbf{u} \, \mathbf{u}^T + \mathbb{I} P\right] &= 0 \\ \frac{\partial E}{\partial t} + \nabla \cdot [(E + P) \, \mathbf{u}] &= 0 \end{aligned}$$

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 Euler equations are represented in USim by refmanual-eulerEqn. USim solves these equations by calculating an upwind approximation of the flux tensor, $\mathcal{F}(\mathbf{w})$ (see refmanual-classicMUSCL) and then using this approximation to advance the conserved state from time t to $t + \Delta t$ (see refmanual-multiUpdater).

In this tutorial, we introduce the structure of a USim simulation, using quickstart-shocktube as an example. At the end of this tutorial, you will understand how to setup simple USim simulations on structured meshes.

3.1.2 Initializing a Simulation

The first step in any USim simulation is to import the macros that are needed to define the simulation. For the *ShockTube* example, there are two macros that are needed:

```
$ import fluidsBase.mac
$ import euler.mac
```

These macros provide basic capabilities for setting up a USim simulation of the Euler Equations. We can now define global parameters that tell USim basic information about the simulation. This is done through the use of the *initializeFluidSimulation* macro:

```
initializeFluidSimulation(NDIM,0.0,END_TIME,NUMDUMPS,CFL,GAS_GAMMA,WRITE_RESTART,DEBUG)
```

The parameters for the version of the macro used for the refmanual-eulerEqn are documented at euler-macro. For completeness, we include them here, showing how the parameters specified here map onto the parameters used in the *initializeFluidSimulation* macro:

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

`0.0` (tStart): Start time for simulation

`END_TIME` (tEnd): End time for simulation

`NUMDUMPS` (numFrames): Number of data outputs

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

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

`WRITE_RESTART` (writeRestartIn): Output data required for simulation restart

DEBUG (debugIn): Run simulation in debug mode

USim includes the ability to perform in place substitution of variables within the input file, as described in *Symbol Definition*. This means that we can define the options given above earlier in the input file and USim will replace them in the *initializeFluidSimulation* macro:

```
# X-extent of domain
$PAR_LENGTH = 1.0
# Y-extent of domain
$PERP_LENGTH = 1.0
# Zones parallel to shock normal direction
$PAR_ZONES = 256
# Zones perpendicular to shock normal direction
$PERP_ZONES = 256
# Shock tube to solve
$SHOCK_TUBE = "SOD"
# adiabatic index
$GAS_GAMMA = 5.0/3.0
# Atmospheric Pressure
$REFERENCE_PRESSURE = 1.0 # [Pa]
# density of air
$REFERENCE_DENSITY = 1.0 # [kg/m^3]
# end time for simulation (units of # of sound wave crossings)
$TEND = 0.125
# number of frames
$NUMDUMPS = 10
# Whether to use a diffusive (but robust) integration scheme
$DIFFUSIVE = False
# Order in time
$TIME_ORDER = "second"
# Write data for restarting the simulation
$WRITE_RESTART = False
# Output info for debugging purposes
$DEBUG = False
# Default dimensionality
$NDIM = 1
```

Note that we have included a range of other variables that are defined in the quickstart-shocktube example. We will refer to these later in this tutorial as needed. In the above, the variable *TEND* is given in units of the the number of times a sound wave crosses the grid. In order to use this to specify the end time for the simulation (*END_TIME*), we convert this to have units of time through:

```
# nominal speed of sound
$ c0 = math.sqrt(GAS_GAMMA*REFERENCE_PRESSURE/REFERENCE_DENSITY) # [m/s]

# Set end time according to (user specified)
# number of times a wave crosses the box.
$END_TIME = $PERP_LENGTH*TEND/c0$
```

The specification of the *CFL* parameter is described in the next section. Our simulation starts off as:

```
# Import macros to setup simulation
$ import fluidsBase.mac
$ import euler.mac

# X-extent of domain
$PAR_LENGTH = 1.0
# Y-extent of domain
$PERP_LENGTH = 1.0
```

```
# Zones parallel to shock normal direction
$PAR_ZONES = 256
# Zones perpendicular to shock normal direction
$PERP_ZONES = 256
# Shock tube to solve
$SHOCK_TUBE = "SOD"
# adiabatic index
$GAS_GAMMA = 5.0/3.0
# Atmospheric Pressure
$REFERENCE_PRESSURE = 1.0 # [Pa]
# density of air
$REFERENCE_DENSITY = 1.0 # [kg/m^3]
# end time for simulation (units of # of sound wave crossings)
$TEND = 0.125
# number of frames
$NUMDUMPS = 10
# Whether to use a diffusive (but robust) integration scheme
$DIFFUSIVE = False
# Order in time
$TIME_ORDER = "second"
# Write data for restarting the simulation
$WRITE_RESTART = False
# Output info for debugging purposes
$DEBUG = False
# Default dimensionality
$NDIM = 1

# nominal speed of sound
$ c0 = math.sqrt(GAS_GAMMA*REFERENCE_PRESSURE/REFERENCE_DENSITY) # [m/s]

# Set end time according to (user specified)
# number of times a wave crosses the box.
$END_TIME = $PERP_LENGTH*TEND/c0$

# Initialize a USim simulation
initializeFluidSimulation(NDIM,0.0,END_TIME,NUMDUMPS,CFL,GAS_GAMMA,WRITE_RESTART,DEBUG)
```

3.1.3 Adding a Simulation Grid

The next step in setting up a USim simulation is to specify a simulation grid. The quickstart-shocktube uses a refmanual-ntcart grid, which is added to the simulation through the *addGrid* macro:

```
addGrid(lowerBounds, upperBounds, numCells, periodicDirections)
```

The options for this macro are documented in grid-macro. For completeness, we include them here:

`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)
```

The options are setup in the *ShockTube* example according to the dimensionality of the simulation, as specified by the variable *NDIM* above:

```
$XMIN = -0.5*PERP_LENGTH
$XMAX =  0.5*PERP_LENGTH
$YMIN = -0.5*PAR_LENGTH
$YMAX =  0.5*PAR_LENGTH
$ZMIN = -0.5*PAR_LENGTH
$ZMAX =  0.5*PAR_LENGTH

$if NDIM==1
$ CFL = 0.5
$ numCells = [PERP_ZONES]
$ periodicDirections = []
$ lowerBounds = [XMIN]
$ upperBounds = [XMAX]
$ else
$if NDIM==2
$ CFL = 0.4
$ numCells = [PERP_ZONES, PAR_ZONES]
# Make the direction perpendicular to the shock
# normal periodic.
$ periodicDirections = [1]
$ lowerBounds = [XMIN,  YMIN]
$ upperBounds = [XMAX,  YMAX]
$ else
$ CFL = 0.32
$ numCells = [PERP_ZONES, PAR_ZONES, PAR_ZONES]
# Make the directions perpendicular to the shock
# normal periodic.
$ periodicDirections = [1,2]
$ lowerBounds = [XMIN,  YMIN,  ZMIN]
$ upperBounds = [XMAX,  YMAX,  ZMAX]
$ endif
$ endif
```

Note as well that the CFL condition that is used for the simulation changes according to the dimension of the simulation. This is because the scheme for solving the hyperbolic equations has different stability requirements in different dimensions:

$$\text{CFL} <= \frac{1}{\text{NDIM}}$$

So, our simulation now looks like:

```
# Import macros to setup simulation
$ import fluidsBase.mac
$ import euler.mac

# X-extent of domain
$PAR_LENGTH = 1.0
# Y-extent of domain
$PERP_LENGTH = 1.0
# Zones parallel to shock normal direction
$PAR_ZONES = 256
# Zones perpendicular to shock normal direction
$PERP_ZONES = 256
# Shock tube to solve
$SHOCK_TUBE = "SOD"
# adiabatic index
```

```
$GAS_GAMMA = 5.0/3.0
# Atmospheric Pressure
$REFERENCE_PRESSURE = 1.0 # [Pa]
# density of air
$REFERENCE_DENSITY = 1.0 # [kg/m^3]
# end time for simulation (units of # of sound wave crossings)
$TEND = 0.125
# number of frames
$NUMDUMPS = 10
# Whether to use a diffusive (but robust) integration scheme
$DIFFUSIVE = False
# Order in time
$TIME_ORDER = "second"
# Write data for restarting the simulation
$WRITE_RESTART = False
# Output info for debugging purposes
$DEBUG = False
# Default dimensionality
$NDIM = 1

# nominal speed of sound
$ c0 = math.sqrt(GAS_GAMMA*REFERENCE_PRESSURE/REFERENCE_DENSITY) # [m/s]

# Set end time according to (user specified)
# number of times a wave crosses the box.
$END_TIME = $PERP_LENGTH*TEND/c0$

$XMIN = -0.5*PERP_LENGTH
$XMAX =  0.5*PERP_LENGTH
$YMIN = -0.5*PAR_LENGTH
$YMAX =  0.5*PAR_LENGTH
$ZMIN = -0.5*PAR_LENGTH
$ZMAX =  0.5*PAR_LENGTH

$if NDIM==1
$ CFL = 0.5
$ numCells = [PERP_ZONES]
$ periodicDirections = []
$ lowerBounds = [XMIN]
$ upperBounds = [XMAX]
$ else
$if NDIM==2
$ CFL = 0.4
$ numCells = [PERP_ZONES, PAR_ZONES]
# Make the direction perpendicular to the shock
# normal periodic.
$ periodicDirections = [1]
$ lowerBounds = [XMIN,  YMIN]
$ upperBounds = [XMAX,  YMAX]
$ else
$ CFL = 0.32
$ numCells = [PERP_ZONES, PAR_ZONES, PAR_ZONES]
# Make the directions perpendicular to the shock
# normal periodic.
$ periodicDirections = [1,2]
$ lowerBounds = [XMIN,  YMIN,  ZMIN]
$ upperBounds = [XMAX,  YMAX,  ZMAX]
$ endif
```

```
$ endif

# Initialize a USim simulation
initializeFluidSimulation(NDIM,0.0,END_TIME,NUMDUMPS,CFL,GAS_GAMMA,WRITE_RESTART,DEBUG)

# Setup the grid
addGrid(lowerBounds, upperBounds, numCells, periodicDirections)
```

3.1.4 Creating a Fluid Simulation

The next step in setting up a USim simulation is to create the basic set of variables needed to simulate the fluid. This is accomplished through the *createFluidSimulation* macro, documented at euler-macro:

```
createFluidSimulation()
```

Now that these variables have been automatically created via the macro, it is possible to specify the distribution of fluid on the grid. This is a three-step process:

1. Use *addVariable* to add variables that are independent of grid position.
2. Use *addPreExpression* to add quantities that are functions of grid position, variables and any previously defined PreExpression in this block. Evaluated before expressions and the result is not accessible outside of this block. Any number of PreExpressions can be added.
3. Use *addExpression* to define each initial condition for the fluid. There is one expression for density, each component of momentum and the total energy. The order of the exprssions correspond to the order in the state vector and there can only be one expression per entry in the state vector.

For the *ShockTube* example, the variables added in Step 1 in the above process are:

```
addVariable(pi_value,math.pi)
addVariable(gas_gamma,GAS_GAMMA)
addVariable(densityL,DENSITY_L)
addVariable(densityR,DENSITY_R)
addVariable(pressureL,PRESSURE_L)
addVariable(pressureR,PRESSURE_R)
addVariable(normalVelocityL,NORMAL_VELOCITY_L)
addVariable(normalVelocityR,NORMAL_VELOCITY_R)
addVariable(perpendicularVelocityL,PERPENDICULAR_VELOCITY_L)
addVariable(perpendicularVelocityR,PERPENDICULAR_VELOCITY_R)
addVariable(tangentialVelocityL,TANGENTIAL_VELOCITY_L)
addVariable(tangentialVelocityR,TANGENTIAL_VELOCITY_R)
```

The variables are then used in the *PreExpression*s that are defined in Step 2:

```
addPreExpression(rho = if (x>0.0, densityR,               densityL))
addPreExpression(pr  = if (x>0.0, pressureR,              pressureL))
addPreExpression(vx  = if (x>0.0, normalVelocityR,        normalVelocityL))
addPreExpression(vy  = if (x>0.0, perpendicularVelocityR, perpendicularVelocityL))
addPreExpression(vz  = if (x>0.0, tangentialVelocityR,    tangentialVelocityL))
```

Finally, these *PreExpression*s are used to specify the initial conditions define in Step 3:

```
addExpression(rho)
addExpression(rho*vx)
addExpression(rho*vy)
addExpression(rho*vz)
addExpression((pr/(gas_gamma-1))+0.5*rho*(vx*vx+vy*vy+vz*vz))
```

The *ShockTube* example includes many different initial conditions that are based on a range of cases proposed in the literature. The simples case is the *SOD* shock tube, specified through *$SHOCK_TUBE = "SOD"* above. This corresponds to the following variable definitions:

```
$ DENSITY_L = $3.0*REFERENCE_DENSITY$
$ DENSITY_R = $0.125*REFERENCE_DENSITY$
$ PRESSURE_L = $1.0*REFERENCE_PRESSURE$
$ PRESSURE_R =  $0.1*REFERENCE_PRESSURE$
$ NORMAL_VELOCITY_L = 0.0
$ NORMAL_VELOCITY_R = 0.0
$ PERPENDICULAR_VELOCITY_L = 0.0
$ PERPENDICULAR_VELOCITY_R = 0.0
$ TANGENTIAL_VELOCITY_L = 0.0
$ TANGENTIAL_VELOCITY_R = 0.0
```

Our simulation now looks like:

```
# Import macros to setup simulation
$ import fluidsBase.mac
$ import euler.mac

# X-extent of domain
$PAR_LENGTH = 1.0
# Y-extent of domain
$PERP_LENGTH = 1.0
# Zones parallel to shock normal direction
$PAR_ZONES = 256
# Zones perpendicular to shock normal direction
$PERP_ZONES = 256
# Shock tube to solve
$SHOCK_TUBE = "SOD"
# adiabatic index
$GAS_GAMMA = 5.0/3.0
# Atmospheric Pressure
$REFERENCE_PRESSURE = 1.0 # [Pa]
# density of air
$REFERENCE_DENSITY = 1.0 # [kg/m^3]
# end time for simulation (units of # of sound wave crossings)
$TEND = 0.125
# number of frames
$NUMDUMPS = 10
# Whether to use a diffusive (but robust) integration scheme
$DIFFUSIVE = False
# Order in time
$TIME_ORDER = "second"
# Write data for restarting the simulation
$WRITE_RESTART = False
# Output info for debugging purposes
$DEBUG = False
# Default dimensionality
$NDIM = 1

# nominal speed of sound
$ c0 = math.sqrt(GAS_GAMMA*REFERENCE_PRESSURE/REFERENCE_DENSITY) # [m/s]

# Set end time according to (user specified)
# number of times a wave crosses the box.
$END_TIME = $PERP_LENGTH*TEND/c0$
```

```
$XMIN = -0.5*PERP_LENGTH
$XMAX =  0.5*PERP_LENGTH
$YMIN = -0.5*PAR_LENGTH
$YMAX =  0.5*PAR_LENGTH
$ZMIN = -0.5*PAR_LENGTH
$ZMAX =  0.5*PAR_LENGTH

$if NDIM==1
$ CFL = 0.5
$ numCells = [PERP_ZONES]
$ periodicDirections = []
$ lowerBounds = [XMIN]
$ upperBounds = [XMAX]
$ else
$if NDIM==2
$ CFL = 0.4
$ numCells = [PERP_ZONES, PAR_ZONES]
# Make the direction perpendicular to the shock
# normal periodic.
$ periodicDirections = [1]
$ lowerBounds = [XMIN,  YMIN]
$ upperBounds = [XMAX,  YMAX]
$ else
$ CFL = 0.32
$ numCells = [PERP_ZONES, PAR_ZONES, PAR_ZONES]
# Make the directions perpendicular to the shock
# normal periodic.
$ periodicDirections = [1,2]
$ lowerBounds = [XMIN,  YMIN,  ZMIN]
$ upperBounds = [XMAX,  YMAX,  ZMAX]
$ endif
$ endif

# Parameters to specify the fluid state at t=0.0
$ DENSITY_L = $3.0*REFERENCE_DENSITY$
$ DENSITY_R = $0.125*REFERENCE_DENSITY$
$ PRESSURE_L = $1.0*REFERENCE_PRESSURE$
$ PRESSURE_R =  $0.1*REFERENCE_PRESSURE$
$ NORMAL_VELOCITY_L = 0.0
$ NORMAL_VELOCITY_R = 0.0
$ PERPENDICULAR_VELOCITY_L = 0.0
$ PERPENDICULAR_VELOCITY_R = 0.0
$ TANGENTIAL_VELOCITY_L = 0.0
$ TANGENTIAL_VELOCITY_R = 0.0

# Initialize a USim simulation
initializeFluidSimulation(NDIM,0.0,END_TIME,NUMDUMPS,CFL,GAS_GAMMA,WRITE_RESTART,DEBUG)

# Setup the grid
addGrid(lowerBounds, upperBounds, numCells, periodicDirections)

# Create data structures needed for the simulation
createFluidSimulation()

# Step 1: Add Variables
addVariable(pi_value,math.pi)
addVariable(gas_gamma,GAS_GAMMA)
addVariable(densityL,DENSITY_L)
```

```
addVariable(densityR,DENSITY_R)
addVariable(pressureL,PRESSURE_L)
addVariable(pressureR,PRESSURE_R)
addVariable(normalVelocityL,NORMAL_VELOCITY_L)
addVariable(normalVelocityR,NORMAL_VELOCITY_R)
addVariable(perpendicularVelocityL,PERPENDICULAR_VELOCITY_L)
addVariable(perpendicularVelocityR,PERPENDICULAR_VELOCITY_R)
addVariable(tangentialVelocityL,TANGENTIAL_VELOCITY_L)
addVariable(tangentialVelocityR,TANGENTIAL_VELOCITY_R)

# Step 2: Add Pre-Expressions
addPreExpression(rho = if (x>0.0, densityR,               densityL))
addPreExpression(pr  = if (x>0.0, pressureR,              pressureL))
addPreExpression(vx  = if (x>0.0, normalVelocityR,        normalVelocityL))
addPreExpression(vy  = if (x>0.0, perpendicularVelocityR, perpendicularVelocityL))
addPreExpression(vz  = if (x>0.0, tangentialVelocityR, tangentialVelocityL))

# Step 3: Add expressions specifying initial condition on density,
# momentum, total energy
addExpression(rho)
addExpression(rho*vx)
addExpression(rho*vy)
addExpression(rho*vz)
addExpression((pr/(gas_gamma-1))+0.5*rho*(vx*vx+vy*vy+vz*vz))
```

3.1.5 Evolving the Fluid

USim implements the well-known MUSCL scheme to advance the conserved variables in time. There is a simple macro that can be called to implement this scheme as shown below:

```
finiteVolumeScheme(DIFFUSIVE)
```

This macro is documented at euler-macro. It's purpose is to compute the numerical flux for the hyperbolic system:

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

The next part of evolving the fluid is to apply boundary conditions at the left and right of the domain to ensure that at the next time step, physically-valid data is used to update the conserved state. Without this, the simulation will fail. It is possible to specify arbitrary boundary conditions in USim. For the Sod shock tube example considered here, appropriate boundary conditions are outflow ("open") boundary conditions at both ends of the domain

```
boundaryCondition(copy,left)
boundaryCondition(copy,right)
```

The copy boundary condition block is described in refmanual-copy. This boundary condition updater copies the values on the layer next to the ghost cells into the ghost cells - this is equivalent to a zero derivative boundary condition. If we are evolving in more than one-dimension, we have to specify boundary conditions on the rest of the domain boundaries. For this example, this is done using periodic boundary conditions (refmanual-periodicCartBc):

```
boundaryCondition(periodic)
```

The final element of advancing the conserved quantities from time t to $t + \Delta t$ is to apply a time integration scheme, specified through:

```
timeAdvance(TIME_ORDER)
```

This applies an explicit Runge-Kutta time-integration scheme with the order of accuracy determined by the *TIME_ORDER* parameter. *TIME_ORDER* can be one of *first*, *second*, *third* or *fourth* according to the desired order of accuracy.

3.1.6 Putting it all Together

The final step in the USim simulation is to add:

```
runFluidSimulation()
```

This tells USim that we're done specifying the simulation and that it can be run. So, our simulation now looks like:

```
# Import macros to setup simulation
$ import fluidsBase.mac
$ import euler.mac

# X-extent of domain
$PAR_LENGTH = 1.0
# Y-extent of domain
$PERP_LENGTH = 1.0
# Zones parallel to shock normal direction
$PAR_ZONES = 256
# Zones perpendicular to shock normal direction
$PERP_ZONES = 256
# Shock tube to solve
$SHOCK_TUBE = "SOD"
# adiabatic index
$GAS_GAMMA = 5.0/3.0
# Atmospheric Pressure
$REFERENCE_PRESSURE = 1.0 # [Pa]
# density of air
$REFERENCE_DENSITY = 1.0 # [kg/m^3]
# end time for simulation (units of # of sound wave crossings)
$TEND = 0.125
# number of frames
$NUMDUMPS = 10
# Whether to use a diffusive (but robust) integration scheme
$DIFFUSIVE = False
# Order in time
$TIME_ORDER = "second"
# Write data for restarting the simulation
$WRITE_RESTART = False
# Output info for debugging purposes
$DEBUG = False
# Default dimensionality
$NDIM = 1

# nominal speed of sound
$ c0 = math.sqrt(GAS_GAMMA*REFERENCE_PRESSURE/REFERENCE_DENSITY) # [m/s]

# Set end time according to (user specified)
# number of times a wave crosses the box.
$END_TIME = $PERP_LENGTH*TEND/c0$

$XMIN = -0.5*PERP_LENGTH
$XMAX =  0.5*PERP_LENGTH
$YMIN = -0.5*PAR_LENGTH
$YMAX =  0.5*PAR_LENGTH
```

```
$ZMIN = -0.5*PAR_LENGTH
$ZMAX =  0.5*PAR_LENGTH

$if NDIM==1
$ CFL = 0.5
$ numCells = [PERP_ZONES]
$ periodicDirections = []
$ lowerBounds = [XMIN]
$ upperBounds = [XMAX]
$ else
$if NDIM==2
$ CFL = 0.4
$ numCells = [PERP_ZONES, PAR_ZONES]
# Make the direction perpendicular to the shock
# normal periodic.
$ periodicDirections = [1]
$ lowerBounds = [XMIN,  YMIN]
$ upperBounds = [XMAX,  YMAX]
$ else
$ CFL = 0.32
$ numCells = [PERP_ZONES, PAR_ZONES, PAR_ZONES]
# Make the directions perpendicular to the shock
# normal periodic.
$ periodicDirections = [1,2]
$ lowerBounds = [XMIN,  YMIN,  ZMIN]
$ upperBounds = [XMAX,  YMAX,  ZMAX]
$ endif
$ endif

# Parameters to specify the fluid state at t=0.0
$ DENSITY_L = $3.0*REFERENCE_DENSITY$
$ DENSITY_R = $0.125*REFERENCE_DENSITY$
$ PRESSURE_L = $1.0*REFERENCE_PRESSURE$
$ PRESSURE_R =  $0.1*REFERENCE_PRESSURE$
$ NORMAL_VELOCITY_L = 0.0
$ NORMAL_VELOCITY_R = 0.0
$ PERPENDICULAR_VELOCITY_L = 0.0
$ PERPENDICULAR_VELOCITY_R = 0.0
$ TANGENTIAL_VELOCITY_L = 0.0
$ TANGENTIAL_VELOCITY_R = 0.0

# Initialize a USim simulation
initializeFluidSimulation(NDIM,0.0,END_TIME,NUMDUMPS,CFL,GAS_GAMMA,WRITE_RESTART,DEBUG)

# Setup the grid
addGrid(lowerBounds, upperBounds, numCells, periodicDirections)

# Create data structures needed for the simulation
createFluidSimulation()

# Step 1: Add Variables
addVariable(pi_value,math.pi)
addVariable(gas_gamma,GAS_GAMMA)
addVariable(densityL,DENSITY_L)
addVariable(densityR,DENSITY_R)
addVariable(pressureL,PRESSURE_L)
addVariable(pressureR,PRESSURE_R)
addVariable(normalVelocityL,NORMAL_VELOCITY_L)
```

```
addVariable(normalVelocityR,NORMAL_VELOCITY_R)
addVariable(perpendicularVelocityL,PERPENDICULAR_VELOCITY_L)
addVariable(perpendicularVelocityR,PERPENDICULAR_VELOCITY_R)
addVariable(tangentialVelocityL,TANGENTIAL_VELOCITY_L)
addVariable(tangentialVelocityR,TANGENTIAL_VELOCITY_R)

# Step 2: Add Pre-Expressions
addPreExpression(rho = if (x>0.0, densityR,               densityL))
addPreExpression(pr  = if (x>0.0, pressureR,              pressureL))
addPreExpression(vx  = if (x>0.0, normalVelocityR,        normalVelocityL))
addPreExpression(vy  = if (x>0.0, perpendicularVelocityR, perpendicularVelocityL))
addPreExpression(vz  = if (x>0.0, tangentialVelocityR, tangentialVelocityL))

# Step 3: Add expressions specifying initial condition on density,
# momentum, total energy
addExpression(rho)
addExpression(rho*vx)
addExpression(rho*vy)
addExpression(rho*vz)
addExpression((pr/(gas_gamma-1))+0.5*rho*(vx*vx+vy*vy+vz*vz))

# Add the spatial discretization of the fluxes
finiteVolumeScheme(DIFFUSIVE)

# Boundary conditions
boundaryCondition(copy,left)
boundaryCondition(copy,right)
boundaryCondition(periodic)

# Time integration
timeAdvance(TIME_ORDER)

# Run the simulation!
runFluidSimulation()
```

Note: For more depth, you can view the actual input blocks to Ulixes in the *Setup* window by choosing *Save And Process Setup* and then clicking on the *shockTube.in* file. In the *.in* file all macros are expanded to produce input blocks.

Most USimBase simulations have a underlying pattern, that can be represented as:

```
# Import macros to setup simulation
$ import fluidsBase.mac
$ import euler.mac

# Specify parameters for the specific physics problem
$ PARAM_1 = <value>
$ PARAM_2 = <value>
$ PARAM_N = <value>

# Initialize a USim simulation
initializeFluidSimulation(NDIM,0.0,END_TIME,NUMDUMPS,CFL,GAS_GAMMA,WRITE_RESTART,DEBUG)

# Setup the grid
addGrid(lowerBounds, upperBounds, numCells, periodicDirections)

# Create data structures needed for the simulation
```

```
createFluidSimulation()

# Specify initial condition
# Step 1: Add Variables
addVariable(NAME,<value>)

# Step 2: Add Pre-Expressions
addPreExpression(<PreExpression>)

# Step 3: Add expressions specifying initial condition on density,
# momentum, total energy
addExpression(<expression>)

# Add the spatial discretization of the fluxes
finiteVolumeScheme(DIFFUSIVE)

# Boundary conditions
boundaryCondition(<boundaryCondition,entity>)

# Time integration
timeAdvance(TIME_ORDER)

# Run the simulation!
runFluidSimulation()
```

We will see this pattern repeated through USimBase.

3.1.7 An Example Simulation

The input file for the problem *Shock Tube* in the USimBase package demonstrates each of the concepts described above to evolve the classic Sod Shock tube problem in one-dimensional hydrodynamics. Executing this input file within USimComposer and switching to the *Visualize* tab yields the plots shown in Fig. 3.1.

3.2 Using USim to solve the Magnetohydrodynamic Equations

In *Using USim to solve the Euler Equations* we discussed the basic methods used by USim to solve the Euler equations. In this tutorial, we show how USim can be used to integrate the magnetohydrodynamic (MHD) equations for problems in one-dimension. This tutorial is based on the quickstart-shocktube example. The *Shock Tube* simulation is designed to set up a variety of tube simulations including those by Einfeldt, Sod, Liska & Wendroff, Brio & Wu, and Ryu & Jones. In this tutorial, we will look at the Brio & Wu shock Tube, described by:

```
Brio, M., & Wu, C.~C. (1988), Journal of Computational Physics, 75, 400
```

Note: While we will be following the quickstart-shocktube example, we will not reproduce the example file in its entirety. The shockTube example is designed to set up a variety of simulations, and by using "Flags" (*if* statements) in the shockTube.pre file, only certain parts of the file are used when simulating the Brio & Wu Shock Tube. Therefore, at the end of this tutorial, our input will not be an exact copy of the shockTube.pre file but will directly represent only the Brio & Wu Shock Tube portion of it.

In this example we will look at the use of equations for ideal compressible magnetohydrodynamics (the *MHD* equations) in one-dimension. It is appropriate to use these equations for transonic, supersonic and hypersonic flows (Mach numbers of 0.1 and above) where compressibility effects are important, at high Reynolds numbers where the effects

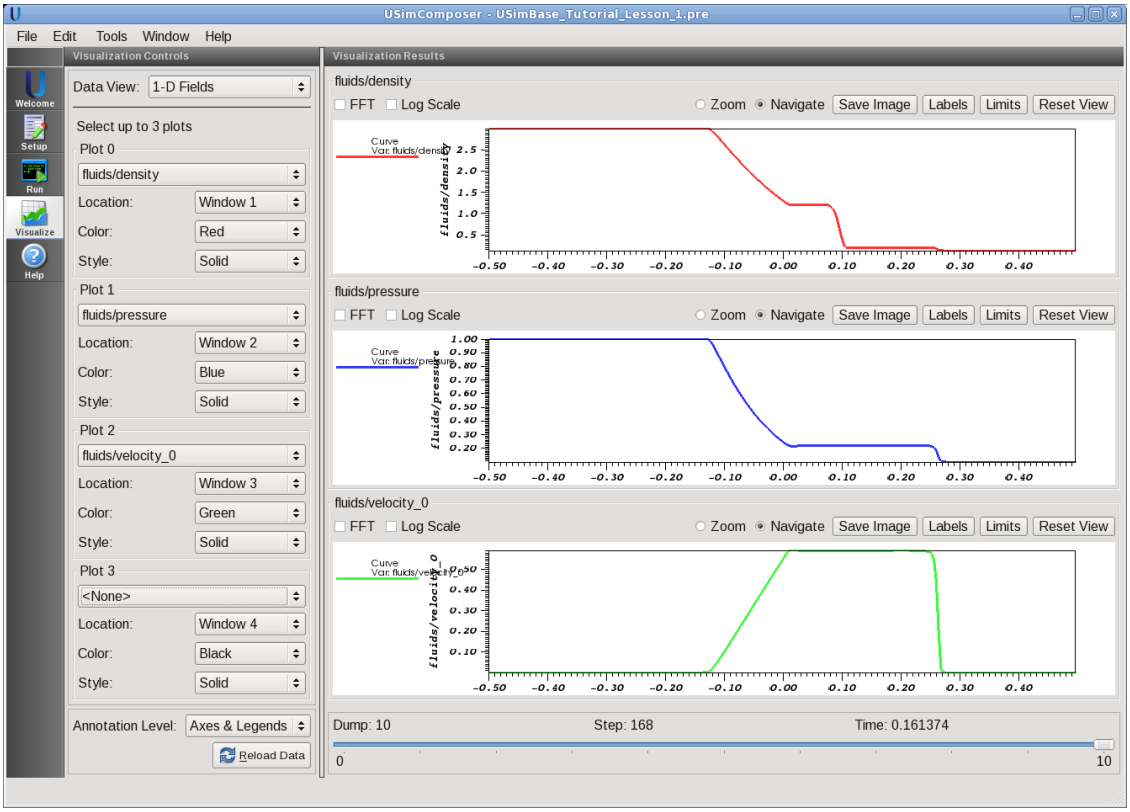


Fig. 3.1: Visualization tab in USimComposer after executing the input file for this lesson.

of viscosity and thermal conductivity are relatively unimportant and at high magnetic Reynolds number where Ohmic resistivity is relatively unimportant.

We will follow the pattern established in *Using USim to solve the Euler Equations* in order that the user can see the differences between solving the MHD equations and the Euler equations. These are summarized in *Notes* at the end of each section of the tutorial.

Contents

3.2.1 The Magnetohydrodynamic Equations

The MHD equations for an adiabatic gas can be written as a hyperbolic conservation law, which has 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), and $\mathbf{w} = \mathbf{w}(\mathbf{q})$. Assuming an adiabatic equations of state, the MHD equations can be written in this form such that:

$$\begin{aligned}
\frac{\partial \rho}{\partial t} + \nabla \cdot \left[\rho\, \mathbf{u}\right] &= 0 \\
\frac{\partial \rho\, \mathbf{u}}{\partial t} + \nabla \cdot \left[\rho\, \mathbf{u}\, \mathbf{u}^T - \mathbf{b}\, \mathbf{b}^T + \mathbb{I}\left(P + \tfrac{1}{2}\left|\mathbf{b}\right|^2\right)\right] &= 0 \\
\frac{\partial E}{\partial t} + \nabla \cdot \left[\left(E + P\right)\mathbf{u} + \mathbf{e} \times \mathbf{b}\right] &= 0 \\
\frac{\partial \mathbf{b}^{\text{plasma}}}{\partial t} + \nabla \times \mathbf{e} + \nabla \psi &= 0 \\
\frac{\partial \psi}{\partial t} + \nabla \cdot \left[c_{\text{fast}}^2 \mathbf{b}\right] &= 0
\end{aligned}$$

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 MHD equations are represented in USim by refmanual-mhdDednerEqn. USim solves these equations by calculating an upwind approximation of the flux tensor, $\mathcal{F}(\mathbf{w})$ (see refmanual-classicMUSCL) and then using this approximation to advance the conserved state from time t to $t + \Delta t$ (see refmanual-multiUpdater).

3.2.2 Solving the MHD Equations in One Dimension

The first step in any USim simulation is to import the macros that are needed to define the simulation. For the *ShockTube* example, there are two macros that are needed:

```
$ import fluidsBase.mac
$ import idealmhd.mac
```

These macros provide basic capabilities for setting up a USim simulation of the ideal MHD Equations. We can now define global parameters that tell USim basic information about the simulation. This is done through the use of the *initializeFluidSimulation* macro:

```
initializeFluidSimulation(NDIM,0.0,END_TIME,NUMDUMPS,CFL,GAS_GAMMA,MU0,WRITE_RESTART,DEBUG)
```

The parameters for the version of the macro used for the refmanual-mhdDednerEqn are documented at idealmhd-macro. For completeness, we include them here, showing how the parameters specified here map onto the parameters used in the *initializeFluidSimulation* macro:

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

`0.0` (tStart): Start time for simulation

`END_TIME` (tEnd): End time for simulation

`NUMDUMPS` (numFrames): Number of data outputs

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

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

`MU0` (muIn): Permeability of free space

`WRITE_RESTART` (writeRestartIn): Output data required for simulation restart

`DEBUG` (debugIn): Run simulation in debug mode

USim includes the ability to perform in place substitution of variables within the input file, as described in *Symbol Definition*. This means that we can define the options given above earlier in the input file and USim will replace them in the *initializeFluidSimulation* macro:

```
# X-extent of domain
$PAR_LENGTH = 1.0
# Y-extent of domain
$PERP_LENGTH = 1.0
# Zones parallel to shock normal direction
$PAR_ZONES = 256
# Zones perpendicular to shock normal direction
$PERP_ZONES = 256
# Shock tube to solve
$SHOCK_TUBE = "BRIOWU"
# adiabatic index
$GAS_GAMMA = 5.0/3.0
# permeability of free-space
$MU0 = 1.0
# Atmospheric Pressure
$REFERENCE_PRESSURE = 1.0 # [Pa]
# density of air
$REFERENCE_DENSITY = 1.0 # [kg/m^3]
# end time for simulation (units of # of sound wave crossings)
$TEND = 0.125
# number of frames
$NUMDUMPS = 10
# Whether to use a diffusive (but robust) integration scheme
$DIFFUSIVE = False
# Order in time
$TIME_ORDER = "second"
```

```
# Write data for restarting the simulation
$WRITE_RESTART = False
# Output info for debugging purposes
$DEBUG = False
# Default dimensionality
$NDIM = 1
```

Note that we have included a range of other variables that are defined in the quickstart-shocktube example. We will refer to these later in this tutorial as needed. In the above, the variable *TEND* is given in units of the the number of times a sound wave crosses the grid. In order to use this to specify the end time for the simulation (*END_TIME*), we convert this to have units of time through:

```
# nominal speed of sound
$ c0 = math.sqrt(GAS_GAMMA*REFERENCE_PRESSURE/REFERENCE_DENSITY) # [m/s]

# Set end time according to (user specified)
# number of times a wave crosses the box.
$END_TIME = $PERP_LENGTH*TEND/c0$
```

The specification of the *CFL* parameter is described in the next section. Our simulation starts off as:

```
# Import macros to setup simulation
$ import fluidsBase.mac
$ import idealmhd.mac

# X-extent of domain
$PAR_LENGTH = 1.0
# Y-extent of domain
$PERP_LENGTH = 1.0
# Zones parallel to shock normal direction
$PAR_ZONES = 256
# Zones perpendicular to shock normal direction
$PERP_ZONES = 256
# Shock tube to solve
$SHOCK_TUBE = "BRIOWU"
# adiabatic index
$GAS_GAMMA = 5.0/3.0
# permeability of free-space
$MU0 = 1.0
# Atmospheric Pressure
$REFERENCE_PRESSURE = 1.0 # [Pa]
# density of air
$REFERENCE_DENSITY = 1.0 # [kg/m^3]
# end time for simulation (units of # of sound wave crossings)
$TEND = 0.125
# number of frames
$NUMDUMPS = 10
# Whether to use a diffusive (but robust) integration scheme
$DIFFUSIVE = False
# Order in time
$TIME_ORDER = "second"
# Write data for restarting the simulation
$WRITE_RESTART = False
# Output info for debugging purposes
$DEBUG = False
# Default dimensionality
$NDIM = 1

# nominal speed of sound
```

```
$ c0 = math.sqrt(GAS_GAMMA*REFERENCE_PRESSURE/REFERENCE_DENSITY) # [m/s]

# Set end time according to (user specified)
# number of times a wave crosses the box.
$END_TIME = $PERP_LENGTH*TEND/c0$

# Initialize a USim simulation
initializeFluidSimulation(NDIM,0.0,END_TIME,NUMDUMPS,CFL,GAS_GAMMA,MU0,WRITE_RESTART,DEBUG)
```

Note: Compared to the Euler equations discussed in *Using USim to solve the Euler Equations*, there are three differences:

1. We have replaced *$ import euler.mac* with *$ import idealmhd.mac* to change the system of equations from the Euler equations to the ideal MHD equations.
2. We have defined the permeability of free-space, μ_0 through the parameter *MU0* (here *MU0 = 1.0*).
3. We have added μ_0 to the parameters used to initialized the simulation, replacing:

```
initializeFluidSimulation(NDIM,0.0,END_TIME,NUMDUMPS,CFL,GAS_GAMMA,WRITE_RESTART,DEBUG)
```

with:

```
initializeFluidSimulation(NDIM,0.0,END_TIME,NUMDUMPS,CFL,GAS_GAMMA,MU0,WRITE_RESTART,DEBUG)
```

3.2.3 Adding a Simulation Grid

The next step in setting up a USim simulation is to specify a simulation grid. The quickstart-shocktube uses a refmanual-ntcart grid, which is added to the simulation through the *addGrid* macro:

```
addGrid(lowerBounds, upperBounds, numCells, periodicDirections)
```

This options for this macro are documented in grid-macro. For completeness, we include them here:

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)
```

The options are setup in the *ShockTube* example according to the dimensionality of the simulation, as specified by the variable *NDIM* above:

```
$XMIN = -0.5*PERP_LENGTH
$XMAX =  0.5*PERP_LENGTH
$YMIN = -0.5*PAR_LENGTH
$YMAX =  0.5*PAR_LENGTH
$ZMIN = -0.5*PAR_LENGTH
$ZMAX =  0.5*PAR_LENGTH

$if NDIM==1
$ CFL = 0.5
```

```
$ numCells = [PERP_ZONES]
$ periodicDirections = []
$ lowerBounds = [XMIN]
$ upperBounds = [XMAX]
$ else
$if NDIM==2
$ CFL = 0.4
$ numCells = [PERP_ZONES, PAR_ZONES]
# Make the direction perpendicular to the shock
# normal periodic.
$ periodicDirections = [1]
$ lowerBounds = [XMIN,  YMIN]
$ upperBounds = [XMAX,  YMAX]
$ else
$ CFL = 0.32
$ numCells = [PERP_ZONES, PAR_ZONES, PAR_ZONES]
# Make the directions perpendicular to the shock
# normal periodic.
$ periodicDirections = [1,2]
$ lowerBounds = [XMIN,  YMIN,  ZMIN]
$ upperBounds = [XMAX,  YMAX,  ZMAX]
$ endif
$ endif
```

Note as well that the CFL condition that is used for the simulation changes according to the dimension of the simulation. This is because the scheme for solving the hyperbolic equations has different stability requirements in different dimensions:

$$\text{CFL} <= \frac{1}{\text{NDIM}}$$

So, our simulation now looks like:

```
# Import macros to setup simulation
$ import fluidsBase.mac
$ import idealmhd.mac

# X-extent of domain
$PAR_LENGTH = 1.0
# Y-extent of domain
$PERP_LENGTH = 1.0
# Zones parallel to shock normal direction
$PAR_ZONES = 256
# Zones perpendicular to shock normal direction
$PERP_ZONES = 256
# Shock tube to solve
$SHOCK_TUBE = "BRIOWU"
# adiabatic index
$GAS_GAMMA = 5.0/3.0
# permeability of free-space
$MU0 = 1.0
# Atmospheric Pressure
$REFERENCE_PRESSURE = 1.0 # [Pa]
# density of air
$REFERENCE_DENSITY = 1.0 # [kg/m^3]
# end time for simulation (units of # of sound wave crossings)
$TEND = 0.125
# number of frames
$NUMDUMPS = 10
```

```
# Whether to use a diffusive (but robust) integration scheme
$DIFFUSIVE = False
# Order in time
$TIME_ORDER = "second"
# Write data for restarting the simulation
$WRITE_RESTART = False
# Output info for debugging purposes
$DEBUG = False
# Default dimensionality
$NDIM = 1

# nominal speed of sound
$ c0 = math.sqrt(GAS_GAMMA*REFERENCE_PRESSURE/REFERENCE_DENSITY) # [m/s]

# Set end time according to (user specified)
# number of times a wave crosses the box.
$END_TIME = $PERP_LENGTH*TEND/c0$

$XMIN = -0.5*PERP_LENGTH
$XMAX =  0.5*PERP_LENGTH
$YMIN = -0.5*PAR_LENGTH
$YMAX =  0.5*PAR_LENGTH
$ZMIN = -0.5*PAR_LENGTH
$ZMAX =  0.5*PAR_LENGTH

$if NDIM==1
$ CFL = 0.5
$ numCells = [PERP_ZONES]
$ periodicDirections = []
$ lowerBounds = [XMIN]
$ upperBounds = [XMAX]
$ else
$if NDIM==2
$ CFL = 0.4
$ numCells = [PERP_ZONES, PAR_ZONES]
# Make the direction perpendicular to the shock
# normal periodic.
$ periodicDirections = [1]
$ lowerBounds = [XMIN,  YMIN]
$ upperBounds = [XMAX,  YMAX]
$ else
$ CFL = 0.32
$ numCells = [PERP_ZONES, PAR_ZONES, PAR_ZONES]
# Make the directions perpendicular to the shock
# normal periodic.
$ periodicDirections = [1,2]
$ lowerBounds = [XMIN,  YMIN,  ZMIN]
$ upperBounds = [XMAX,  YMAX,  ZMAX]
$ endif
$ endif

# Initialize a USim simulation
initializeFluidSimulation(NDIM,0.0,END_TIME,NUMDUMPS,CFL,GAS_GAMMA,MU0,WRITE_RESTART,DEBUG)

# Setup the grid
addGrid(lowerBounds, upperBounds, numCells, periodicDirections)
```

Note: The process of adding the simulation grid is **identical** for both the Euler equations and the MHD equations.

This is true for all USim simulations.

3.2.4 Creating a Fluid Simulation

The next step in setting up a USim simulation is to create the basic set of variables need to simulate the fluid. This is accomplished through the *createFluidSimulation* macro, documented at idealmhd-macro:

```
createFluidSimulation()
```

Now that these variables have been created, it is possible to specify the distribution of fluid on the grid. This is a three-step process:

1. Use *addVariable* to add variables that are independent of grid position.
2. Use *addPreExpression* to add quantities that are functions of grid position, variables and any previously defined PreExpression in this block. Evaluated before expressions and the result is not accessible outside of this block. Any number of PreExpressions can be added.
3. Use *addExpression* to define each initial condition for the fluid. There is one expression for density, each component of momentum and the total energy. The order of the exprssions correspond to the order in the state vector and there can only be one expression per entry in the state vector.

For the Brio & Wu variant of the *ShockTube* example, the variables added in Step 1 in the above process are:

```
addVariable(pi_value,math.pi)
addVariable(gas_gamma,GAS_GAMMA)
addVariable(densityL,DENSITY_L)
addVariable(densityR,DENSITY_R)
addVariable(pressureL,PRESSURE_L)
addVariable(pressureR,PRESSURE_R)
addVariable(normalVelocityL,NORMAL_VELOCITY_L)
addVariable(normalVelocityR,NORMAL_VELOCITY_R)
addVariable(perpendicularVelocityL,PERPENDICULAR_VELOCITY_L)
addVariable(perpendicularVelocityR,PERPENDICULAR_VELOCITY_R)
addVariable(tangentialVelocityL,TANGENTIAL_VELOCITY_L)
addVariable(tangentialVelocityR,TANGENTIAL_VELOCITY_R)
addVariable(mu0,MU0)
addVariable(normalFieldL,NORMAL_FIELD_L)
addVariable(normalFieldR,NORMAL_FIELD_R)
addVariable(perpendicularFieldL,PERPENDICULAR_FIELD_L)
addVariable(perpendicularFieldR,PERPENDICULAR_FIELD_R)
addVariable(tangentialFieldL,TANGENTIAL_FIELD_L)
addVariable(tangentialFieldR,TANGENTIAL_FIELD_R)
```

The variables are then used in the *PreExpression*s that are defined in Step 2:

```
addPreExpression(rho = if (x>0.0, densityR,               densityL))
addPreExpression(pr  = if (x>0.0, pressureR,              pressureL))
addPreExpression(vx  = if (x>0.0, normalVelocityR,        normalVelocityL))
addPreExpression(vy  = if (x>0.0, perpendicularVelocityR, perpendicularVelocityL))
addPreExpression(vz  = if (x>0.0, tangentialVelocityR,    tangentialVelocityL))
addPreExpression(bx  = if (x>0.0, normalFieldR,           normalFieldL))
addPreExpression(by  = if (x>0.0, perpendicularFieldR,    perpendicularFieldL))
addPreExpression(bz  = if (x>0.0, tangentialFieldR,       tangentialFieldL))
addPreExpression(psi = 0.0)
```

Finally, these *PreExpression*s are used to specify the initial conditions defined in Step 3:

```
addExpression(rho)
addExpression(rho*vx)
addExpression(rho*vy)
addExpression(rho*vz)
addExpression(pr/(gas_gamma-1.0) + 0.5*rho*(vx*vx+vy*vy+vz*vz)+0.5*((bx*bx+by*by+bz*bz)/mu0))
addExpression(bx)
addExpression(by)
addExpression(bz)
addExpression(psi)
```

The *ShockTube* example includes many different initial conditions that are based on a range of cases proposed in the literature. The Brio & Wu shock tube is specified through *$SHOCK_TUBE = "BRIOWU"* above. This corresponds to the following variable definitions:

```
$ DENSITY_L = $1.0*REFERENCE_DENSITY$
$ DENSITY_R = $0.125*REFERENCE_DENSITY$
$ PRESSURE_L = $1.0*REFERENCE_PRESSURE$
$ PRESSURE_R =  $0.1*REFERENCE_PRESSURE$
$ NORMAL_VELOCITY_L = 0.0
$ NORMAL_VELOCITY_R = 0.0
$ PERPENDICULAR_VELOCITY_L = 0.0
$ PERPENDICULAR_VELOCITY_R = 0.0
$ TANGENTIAL_VELOCITY_L = 0.0
$ TANGENTIAL_VELOCITY_R = 0.0
$ NORMAL_FIELD_L = $0.75*math.sqrt(REFERENCE_DENSITY)$
$ NORMAL_FIELD_R = $0.75*math.sqrt(REFERENCE_DENSITY)$
$ PERPENDICULAR_FIELD_L = $1.0*math.sqrt(REFERENCE_DENSITY)$
$ PERPENDICULAR_FIELD_R = $-1.0*math.sqrt(REFERENCE_DENSITY)$
$ TANGENTIAL_FIELD_L = 0.0
$ TANGENTIAL_FIELD_R = 0.0
```

Note that this initialization procedure sets up up the following initial condition:

$$\rho = 1.0\rho_0 \quad x \leq 0.0 \tag{3.-5}$$
$$\rho = 0.125\rho_0 \quad x > 0.0 \tag{3.-4}$$
$$P_g = 1.0P_0 \quad x \leq 0.0 \tag{3.-3}$$
$$P_g = 0.1P_0 \quad x > 0.0 \tag{3.-2}$$
$$b_y = \sqrt{\rho_0} \quad x \leq 0.0 \tag{3.-1}$$
$$b_y = -\sqrt{\rho_0} \quad x > 0.0 \tag{3.0}$$
$$b_x = 0.75\sqrt{\rho_0} \tag{3.1}$$

where ρ_0, P_0 are the reference density and pressure respectively. Our simulation now looks like:

```
# Import macros to setup simulation
$ import fluidsBase.mac
$ import idealmhd.mac

# X-extent of domain
$PAR_LENGTH = 1.0
# Y-extent of domain
$PERP_LENGTH = 1.0
# Zones parallel to shock normal direction
$PAR_ZONES = 256
# Zones perpendicular to shock normal direction
$PERP_ZONES = 256
# Shock tube to solve
$SHOCK_TUBE = "BRIOWU"
```

```
# adiabatic index
$GAS_GAMMA = 5.0/3.0
# permeability of free-space
$MU0 = 1.0
# Atmospheric Pressure
$REFERENCE_PRESSURE = 1.0 # [Pa]
# density of air
$REFERENCE_DENSITY = 1.0 # [kg/m^3]
# end time for simulation (units of # of sound wave crossings)
$TEND = 0.125
# number of frames
$NUMDUMPS = 10
# Whether to use a diffusive (but robust) integration scheme
$DIFFUSIVE = False
# Order in time
$TIME_ORDER = "second"
# Write data for restarting the simulation
$WRITE_RESTART = False
# Output info for debugging purposes
$DEBUG = False
# Default dimensionality
$NDIM = 1

# nominal speed of sound
$ c0 = math.sqrt(GAS_GAMMA*REFERENCE_PRESSURE/REFERENCE_DENSITY) # [m/s]

# Set end time according to (user specified)
# number of times a wave crosses the box.
$END_TIME = $PERP_LENGTH*TEND/c0$

$XMIN = -0.5*PERP_LENGTH
$XMAX =  0.5*PERP_LENGTH
$YMIN = -0.5*PAR_LENGTH
$YMAX =  0.5*PAR_LENGTH
$ZMIN = -0.5*PAR_LENGTH
$ZMAX =  0.5*PAR_LENGTH

$if NDIM==1
$ CFL = 0.5
$ numCells = [PERP_ZONES]
$ periodicDirections = []
$ lowerBounds = [XMIN]
$ upperBounds = [XMAX]
$ else
$if NDIM==2
$ CFL = 0.4
$ numCells = [PERP_ZONES, PAR_ZONES]
# Make the direction perpendicular to the shock
# normal periodic.
$ periodicDirections = [1]
$ lowerBounds = [XMIN,  YMIN]
$ upperBounds = [XMAX,  YMAX]
$ else
$ CFL = 0.32
$ numCells = [PERP_ZONES, PAR_ZONES, PAR_ZONES]
# Make the directions perpendicular to the shock
# normal periodic.
$ periodicDirections = [1,2]
```

```
$ lowerBounds = [XMIN,  YMIN,  ZMIN]
$ upperBounds = [XMAX,  YMAX,  ZMAX]
$ endif
$ endif

# Parameters to specify the fluid state at t=0.0
$ DENSITY_L = $1.0*REFERENCE_DENSITY$
$ DENSITY_R = $0.125*REFERENCE_DENSITY$
$ PRESSURE_L = $1.0*REFERENCE_PRESSURE$
$ PRESSURE_R =  $0.1*REFERENCE_PRESSURE$
$ NORMAL_VELOCITY_L = 0.0
$ NORMAL_VELOCITY_R = 0.0
$ PERPENDICULAR_VELOCITY_L = 0.0
$ PERPENDICULAR_VELOCITY_R = 0.0
$ TANGENTIAL_VELOCITY_L = 0.0
$ TANGENTIAL_VELOCITY_R = 0.0
$ NORMAL_FIELD_L = $0.75*math.sqrt(REFERENCE_DENSITY)$
$ NORMAL_FIELD_R = $0.75*math.sqrt(REFERENCE_DENSITY)$
$ PERPENDICULAR_FIELD_L = $1.0*math.sqrt(REFERENCE_DENSITY)$
$ PERPENDICULAR_FIELD_R = $-1.0*math.sqrt(REFERENCE_DENSITY)$
$ TANGENTIAL_FIELD_L = 0.0
$ TANGENTIAL_FIELD_R = 0.0

# Initialize a USim simulation
initializeFluidSimulation(NDIM,0.0,END_TIME,NUMDUMPS,CFL,GAS_GAMMA,MU0,WRITE_RESTART,DEBUG)

# Setup the grid
addGrid(lowerBounds, upperBounds, numCells, periodicDirections)

# Create data structures needed for the simulation
createFluidSimulation()

# Step 1: Add Variables
addVariable(pi_value,math.pi)
addVariable(gas_gamma,GAS_GAMMA)
addVariable(densityL,DENSITY_L)
addVariable(densityR,DENSITY_R)
addVariable(pressureL,PRESSURE_L)
addVariable(pressureR,PRESSURE_R)
addVariable(normalVelocityL,NORMAL_VELOCITY_L)
addVariable(normalVelocityR,NORMAL_VELOCITY_R)
addVariable(perpendicularVelocityL,PERPENDICULAR_VELOCITY_L)
addVariable(perpendicularVelocityR,PERPENDICULAR_VELOCITY_R)
addVariable(tangentialVelocityL,TANGENTIAL_VELOCITY_L)
addVariable(tangentialVelocityR,TANGENTIAL_VELOCITY_R)
addVariable(mu0,MU0)
addVariable(normalFieldL,NORMAL_FIELD_L)
addVariable(normalFieldR,NORMAL_FIELD_R)
addVariable(perpendicularFieldL,PERPENDICULAR_FIELD_L)
addVariable(perpendicularFieldR,PERPENDICULAR_FIELD_R)
addVariable(tangentialFieldL,TANGENTIAL_FIELD_L)
addVariable(tangentialFieldR,TANGENTIAL_FIELD_R)

# Step 2: Add Pre-Expressions
addPreExpression(rho = if (x>0.0, densityR,               densityL))
addPreExpression(pr  = if (x>0.0, pressureR,              pressureL))
addPreExpression(vx  = if (x>0.0, normalVelocityR,        normalVelocityL))
addPreExpression(vy  = if (x>0.0, perpendicularVelocityR, perpendicularVelocityL))
```

```
addPreExpression(vz  = if (x>0.0, tangentialVelocityR,    tangentialVelocityL))
addPreExpression(bx  = if (x>0.0, normalFieldR,           normalFieldL))
addPreExpression(by  = if (x>0.0, perpendicularFieldR,    perpendicularFieldL))
addPreExpression(bz  = if (x>0.0, tangentialFieldR,       tangentialFieldL))
addPreExpression(psi = 0.0)

# Step 3: Add expressions specifying initial condition on density,
# momentum, total energy, magnetic field and correction potential
addExpression(rho)
addExpression(rho*vx)
addExpression(rho*vy)
addExpression(rho*vz)
addExpression(pr/(gas_gamma-1.0) + 0.5*rho*(vx*vx+vy*vy+vz*vz)+0.5*((bx*bx+by*by+bz*bz)/mu0))
addExpression(bx)
addExpression(by)
addExpression(bz)
addExpression(psi)
```

Note: Compared to the Euler equations discussed in *Using USim to solve the Euler Equations*, there are three important differences in specifying the initial conditions:

1. We have to specify the three components of the magnetic field, here these are denoted $b_{\{x,y,z\}}$.
2. We have to specify the correction potential, here denoted, *psi*. Typically, $\psi = 0$ at $t = 0$.
3. We have to change the definition of the total energy from $E = \frac{P}{\Gamma-1} + \frac{\rho|\mathbf{u}|^2}{2}$ to $E = \frac{P}{\Gamma-1} + \frac{\rho|\mathbf{u}|^2}{2} + \frac{|\mathbf{b}|^2}{2}$; that is we include the contribution of the magnetic field to the total energy

3.2.5 Evolving the Fluid

USim implements the well-known MUSCL scheme to advance the conserved variables in time. There is a simple macro that can be called to implement this scheme as shown below:

```
finiteVolumeScheme(DIFFUSIVE)
```

This macro is documented at idealmhd-macro. It's purpose is to compute the numerical flux for the hyperbolic system:

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

The next part of evolving the fluid is to apply boundary conditions at the left and right of the domain to ensure that at the next time step, physically-valid data is used to update the conserved state. Without this, the simulation will fail. It is possible to specify arbitrary boundary conditions in USim. For the Brio & Wu shock tube example considered here, appropriate boundary conditions are outflow ("open") boundary conditions at both ends of the domain

```
boundaryCondition(copy,left)
boundaryCondition(copy,right)
```

The copy boundary condition block is described in refmanual-copy. This boundary condition updater copies the values on the layer next to the ghost cells into the ghost cells - this is equivalent to a zero derivative boundary condition. If we are evolving in more than one-dimension, we have to specify boundary conditions on the rest of the domain boundaries. For this example, this is done using periodic boundary conditions (refmanual-periodicCartBc):

```
boundaryCondition(periodic)
```

The final element of advancing the conserved quantities from time t to $t + \Delta t$ is to apply a time integration scheme, specified through:

```
timeAdvance(TIME_ORDER)
```

This applies an explicit Runge-Kutta time-integration scheme with the order of accuracy determined by the *TIME_ORDER* parameter. *TIME_ORDER* can be one of *first*, *second*, *third* or *fourth* according to the desired order of accuracy.

Note: The method for evolving the fluid is **identical** for the Euler equations and the MHD equations in one-dimension.

3.2.6 Putting it all Together

The final step in the USim simulation is to add:

```
runFluidSimulation()
```

This tells USim that we're done specifying the simulation and that it can be run. So, our simulation now looks like:

```
# Import macros to setup simulation
$ import fluidsBase.mac
$ import idealmhd.mac

# X-extent of domain
$PAR_LENGTH = 1.0
# Y-extent of domain
$PERP_LENGTH = 1.0
# Zones parallel to shock normal direction
$PAR_ZONES = 256
# Zones perpendicular to shock normal direction
$PERP_ZONES = 256
# Shock tube to solve
$SHOCK_TUBE = "BRIOWU"
# adiabatic index
$GAS_GAMMA = 5.0/3.0
# permeability of free-space
$MU0 = 1.0
# Atmospheric Pressure
$REFERENCE_PRESSURE = 1.0 # [Pa]
# density of air
$REFERENCE_DENSITY = 1.0 # [kg/m^3]
# end time for simulation (units of # of sound wave crossings)
$TEND = 0.125
# number of frames
$NUMDUMPS = 10
# Whether to use a diffusive (but robust) integration scheme
$DIFFUSIVE = False
# Order in time
$TIME_ORDER = "second"
# Write data for restarting the simulation
$WRITE_RESTART = False
# Output info for debugging purposes
$DEBUG = False
# Default dimensionality
$NDIM = 1

# nominal speed of sound
$ c0 = math.sqrt(GAS_GAMMA*REFERENCE_PRESSURE/REFERENCE_DENSITY) # [m/s]

# Set end time according to (user specified)
```

```
# number of times a wave crosses the box.
$END_TIME = $PERP_LENGTH*TEND/c0$

$XMIN = -0.5*PERP_LENGTH
$XMAX =  0.5*PERP_LENGTH
$YMIN = -0.5*PAR_LENGTH
$YMAX =  0.5*PAR_LENGTH
$ZMIN = -0.5*PAR_LENGTH
$ZMAX =  0.5*PAR_LENGTH

$if NDIM==1
$ CFL = 0.5
$ numCells = [PERP_ZONES]
$ periodicDirections = []
$ lowerBounds = [XMIN]
$ upperBounds = [XMAX]
$ else
$if NDIM==2
$ CFL = 0.4
$ numCells = [PERP_ZONES, PAR_ZONES]
# Make the direction perpendicular to the shock
# normal periodic.
$ periodicDirections = [1]
$ lowerBounds = [XMIN,  YMIN]
$ upperBounds = [XMAX,  YMAX]
$ else
$ CFL = 0.32
$ numCells = [PERP_ZONES, PAR_ZONES, PAR_ZONES]
# Make the directions perpendicular to the shock
# normal periodic.
$ periodicDirections = [1,2]
$ lowerBounds = [XMIN,  YMIN,  ZMIN]
$ upperBounds = [XMAX,  YMAX,  ZMAX]
$ endif
$ endif

# Parameters to specify the fluid state at t=0.0
$ DENSITY_L = $1.0*REFERENCE_DENSITY$
$ DENSITY_R = $0.125*REFERENCE_DENSITY$
$ PRESSURE_L = $1.0*REFERENCE_PRESSURE$
$ PRESSURE_R =  $0.1*REFERENCE_PRESSURE$
$ NORMAL_VELOCITY_L = 0.0
$ NORMAL_VELOCITY_R = 0.0
$ PERPENDICULAR_VELOCITY_L = 0.0
$ PERPENDICULAR_VELOCITY_R = 0.0
$ TANGENTIAL_VELOCITY_L = 0.0
$ TANGENTIAL_VELOCITY_R = 0.0
$ NORMAL_FIELD_L = $0.75*math.sqrt(REFERENCE_DENSITY)$
$ NORMAL_FIELD_R = $0.75*math.sqrt(REFERENCE_DENSITY)$
$ PERPENDICULAR_FIELD_L = $1.0*math.sqrt(REFERENCE_DENSITY)$
$ PERPENDICULAR_FIELD_R = $-1.0*math.sqrt(REFERENCE_DENSITY)$
$ TANGENTIAL_FIELD_L = 0.0
$ TANGENTIAL_FIELD_R = 0.0

# Initialize a USim simulation
initializeFluidSimulation(NDIM,0.0,END_TIME,NUMDUMPS,CFL,GAS_GAMMA,MU0,WRITE_RESTART,DEBUG)

# Setup the grid
```

```
addGrid(lowerBounds, upperBounds, numCells, periodicDirections)

# Create data structures needed for the simulation
createFluidSimulation()

# Step 1: Add Variables
addVariable(pi_value,math.pi)
addVariable(gas_gamma,GAS_GAMMA)
addVariable(densityL,DENSITY_L)
addVariable(densityR,DENSITY_R)
addVariable(pressureL,PRESSURE_L)
addVariable(pressureR,PRESSURE_R)
addVariable(normalVelocityL,NORMAL_VELOCITY_L)
addVariable(normalVelocityR,NORMAL_VELOCITY_R)
addVariable(perpendicularVelocityL,PERPENDICULAR_VELOCITY_L)
addVariable(perpendicularVelocityR,PERPENDICULAR_VELOCITY_R)
addVariable(tangentialVelocityL,TANGENTIAL_VELOCITY_L)
addVariable(tangentialVelocityR,TANGENTIAL_VELOCITY_R)
addVariable(mu0,MU0)
addVariable(normalFieldL,NORMAL_FIELD_L)
addVariable(normalFieldR,NORMAL_FIELD_R)
addVariable(perpendicularFieldL,PERPENDICULAR_FIELD_L)
addVariable(perpendicularFieldR,PERPENDICULAR_FIELD_R)
addVariable(tangentialFieldL,TANGENTIAL_FIELD_L)
addVariable(tangentialFieldR,TANGENTIAL_FIELD_R)

# Step 2: Add Pre-Expressions
 addPreExpression(rho = if (x>0.0, densityR,                densityL))
addPreExpression(pr  = if (x>0.0, pressureR,              pressureL))
addPreExpression(vx  = if (x>0.0, normalVelocityR,        normalVelocityL))
addPreExpression(vy  = if (x>0.0, perpendicularVelocityR, perpendicularVelocityL))
addPreExpression(vz  = if (x>0.0, tangentialVelocityR,    tangentialVelocityL))
addPreExpression(bx  = if (x>0.0, normalFieldR,           normalFieldL))
addPreExpression(by  = if (x>0.0, perpendicularFieldR,    perpendicularFieldL))
addPreExpression(bz  = if (x>0.0, tangentialFieldR,       tangentialFieldL))
addPreExpression(psi = 0.0)

# Step 3: Add expressions specifying initial condition on density,
# momentum, total energy, magnetic field and correction potential
addExpression(rho)
addExpression(rho*vx)
addExpression(rho*vy)
addExpression(rho*vz)
addExpression(pr/(gas_gamma-1.0) + 0.5*rho*(vx*vx+vy*vy+vz*vz)+0.5*((bx*bx+by*by+bz*bz)/mu0))
addExpression(bx)
addExpression(by)
addExpression(bz)
addExpression(psi)

# Add the spatial discretization of the fluxes
finiteVolumeScheme(DIFFUSIVE)

# Boundary conditions
boundaryCondition(copy,left)
boundaryCondition(copy,right)
boundaryCondition(periodic)

# Time integration
```

```
timeAdvance(TIME_ORDER)

# Run the simulation!
runFluidSimulation()
```

Note: For more depth, you can view the actual input blocks to Ulixes in the *Setup* window by choosing *Save And Process Setup* and then clicking on the *shockTube.in* file. In the *.in* file all macros are expanded to produce input blocks.

3.2.7 An Example Simulation

The input file for the quickstart-shocktube example for the USimBase package with *SHOCK_TUBE = "BRIOWU"* demonstrates each of the concepts described above to evolve the classic Brio & Wu shock tube problem in one-dimensional magnetohydrodynamics. Executing this input file within USimComposer and switching to the *Visualize* tab yields the plots shown in Fig. 3.2.

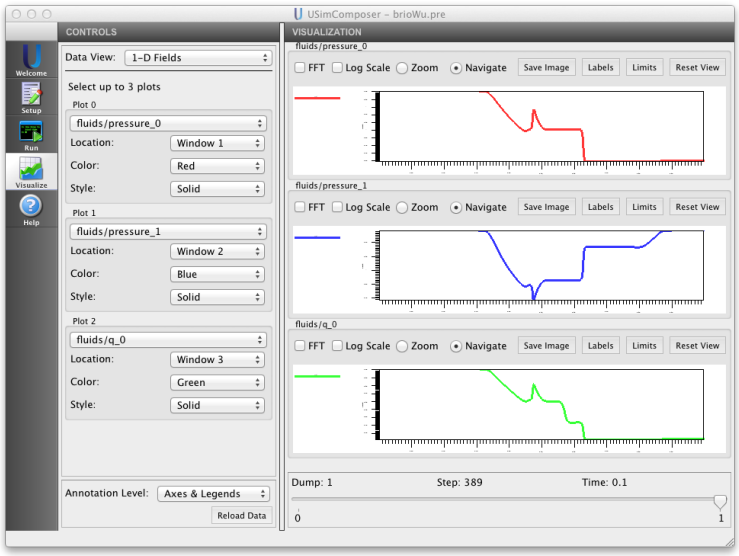


Fig. 3.2: Visualization tab in USimComposer after executing the *Brio & Wu shock tube* input file for the tutorial.

3.2.8 Combining Euler and MHD Equations in the Same Input File

In *Using USim to solve the Euler Equations* we saw how most USimBase simulations for the Euler equations have an underlying pattern. Based on what we have discussed for the MHD equations, we can now extend this pattern to easily switch between the Euler and MHD equations:

```
# Are we solving the MHD equations?
$ MHD = True

# Import macros to setup simulation
$ import fluidsBase.mac
# if MHD
$ import idealmhd.mac
$ else
```

```
$ import euler.mac
$ endif

# Specify parameters for the specific physics problem
$ PARAM_1 = <value>
$ PARAM_2 = <value>
$ PARAM_N = <value>

# Initialize a USim simulation
$ if MHD
initializeFluidSimulation(NDIM,0.0,END_TIME,NUMDUMPS,CFL,GAS_GAMMA,MU0,WRITE_RESTART,DEBUG)
$ else
initializeFluidSimulation(NDIM,0.0,END_TIME,NUMDUMPS,CFL,GAS_GAMMA,WRITE_RESTART,DEBUG)
$ endif

# Setup the grid
addGrid(lowerBounds, upperBounds, numCells, periodicDirections)

# Create data structures needed for the simulation
createFluidSimulation()

# Specify initial condition
# Step 1: Add Variables
addVariable(NAME,<value>)

# Step 2: Add Pre-Expressions
addPreExpression(<PreExpression>)

# Step 3: a) Add expressions specifying initial condition on density,
# momentum
addExpression(<expression>)
$ if MHD
# Step 3: b) Add expression specifying initial conditions on total
# energy, magnetic field, correction potential
addExpression(<expression>)
$ else
# Step 3: b) Add expression specifying initial conditions on total
# energy
addExpression(<expression>)
$ endif

# Add the spatial discretization of the fluxes
finiteVolumeScheme(DIFFUSIVE)

# Boundary conditions
boundaryCondition(<boundaryCondition,entity>)

# Time integration
timeAdvance(TIME_ORDER)

# Run the simulation!
runFluidSimulation()
```

You can see this pattern demonstrated in the quickstart-shocktube example.

3.3 Solving Multi-Dimensional Problems in USim

In *Using USim to solve the Euler Equations* we discussed the basic methods used by USim to solve the Euler equations. Next, in *Using USim to solve the Magnetohydrodynamic Equations*, we extended these ideas to solve the MHD equations in one-dimension and showed how USimBase simulations for both the Euler and MHD equations follow the same basic pattern. In this tutorial, we continue to build on these concepts and demonstrate how to use USim to solve the Euler and MHD equations in multi-dimensions, how to utilize more advanced boundary conditions and how to apply external fources (such as gravity) to the equations.

This tutorial is based on quickstart-rtinstability in USimBase, which demonstrates the well-known Rayleigh-Taylor instability problem described by:

```
Jun, Norman, & Stone, ApJ 453, 332 (1995).
```

Contents

3.3.1 Initializing the Simulation

Compared to the *ShockTube* example used in *Using USim to solve the Euler Equations*, the initialization of the simulation proceeds in a similar fashion, with parameters customized for the Rayleigh-Taylor problem:

```
# Import macros to setup simulation
$ import fluidsBase.mac
$ import euler.mac

# X-extent of domain
$ PAR_LENGTH = 1.5
# Y-extent of domain
$ TRANS_LENGTH = 0.5
# Zones parallel to shear direction
$ PAR_ZONES = 192
# Zones perpendicular to shear direction
$ TRANS_ZONES = 64
# adiabatic index
$ GAS_GAMMA = 1.4
# acceleration due to gravity
$ GRAVITY_ACCEL = 0.1
# Upper fluid density
$ RHO_HEAVY = 2.0
# Lower fluid density
$ RHO_LIGHT = 1.0
# Magnetic field strength
$ BETA = 1.0e3
# Amplitude of perturbation
$ PERTURB_AMP = 0.01
```

```
# end time for simulation
$ TEND = 12.75
# number of frames
$ NUMDUMPS = 10
# Whether to use diffusive (but robust) fluxes
$ DIFFUSIVE = False
# Order in time
$ TIME_ORDER = "second"
# Write data for restarting the simulation
$ WRITE_RESTART = False
# Output info for debugging purposes
$ DEBUG = False
# Default dimensionality
$ NDIM = 2

# Initialize a USim simulation
initializeFluidSimulation(NDIM,0.0,TEND,NUMDUMPS,CFL,GAS_GAMMA,WRITE_RESTART,DEBUG)
```

3.3.2 Adding a Simulation Grid

Compared to the *ShockTube* example used in *Using USim to solve the Euler Equations*, adding a grid again proceeds in a similar fashion, with parameters customized for the Rayleigh-Taylor problem:

```
$XMIN = -0.5*TRANS_LENGTH
$XMAX =  0.5*TRANS_LENGTH

$YMIN = -0.5*PAR_LENGTH
$YMAX =  0.5*PAR_LENGTH

$ZMIN = -0.5*TRANS_LENGTH
$ZMAX =  0.5*TRANS_LENGTH

$if NDIM==2
$ CFL = 0.4
$ numCells = [TRANS_ZONES, PAR_ZONES]
$ periodicDirections = [0]
$ lowerBounds = [XMIN,  YMIN]
$ upperBounds = [XMAX,  YMAX]
$ else
$ CFL = 0.32
$ numCells = [TRANS_ZONES, PAR_ZONES, TRANS_ZONES]
$ periodicDirections = [0 2]
$ lowerBounds = [XMIN,  YMIN,  ZMIN]
$ upperBounds = [XMAX,  YMAX,  ZMAX]
$ endif

addGrid(lowerBounds, upperBounds, numCells, periodicDirections)
```

Note that directions *0* (x) and (in three-dimensions) *2* (z) are periodic. Our simulation now looks like:

```
# Import macros to setup simulation
$ import fluidsBase.mac
$ import euler.mac

# X-extent of domain
$ PAR_LENGTH = 1.5
# Y-extent of domain
```

```
$ TRANS_LENGTH = 0.5
# Zones parallel to shear direction
$ PAR_ZONES = 192
# Zones perpendicular to shear direction
$ TRANS_ZONES = 64
# adiabatic index
$ GAS_GAMMA = 1.4
# acceleration due to gravity
$ GRAVITY_ACCEL = 0.1
# Upper fluid density
$ RHO_HEAVY = 2.0
# Lower fluid density
$ RHO_LIGHT = 1.0
# Magnetic field strength
$ BETA = 1.0e3
# Amplitude of perturbation
$ PERTURB_AMP = 0.01
# end time for simulation
$ TEND = 12.75
# number of frames
$ NUMDUMPS = 10
# Whether to use diffusive (but robust) fluxes
$ DIFFUSIVE = False
# Order in time
$ TIME_ORDER = "second"
# Write data for restarting the simulation
$ WRITE_RESTART = False
# Output info for debugging purposes
$ DEBUG = False
# Default dimensionality
$ NDIM = 2

$XMIN = -0.5*TRANS_LENGTH
$XMAX =  0.5*TRANS_LENGTH

$YMIN = -0.5*PAR_LENGTH
$YMAX =  0.5*PAR_LENGTH

$ZMIN = -0.5*TRANS_LENGTH
$ZMAX =  0.5*TRANS_LENGTH

$if NDIM==2
$ CFL = 0.4
$ numCells = [TRANS_ZONES, PAR_ZONES]
$ periodicDirections = [0]
$ lowerBounds = [XMIN,  YMIN]
$ upperBounds = [XMAX,  YMAX]
$ else
$ CFL = 0.32
$ numCells = [TRANS_ZONES, PAR_ZONES, TRANS_ZONES]
$ periodicDirections = [0 2]
$ lowerBounds = [XMIN,  YMIN,  ZMIN]
$ upperBounds = [XMAX,  YMAX,  ZMAX]
$ endif

# Initialize a USim simulation
initializeFluidSimulation(NDIM,0.0,TEND,NUMDUMPS,CFL,GAS_GAMMA,WRITE_RESTART,DEBUG)
```

```
# Setup the grid
addGrid(lowerBounds, upperBounds, numCells, periodicDirections)
```

3.3.3 Creating a Fluid Simulation

This initial condition for this problem is hydrostatic equilibrium consisting of a heavier fluid supported by a lighter fluid, which is perturbed with a single mode. Initialization follows the same pattern as in *Using USim to solve the Euler Equations*. First, we create the variables needed to simulate the fluid:

```
createFluidSimulation()
```

We then proceed through the three-step process to specify the distribution of the fluid on the grid. Step 1: Add Variables:

```
addVariable(gas_gamma,GAS_GAMMA)
addVariable(rhoTop,RHO_HEAVY)
addVariable(rhoBottom,RHO_LIGHT)
addVariable(perturb,PERTURB_AMP)
addVariable(ytop,YMAX)
addVariable(gravity,GRAVITY_ACCEL)
addVariable(lx,TRANS_LENGTH)
addVariable(ly,PAR_LENGTH)
```

Step 2: Add Pre Expression's:

```
addPreExpression(p0 = 0.01)
addPreExpression(pert = 0.01)
addPreExpression(pi = 3.14159)
addPreExpression(rho = if (y < 0.0, rhoBottom, rhoTop))
addPreExpression(pr = (1.0/gas_gamma) - (gravity*rho*y))
addPreExpression(vx = 0.0)
addPreExpression(vy = (0.25*perturb)*(1.0 + cos(2.0*pi*x/lx))*(1.0 + cos(2.0*pi*y/ly)))
addPreExpression(vz = 0.0)
```

Step 3: Add Expressions for density, momentum and total energy:

```
addExpression(rho)
addExpression(rho*vx)
addExpression(rho*vy)
addExpression(rho*vz)
addExpression((pr/(gas_gamma-1.0))+0.5*rho*(vx*vx+vy*vy+vz*vz))
```

Note that Step 3 is **identical** to that used in *Using USim to solve the Euler Equations*. Our simulation now looks like:

```
# Import macros to setup simulation
$ import fluidsBase.mac
$ import euler.mac

# X-extent of domain
$ PAR_LENGTH = 1.5
# Y-extent of domain
$ TRANS_LENGTH = 0.5
# Zones parallel to shear direction
$ PAR_ZONES = 192
# Zones perpendicular to shear direction
$ TRANS_ZONES = 64
# adiabatic index
$ GAS_GAMMA = 1.4
```

```
# acceleration due to gravity
$ GRAVITY_ACCEL = 0.1
# Upper fluid density
$ RHO_HEAVY = 2.0
# Lower fluid density
$ RHO_LIGHT = 1.0
# Magnetic field strength
$ BETA = 1.0e3
# Amplitude of perturbation
$ PERTURB_AMP = 0.01
# end time for simulation
$ TEND = 12.75
# number of frames
$ NUMDUMPS = 10
# Whether to use diffusive (but robust) fluxes
$ DIFFUSIVE = False
# Order in time
$ TIME_ORDER = "second"
# Write data for restarting the simulation
$ WRITE_RESTART = False
# Output info for debugging purposes
$ DEBUG = False
# Default dimensionality
$ NDIM = 2

$XMIN = -0.5*TRANS_LENGTH
$XMAX =  0.5*TRANS_LENGTH

$YMIN = -0.5*PAR_LENGTH
$YMAX =  0.5*PAR_LENGTH

$ZMIN = -0.5*TRANS_LENGTH
$ZMAX =  0.5*TRANS_LENGTH

$if NDIM==2
$ CFL = 0.4
$ numCells = [TRANS_ZONES, PAR_ZONES]
$ periodicDirections = [0]
$ lowerBounds = [XMIN,  YMIN]
$ upperBounds = [XMAX,  YMAX]
$ else
$ CFL = 0.32
$ numCells = [TRANS_ZONES, PAR_ZONES, TRANS_ZONES]
$ periodicDirections = [0 2]
$ lowerBounds = [XMIN,  YMIN,  ZMIN]
$ upperBounds = [XMAX,  YMAX,  ZMAX]
$ endif

# Initialize a USim simulation
initializeFluidSimulation(NDIM,0.0,TEND,NUMDUMPS,CFL,GAS_GAMMA,WRITE_RESTART,DEBUG)

# Setup the grid
addGrid(lowerBounds, upperBounds, numCells, periodicDirections)

# Create data structures needed for the simulation
createFluidSimulation()

# Step 1: Add Variables
```

```
addVariable(gas_gamma,GAS_GAMMA)
addVariable(rhoTop,RHO_HEAVY)
addVariable(rhoBottom,RHO_LIGHT)
addVariable(perturb,PERTURB_AMP)
addVariable(ytop,YMAX)
addVariable(gravity,GRAVITY_ACCEL)
addVariable(lx,TRANS_LENGTH)
addVariable(ly,PAR_LENGTH)

# Step 2: Add Pre-Expressions
addPreExpression(p0 = 0.01)
addPreExpression(pert = 0.01)
addPreExpression(pi = 3.14159)
addPreExpression(rho = if (y < 0.0, rhoBottom, rhoTop))
addPreExpression(pr = (1.0/gas_gamma) - (gravity*rho*y))
addPreExpression(vx = 0.0)
addPreExpression(vy = (0.25*perturb)*(1.0 + cos(2.0*pi*x/lx))*(1.0 + cos(2.0*pi*y/ly)))
addPreExpression(vz = 0.0)

# Step 3: Add expressions specifying initial condition on density,
# momentum, total energy
addExpression(rho)
addExpression(rho*vx)
addExpression(rho*vy)
addExpression(rho*vz)
addExpression((pr/(gas_gamma-1))+0.5*rho*(vx*vx+vy*vy+vz*vz))
```

Compared to the simulation described in *Using USim to solve the Euler Equations*, the similarities should be obvious as the simulation is following the pattern:

```
# Specify parameters for the specific physics problem
$ PARAM_1 = <value>
$ PARAM_2 = <value>
$ PARAM_N = <value>

# Initialize a USim simulation
initializeFluidSimulation(NDIM,0.0,END_TIME,NUMDUMPS,CFL,GAS_GAMMA,WRITE_RESTART,DEBUG)

# Setup the grid
addGrid(lowerBounds, upperBounds, numCells, periodicDirections)

# Create data structures needed for the simulation
createFluidSimulation()

# Specify initial condition
# Step 1: Add Variables
addVariable(NAME,<value>)

# Step 2: Add Pre-Expressions
addPreExpression(<PreExpression>)

# Step 3: Add expressions specifying initial condition on density,
# momentum, total energy
addExpression(<expression>)
```

3.3.4 Evolving the Fluid

There are two major differences in simulating the Rayleigh-Taylor problem compared to a shock tube: acceleration due to gravity and boundary conditions. These change the scheme for integrating the hyperbolic conservation law, which now looks like:

```
#
# Add the spatial discretization of the fluxes
finiteVolumeScheme(DIFFUSIVE)

#
# Add source term for gravitational acceleration to the equations
addGravitationalAcceleration(GRAVITY_ACCEL)

#
# Boundary conditions
boundaryCondition(wall,top)
boundaryCondition(wall,bottom)
boundaryCondition(periodic)

#
# Time integration
timeAdvance(TIME_ORDER)
```

The first major difference is the addition of a gravitational acceleration. This is done through a simple macro call after we have defined the finiteVolumeScheme:

```
addGravitationalAcceleration(GRAVITY_ACCEL)
```

The next is that we add *wall* or *reflecting* boundary conditions at the top and bottom of the grid. These boundary conditions are documented at refmanual-eulerBc:

```
boundaryCondition(wall,top)
boundaryCondition(wall,bottom)
```

3.3.5 Putting it all Together

As before, the final step in the USim simulation is to add:

```
runFluidSimulation()
```

So, our simulation now looks like:

```
# Import macros to setup simulation
$ import fluidsBase.mac
$ import euler.mac

# X-extent of domain
$ PAR_LENGTH = 1.5
# Y-extent of domain
$ TRANS_LENGTH = 0.5
# Zones parallel to shear direction
$ PAR_ZONES = 192
# Zones perpendicular to shear direction
$ TRANS_ZONES = 64
# adiabatic index
$ GAS_GAMMA = 1.4
# acceleration due to gravity
```

```
$ GRAVITY_ACCEL = 0.1
# Upper fluid density
$ RHO_HEAVY = 2.0
# Lower fluid density
$ RHO_LIGHT = 1.0
# Magnetic field strength
$ BETA = 1.0e3
# Amplitude of perturbation
$ PERTURB_AMP = 0.01
# end time for simulation
$ TEND = 12.75
# number of frames
$ NUMDUMPS = 10
# Whether to use diffusive (but robust) fluxes
$ DIFFUSIVE = False
# Order in time
$ TIME_ORDER = "second"
# Write data for restarting the simulation
$ WRITE_RESTART = False
# Output info for debugging purposes
$ DEBUG = False
# Default dimensionality
$ NDIM = 2

$XMIN = -0.5*TRANS_LENGTH
$XMAX =  0.5*TRANS_LENGTH

$YMIN = -0.5*PAR_LENGTH
$YMAX =  0.5*PAR_LENGTH

$ZMIN = -0.5*TRANS_LENGTH
$ZMAX =  0.5*TRANS_LENGTH

$if NDIM==2
$ CFL = 0.4
$ numCells = [TRANS_ZONES, PAR_ZONES]
$ periodicDirections = [0]
$ lowerBounds = [XMIN,  YMIN]
$ upperBounds = [XMAX,  YMAX]
$ else
$ CFL = 0.32
$ numCells = [TRANS_ZONES, PAR_ZONES, TRANS_ZONES]
$ periodicDirections = [0 2]
$ lowerBounds = [XMIN,  YMIN,  ZMIN]
$ upperBounds = [XMAX,  YMAX,  ZMAX]
$ endif

# Initialize a USim simulation
initializeFluidSimulation(NDIM,0.0,TEND,NUMDUMPS,CFL,GAS_GAMMA,WRITE_RESTART,DEBUG)

# Setup the grid
addGrid(lowerBounds, upperBounds, numCells, periodicDirections)

# Create data structures needed for the simulation
createFluidSimulation()

# Step 1: Add Variables
addVariable(gas_gamma,GAS_GAMMA)
```

```
addVariable(rhoTop,RHO_HEAVY)
addVariable(rhoBottom,RHO_LIGHT)
addVariable(perturb,PERTURB_AMP)
addVariable(ytop,YMAX)
addVariable(gravity,GRAVITY_ACCEL)
addVariable(lx,TRANS_LENGTH)
addVariable(ly,PAR_LENGTH)

# Step 2: Add Pre-Expressions
addPreExpression(p0 = 0.01)
addPreExpression(pert = 0.01)
addPreExpression(pi = 3.14159)
addPreExpression(rho = if (y < 0.0, rhoBottom, rhoTop))
addPreExpression(pr = (1.0/gas_gamma) - (gravity*rho*y))
addPreExpression(vx = 0.0)
addPreExpression(vy = (0.25*perturb)*(1.0 + cos(2.0*pi*x/lx))*(1.0 + cos(2.0*pi*y/ly)))
addPreExpression(vz = 0.0)

# Step 3: Add expressions specifying initial condition on density,
# momentum, total energy
addExpression(rho)
addExpression(rho*vx)
addExpression(rho*vy)
addExpression(rho*vz)
addExpression((pr/(gas_gamma-1))+0.5*rho*(vx*vx+vy*vy+vz*vz))

# Add the spatial discretization of the fluxes
finiteVolumeScheme(DIFFUSIVE)

# Add source term for gravitational acceleration to the equations
addGravitationalAcceleration(GRAVITY_ACCEL)

# Boundary conditions
boundaryCondition(wall,top)
boundaryCondition(wall,bottom)
boundaryCondition(periodic)

# Time integration
timeAdvance(TIME_ORDER)

# Run the simulation!
runFluidSimulation()
```

Note: For more depth, you can view the actual input blocks to Ulixes in the *Setup* window by choosing *Save And Process Setup* and then clicking on the *shockTube.in* file. In the *.in* file all macros are expanded to produce input blocks.

3.3.6 Solving the MHD Equations in Multi-Dimensions

We can extend the approach above to solving the MHD equations in multi-dimensions in a straightforward fashion. The steps for *Creating a Fluid Simulation* is unchanged from *Using USim to solve the Magnetohydrodynamic Equations*, while the steps for *Adding a Simulation Grid* are identical to that given above. The differences come in *Creating a Fluid Simulation*, where Steps 1) - 3) of specifying the initial condition change:

Step 1: Add Variables:

```
addVariable(gas_gamma,GAS_GAMMA)
addVariable(rhoTop,RHO_HEAVY)
addVariable(rhoBottom,RHO_LIGHT)
addVariable(perturb,PERTURB_AMP)
addVariable(ytop,YMAX)
addVariable(gravity,GRAVITY_ACCEL)
addVariable(lx,TRANS_LENGTH)
addVariable(ly,PAR_LENGTH)
addVariable(mu0,MU0)
addVariable(beta,BETA)
```

Step 2: Add Pre Expression's:

```
addPreExpression(p0 = 0.01)
addPreExpression(pert = 0.01)
addPreExpression(pi = 3.14159)
addPreExpression(rho = if (y < 0.0, rhoBottom, rhoTop))
addPreExpression(pr = (1.0/gas_gamma) - (gravity*rho*y))
addPreExpression(vx = 0.0)
addPreExpression(vy = (0.25*perturb)*(1.0 + cos(2.0*pi*x/lx))*(1.0 + cos(2.0*pi*y/ly)))
addPreExpression(vz = 0.0)
addPreExpression(bx = sqrt(2.0*pr/(beta*beta)))
addPreExpression(by = 0.0)
addPreExpression(bz = 0.0)
addPreExpression(psi = 0.0)
```

Step 3: Add Expressions for density, momentum, total energy, magnetic field and correction potential:

```
addExpression(rho)
addExpression(rho*vx)
addExpression(rho*vy)
addExpression(rho*vz)
addExpression(pr/(gas_gamma-1.0) + 0.5*rho*(vy*vy)+0.5*((bx*bx)/mu0))
addExpression(bx)
addExpression(by)
addExpression(bz)
addExpression(psi)
```

The remaining steps in setting up the simulation are unchanged. We can then extend the underlying pattern for USimBase simulations:

```
# Are we solving the MHD equations?
$ MHD = True

# Import macros to setup simulation
$ import fluidsBase.mac
# if MHD
$ import idealmhd.mac
$ else
$ import euler.mac
$ endif

# Specify parameters for the specific physics problem
$ PARAM_1 = <value>
$ PARAM_2 = <value>
$ PARAM_N = <value>

# Initialize a USim simulation
$ if MHD
```

```
initializeFluidSimulation(NDIM,0.0,END_TIME,NUMDUMPS,CFL,GAS_GAMMA,MU0,WRITE_RESTART,DEBUG)
$ else
initializeFluidSimulation(NDIM,0.0,END_TIME,NUMDUMPS,CFL,GAS_GAMMA,WRITE_RESTART,DEBUG)
$ endif

# Setup the grid
addGrid(lowerBounds, upperBounds, numCells, periodicDirections)

# Create data structures needed for the simulation
createFluidSimulation()

# Specify initial condition
# Step 1: Add Variables
addVariable(NAME,<value>)

# Step 2: Add Pre-Expressions
addPreExpression(<PreExpression>)

# Step 3: a) Add expressions specifying initial condition on density,
# momentum
addExpression(<expression>)
$ if MHD
# Step 3: b) Add expression specifying initial conditions on total
# energy, magnetic field, correction potential
addExpression(<expression>)
$ else
# Step 3: b) Add expression specifying initial conditions on total
# energy
addExpression(<expression>)
$ endif

# Add the spatial discretization of the fluxes
finiteVolumeScheme(DIFFUSIVE)

# Add (optional) physics to the finite volume scheme
< physics macros >

# Boundary conditions
boundaryCondition(<boundaryCondition,entity>)

# Time integration
timeAdvance(TIME_ORDER)

# Run the simulation!
runFluidSimulation()
```

Note: The algorithms in USim automatically preserve $\nabla \cdot B = 0$ (the solenoidal constraint on the magnetic field), so there is no need for the user to make changes to the algorithm at the input file level when running in multi-dimensions.

3.3.7 An Example Simulation

The input file for the problem *Rayleigh-Taylor Instability* in the USimBase package demonstrates each of the concepts described above to evolve the Rayleigh-Taylor instability for a single mode perturbation. An additional example of these concepts can be found in the *Kelvin-Helmholtz Instability* input file in the USimBase package.

Executing the *Rayleigh-Taylor* input file within USimComposer and switching to the *Visualize* tab yields the plots

shown in Fig. 3.3.

Note that it is possible to execute both the *Rayleigh-Taylor Instability* and *Kelvin-Helmholtz Instability* examples in the USimBase package using MHD by selecting *MHD = True*.

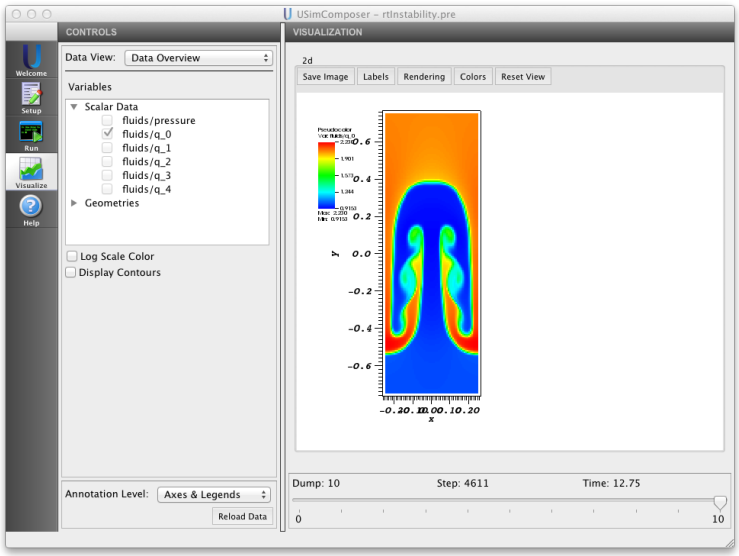


Fig. 3.3: Visualization tab in USimComposer after executing the input file for the this tutorial.

3.4 Solving Problems on Advanced Structured Meshes in USim

In *Using USim to solve the Euler Equations* we discussed the basic methods used by USim to solve the Euler equations. Next, in *Using USim to solve the Magnetohydrodynamic Equations*, we extended these ideas to solve the MHD equations in one-dimension and showed how USimBase simulations for both the Euler and MHD equations follow the same basic pattern. Then, in *Solving Multi-Dimensional Problems in USim*, we built on these concepts to demonstrate how to use USim to solve the Euler and MHD equations in multi-dimensions, how to utilize more advanced boundary conditions and how to apply external forces (such as gravity) to the equations. In this tutorial, we extend these ideas to demonstrate how USim can solve problems in axisymmetric curvilinear coordinates and how to use two- and three-dimensional body fitted meshes to solve problems around simple geometries.

Contents

3.4.1 Solving Problems in Axi-Symmetric Curvilinear Coordinates

An example of using USim to solve the MHD equations in axi-symmetric curvilinear coordinates is found in quickstart-zpinch.

Adding a Simulation Grid

The use of axisymmetric cylindrical coordinates in this problem is specified through the use of the grid:

```
addCylindricalGrid(lowerBounds, upperBounds, numCells, periodicDirections)
```

This macro is documented at grid-macro and follows the same pattern that we have seen previously for logically structured Cartesian grids:

`lowerBounds`: Vector of coordinates for lower edge of grid, lowerBounds = [RMIN ZMIN PHIMIN]

`upperBounds`: Vector of coordinates for upper edge of grid, upperBounds = [RMAX ZMAX PHIMAX]

`numCells`: Vector of number of cells in grid, numCells = [NR NZ NPHI]

`periodicDirections`: List of directions that are periodic

```
periodicDirections = [ 0 ] (R-direction periodic)
periodicDirections = [ 0 1 ] (R,Z-directions periodic)
periodicDirections = [ 0 1 2 ] (R,Z,PHI-directions periodic)
```

Using the *addCylindricalGrid* macro automatically tells the rest of the USim solvers to use the cylindrical form of the operator. If a cylindrical form of an operator is not available, USim will stop with an error message.

Note: Cylindrical grids in USim are designed with **axisymmetric** coordinates in mind, so a one-dimensional mesh simulates the R coordinate, a two-dimensional mesh simulates the (R, Z) coordinates and a three-dimensional mesh simulates the (R, Z, ϕ) coordinates.

Creating a Fluid Simulation

The initial conditions for the quickstart-zpinch in axisymmetric cylindrical coordinates are given by:

Step 1:

```
addVariable(gas_gamma,GAS_GAMMA)
addVariable(mu0,MU0)
addVariable(invSqrtMu0,INV_SQRT_MU0)
addVariable(Rp,RP)
addVariable(n_0,N0)
addVariable(j_0,J0)
addVariable(p_0,P0)
addVariable(mi,MI)
addVariable(k,WAVENUMBER)
addVariable(alpha,BASE_PRESSURE_RATIO)
addVariable(perturb,PERTURBATION_AMPLITUDE)
```

Step2:

```
addPreExpression(r = x)
addPreExpression(Z = y)
addPreExpression(phi = z)
addPreExpression(rho = if(r<Rp, mi*n_0*(alpha+(1.0-(r*r)/(Rp*Rp))), mi*n_0*alpha))
addPreExpression(vr = 0.0)
addPreExpression(vphi = 0.0)
addPreExpression(vz = 0.0)
addPreExpression(pr = if(r < Rp, p_0-0.25*mu0*j_0*j_0*r*r, alpha*0.25*mu0*j_0*j_0*Rp*Rp))
addPreExpression(br = 0.0)
addPreExpression(bphi = if(r < Rp, -0.5*r*mu0*j_0*(1.0+perturb*sin(k*Z)), -0.5*(Rp*Rp/r)*mu0*j_0*(1.0
addPreExpression(bz = 0.0)
addPreExpression(psi = 0.0)
```

Step 3:

```
addExpression(rho)
addExpression(rho*vr)
addExpression(rho*vphi)
addExpression(rho*vz)
addExpression(pr/(gas_gamma-1.0) + 0.5*rho*(vr*vr+vphi*vphi+vz*vz)+0.5*((br*br+bphi*bphi+bz*bz)/mu0))
addExpression(br*invSqrtMu0)
addExpression(bphi*invSqrtMu0)
addExpression(bz*invSqrtMu0)
addExpression(psi)
```

Note: While the ordering of coordinates on cylindrical grids in USim is (R, Z, ϕ), the components of vectors (such as the momentum and magnetic field defined in step 3 above) take the order $(\hat{\mathbf{R}}, \hat{\phi}, \hat{\mathbf{Z}})$. Many issues in simulations involving cylindrical coordinates originate in an incorrect ordering of vector components.

Putting it all Together

We can then extend the underlying pattern for USimBase simulations to incorporate curvilinear coordinates:

```
# Are we solving the MHD equations?
$ MHD = True

# Are we using cylindrical coordinates?
$ CYLINDRICAL = True

# Import macros to setup simulation
$ import fluidsBase.mac
```

```
# if MHD
$ import idealmhd.mac
$ else
$ import euler.mac
$ endif

# Specify parameters for the specific physics problem
$ PARAM_1 = <value>
$ PARAM_2 = <value>
$ PARAM_N = <value>

# Initialize a USim simulation
$ if MHD
initializeFluidSimulation(NDIM,0.0,END_TIME,NUMDUMPS,CFL,GAS_GAMMA,MU0,WRITE_RESTART,DEBUG)
$ else
initializeFluidSimulation(NDIM,0.0,END_TIME,NUMDUMPS,CFL,GAS_GAMMA,WRITE_RESTART,DEBUG)
$ endif

# Setup the grid
$ if CYLINDRICAL
addCylindricalGrid(lowerBounds, upperBounds, numCells, periodicDirections)
$ else
addGrid(lowerBounds, upperBounds, numCells, periodicDirections)
$ endif

# Create data structures needed for the simulation
createFluidSimulation()

# Specify initial condition
# Step 1: Add Variables
addVariable(NAME,<value>)

# Step 2: Add Pre-Expressions
addPreExpression(<PreExpression>)

# Step 3: a) Add expressions specifying initial condition on density,
# momentum
addExpression(<expression>)
$ if MHD
# Step 3: b) Add expression specifying initial conditions on total
# energy, magnetic field, correction potential
addExpression(<expression>)
$ else
# Step 3: b) Add expression specifying initial conditions on total
# energy
addExpression(<expression>)
$ endif

# Add the spatial discretization of the fluxes
finiteVolumeScheme(DIFFUSIVE)

# Boundary conditions
boundaryCondition(<boundaryCondition,entity>)

# Time integration
timeAdvance(TIME_ORDER)

# Run the simulation!
```

```
runFluidSimulation()
```

An Example Simulation

The input file for quickstart-zpinch in the USimBase package demonstrates each of the concepts described above to evolve the z-Pinch problem in two-dimensional magnetohydrodynamics. Executing this input file within USimComposer and switching to the *Visualize* tab yields the plots shown in Fig. 3.4.

Note: For more depth, you can view the actual input blocks to Ulixes in the *Setup* window by choosing *Save And Process Setup* and then clicking on the *zPinch.in* file. In the *.in* file all macros are expanded to produce input blocks.

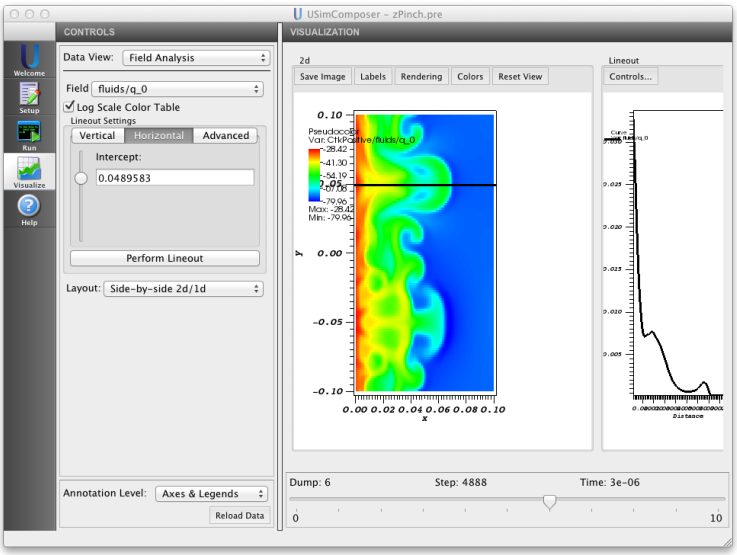


Fig. 3.4: Visualization tab in USimComposer after executing the *Unstable plasma z-Pinch* input file for the tutorial.

3.4.2 Solving Problems on Two-Dimensional Body-Fitted Meshes in USim

This tutorial is based on the quickstart-rampflow example in USimBase, which we use to demonstrate the creation of two-dimensional body fitted meshes.

Adding a Simulation Grid

The *Ramp Flow* simulation uses a refmanual-bodyFitted grid, which is added to the simulation through the *addBodyFittedGrid* macro:

```
addBodyFittedGrid(lowerBounds, upperBounds, numCells, periodicDirections)
```

The options for this macro are documented in grid-macro. For completeness, we include them here:

`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)
```

Now that the body-fitted grid has been created, it is possible to transform the coordinates of the grid into the desired shaped. This is a three-step process, much like specifying an initial condition:

1. Use *addGridVariable* to add variables that are independent of grid position.
2. Use *addGridPreExpression* to add quantities that are functions of grid position, variables and any previously defined PreExpression in this block. Evaluated before expressions and the result is not accessible outside of this block. Any number of PreExpressions can be added.
3. Use *addGridExpression* to define the new coordinates in the grid. The order of the expressions correspond to the order of the coordinates in the grid and there must be the same number of expressions as dimensions in the grid.

For the *RampFlow* example, the *GridVariable*s added in Step 1 in the above process are:

```
# Grid properties
addGridVariable(xmin,XIN)
addGridVariable(xmax,XUP)
addGridVariable(ymax,YUP)
addGridVariable(slope,$math.tan(math.pi*THETA/180)$)

# Numbers of cells
addGridVariable(inletCells,1.0*NXI)
addGridVariable(rampCells,1.0*NXR)
addGridVariable(yCells,$1.0*NY$)

# x direction cell spacing in computational space
addGridVariable(dxc,$1.0/(NXI+NXR)$)
# y direction cell spacing in computational space
addGridVariable(dyc,$1.0/NY$)
# z direction cell spacing in computational space
```

Next, the *GridPreExpression*s added in Step 2 in the above process are:

```
# Grid preExpressions
addGridPreExpression(ix=rint(x/dxc))
addGridPreExpression(iy=rint(y/dyc))
addGridPreExpression(dxi=xmin/inletCells)
addGridPreExpression(dxr=(xmax-xmin)/rampCells)
addGridPreExpression(xp=if(inletCells>=ix,dxi*ix,xmin+dxr*(ix-inletCells)))
addGridPreExpression(dyi = ymax/yCells)
addGridPreExpression(b = -slope*xmin)
addGridPreExpression(y0 = slope*xp + b)
addGridPreExpression(dyr = (ymax-y0)/yCells)
addGridPreExpression(yp=if(inletCells>=ix,dyi*iy,y0+dyr*iy))
```

Finally, the *GridExpression*s added in Step 3 in the above process are:

```
addGridExpression(xp)
addGridExpression(yp)
```

The *bodyFitted* Grid in USim is an extension of the *cart* grid. The *cart* Grid lays out the grid in a uniform manner and provides an x-y coordinate for each of the vertices of the cells. The *bodyFitted* grid allows the user to modify the locations of these x-y vertices. The *lowerBounds upperBounds* and *numCells* parameters set up the uniform cartesian

grid and its vertices. The vertices of this cartesian grid are then modified according to the *GridExpressions* which defines the mapping from the cartesian grid to the *bodyFitted* grid, therefore if we specify:

```
addGridExpression(x*x)
addGridExpression(y*y)
```

then the cartesian grid is mapped to a grid with where x and y vary quadratically. USim loops through every vertex and replaces the cartesian vertex with the new vertex specified by each of the *addGridExpression* macros. The user must be careful to make sure that the new vertices are a result of simple stretching the cartesian grid so that cells in the body fitted grid do not overlap.

Note: In the body fitted grid, ghost cells are also mapped using the *addGridExpression* macros. Ghost cells add an extra layer to the grid beyond what is specified by *cells*. The location of these additional vertices can be computed by adding additional cells to the cartesian grid. In many cases it is also important that the mapping of these ghost cells do not produce overlapping cells which can result in negative areas and misdirected tangents and normals.

An Example Simulation

The input file for the problem *Ramp Flow* in the USimBase package demonstrates each of the concepts described above. Executing this input file within USimComposer and switching to the *Visualize* tab yields the plots shown in Fig. 3.5.

Note: For more depth, you can view the actual input blocks to Ulixes in the *Setup* window by choosing *Save And Process Setup* and then clicking on the *rampFlow.in* file. In the *.in* file all macros are expanded to produce input blocks.

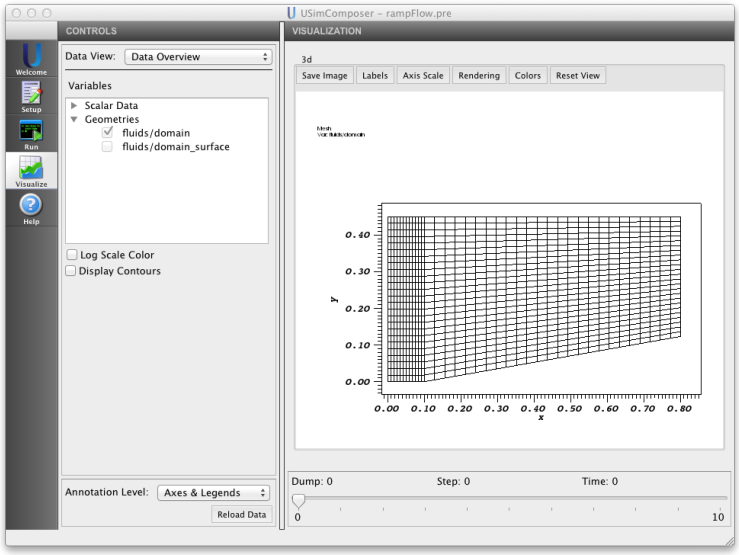


Fig. 3.5: Visualization of mesh geometry in USimComposer after executing the *MHD Ramp Flow* input file for the tutorial.

3.4.3 Solving Problems on Three-Dimensional Body-Fitted Meshes in USim

The input file for quickstart-zpinch in the USimBase package demonstrates each of the concepts described above to evolve the z-Pinch problem in three-dimensional magnetohydrodynamics

Adding a Simulation Grid

The three-dimensional version of the *zPinch* simulation uses a refmanual-bodyFitted grid, which is added to the simulation through the *addBodyFittedGrid* macro:

```
addBodyFittedGrid(lowerBounds, upperBounds, numCells, periodicDirections)
```

The options for this macro are documented in grid-macro. For completeness, we include them here:

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)
```

In this example, we use the body-fitted grid to transform a three-dimensional cylindrical mesh into a three-dimensional (X, Y, Z) mesh that utilizes a cylindrical distribution. To accomplish this, we only have to perform the third step of the three-step process above:

```
addGridExpression(x*cos(y))
addGridExpression(x*sin(y))
addGridExpression(z)
```

Creating a Fluid Simulation

In order to specify the initial condition on a three-dimensional Cartesian mesh when we start from a cylindrical initial condition, we need to transform components of vectors from cylindrical to cartesian coordinates. We can do this during the specification of the initial condition:

Step 1

```
addVariable(gas_gamma,GAS_GAMMA)
addVariable(mu0,MU0)
addVariable(invSqrtMu0,INV_SQRT_MU0)
addVariable(Rp,RP)
addVariable(n_0,N0)
addVariable(j_0,J0)
addVariable(p_0,P0)
addVariable(mi,MI)
addVariable(k,WAVENUMBER)
addVariable(alpha,BASE_PRESSURE_RATIO)
addVariable(perturb,PERTURBATION_AMPLITUDE)
```

Step 2

```
# Compute cylindrical coordinatees based on our x,y,z coordinates in
# the grid
addPreExpression(r = sqrt(x^2+y^2))
addPreExpression(phi = atan2(y,x))
addPreExpression(Z = z)

# Setup our plasma parameters in cylindrical coordinates
addPreExpression(rho = if(r<Rp, mi*n_0*(alpha+(1.0-(r*r)/(Rp*Rp))), mi*n_0*alpha))
```

```
addPreExpression(vr = 0.0)
addPreExpression(vphi = 0.0)
addPreExpression(vz = 0.0)
addPreExpression(pr = if(r < Rp, p_0-0.25*mu0*j_0*j_0*r*r, alpha*0.25*mu0*j_0*j_0*Rp*Rp))
addPreExpression(br = 0.0)
addPreExpression(bphi = if(r < Rp, -0.5*r*mu0*j_0*(1.0+perturb*sin(k*Z)), -0.5*(Rp*Rp/r)*mu0*j_0*(1.0
addPreExpression(bz = 0.0)
addPreExpression(psi = 0.0)

# Transform from cylindrical coordinates into Cartesian.
addPreExpression(vx = vr*cos(phi)+vphi*sin(phi))
addPreExpression(vy = vr*sin(phi)-vphi*cos(phi))
addPreExpression(bx = br*cos(phi)+bphi*sin(phi))
addPreExpression(by = br*sin(phi)-bphi*cos(phi))
```

Step 3

```
# Specify initial condition according to cartesian vector components
addExpression(rho)
addExpression(rho*vx)
addExpression(rho*vy)
addExpression(rho*vz)
addExpression(pr/(gas_gamma-1.0) + 0.5*rho*(vx*vx+vy*vy+vz*vz)+0.5*((bx*bx+by*by+bz*bz)/mu0))
addExpression(bx*invSqrtMu0)
addExpression(by*invSqrtMu0)
addExpression(bz*invSqrtMu0)
addExpression(psi)
```

An Example Simulation

The input file for quickstart-zpinch in the USimBase package with *NDIM=3* demonstrates each of the concepts described above to evolve the z-Pinch problem in three-dimensional magnetohydrodynamics.

Note: For more depth, you can view the actual input blocks to Ulixes in the *Setup* window by choosing *Save And Process Setup* and then clicking on the *zPinch.in* file. In the *.in* file all macros are expanded to produce input blocks.

3.5 Solving Problems on Unstructured Meshes in USim

In *Using USim to solve the Euler Equations* we discussed the basic methods used by USim to solve the Euler equations. Next, in *Using USim to solve the Magnetohydrodynamic Equations*, we extended these ideas to solve the MHD equations in one-dimension and showed how USimBase simulations for both the Euler and MHD equations follow the same basic pattern. Then, in *Solving Multi-Dimensional Problems in USim*, we built on these concepts to demonstrate how to use USim to solve the Euler and MHD equations in multi-dimensions, how to utilize more advanced boundary conditions and how to apply external fources (such as gravity) to the equations. Following on from this in *Solving Problems on Advanced Structured Meshes in USim*, we demonstrated how USim can solve problems in axisymmetric curvilinear coordinates and how to use two- and three-dimensional body fitted meshes to solve problems around simple geometries. In this tutorial, we introduce unstructured meshes in USim, show how to create boundary conditions for simple geometries, apply custom boundary conditions based on user specified parameters, and compute flow diagnostics.

The quickstart-forwardfacingstep example in USimBase illustrates how to solve problems on unstructured meshes.

Note: In USim Version 3.0 unstructured meshes should be quadrilateral or hexahedral. If unstructured grids are

used in parallel then the user needs to define the partitioning of the grid prior to import; USim does not partition unstructured meshes itself.

Contents

3.5.1 Creating Unstructured Meshes in CUBIT for USim

USim currently accepts Exodus II files created in a number of programs including CUBIT and Trelis. Exodus II files generally have the extension .exo, however we frequently use the genesis format (.g) which contains only the mesh data of the Exodus II file.

USim 3.0 currently accepts only quad or hex elements.

- First, create a mesh in CUBIT following the CUBIT tutorials.
- After the mesh has been created, sidesets (used for boundaries) can also be created and then used in defining the locations of boundary conditions.
  - In CUBIT, select 'Materials and Bcs' then select 'Entities - Blocks'. From the drop down menu select 'Add' and set the 'Block ID' to 1. If the mesh is 2d pick 'Surface' and hit 'Apply'. If the mesh is 3D, instead select 'Volume' and hit 'Apply'.
  - Next, in 'Entities - Blocks' switch the drop down menu to 'Element Type'. If the mesh is 2D select 'Surfaces' and then 'Quad4', if the mesh is 3D select 'Volumes' and 'Hex8', hit 'Apply'.
  - Next switch from 'Entity - Blocks' to 'Entity - Sidesets'. In 2D select 'Curve' then click on the mesh or the boundary you wish to choose for your sideset. In 3D select 'Surface' then select the boundary you are interested in. You will now have a boundary defined as *sideset 1* (for example).
- Once the grid is read into USim an *entity* is defined over the side with the name *sideSetHalosId1*. This entity can then be used USim boundary conditions.
- Add any other side sets you wish to use as boundaries at this point. The mesh is now ready to export.
- In the main CUBIT window select 'File->Export'. Choose the directory to export the file and from the dropdown menu select 'Genesis' to export a Genesis file. Type in the file name and select 'Save'. You should now have a grid that can be read into USim.

3.5.2 Creating Unstructured Meshes in Trelis for USim

USim currently accepts Exodus II files created in a number of programs including CUBIT and Trelis. Exodus II files generally have the extension .exo, however we frequently use the genesis format (.g) which contains only the mesh data of the Exodus II file.

USim 3.0 currently accepts only quad or hex elements.

- First, create a mesh in Trelis using tutorials and documentation.
- After the mesh has been created, sidesets (used for boundaries) can also be created and then used in defining the locations of boundary conditions.
  - In Trelis, select the 'Analysis Groups and Materials' mode, then select 'Entity-Sidesets'.
  - Set the 'Action' to 'Create sideset' and set the 'Sideset ID' to 1
  - Select 'Curve' if the mesh is 2D and 'Surface' if the mesh is 3D and type in the curve/surface ID or select it with your mouse and hit 'Apply'
- Once the grid is read into USim an *entity* is defined over the side with the name *sideSetHalosId1*. This entity can then be used USim boundary conditions.
- Add any other side sets you wish to use as boundaries at this point. The mesh is now ready to export.
- In the main Trelis window select 'File->Export'. Choose the directory to export the file and from the dropdown menu select 'Exodus' to export an Exodus file. Type in the file name and select 'Save'. You should now have a grid that can be read into USim.

3.5.3 Creating Unstructured Meshes in Gmsh for USim

USim also accepts Gmsh meshes with extension .msh.

- First, create a geometry in Gmsh following any Gmsh tutorials.
- Gmsh will mesh using triangles/tets unless instructed otherwise. Open the 'Tools -> Options' menu, and under the 'General' tab for 'Mesh', check the box "Recombine all triangular meshes". Also set the subdivision algorithm to "All Quads" or "All Hexes" for 2D or 3D meshes.
- Save the mesh. You should now have a grid that can be read into USim.

Gmsh does not have a facility for setting sideSets used for USim boundary conditions. Instead, an *entity* can be created in USim by masking off ghost cells to isolate those you would like to create a boundary condition on.

An example of masking off a region of the ghost cells is as follows:

```
<Updater generateOpenBc>
  kind = entityGenerator2d
  onGrid = domain
  newEntityName = openBc
  onEntity = ghost
  <Function mask>
    kind = exprFunc
    exprs = ["if(x>0 && y>0.03,1.0,-1.0)"]
  </Function>
</Updater>
```

This block creates a new entity of name "openBC" which can be called in an Updater and used for setting a boundary condition.

3.5.4 Running Unstructured Meshes in Parallel

CUBIT/Trelis

If you wish to run your simulation in parallel, the mesh will need to be decomposed prior to use in USim. CUBIT and Trelis do not have a facility to do this, so an external program must be used. We recommend using SEACAS, which is a suite of applications for supporting finite element analysis software using Exodus file format. SEACAS Specifically, nem_slice and nem_spread.

In the Linux distribution of USim 3.0, an Exodus II mesh file can be decomposed using the provided partitioner script, decomp.

For example, if you plan to use 8 processors, from the command line run:

```
<ULIXES_BIN_DIR>/decomp -p 8 --root ./ meshfile.g
```

or

```
<ULIXES_BIN_DIR>/decomp -p 8 --root ./ meshfile.exo
```

For more information about the decomp script, run the script with the help option:

```
<ULIXES_BIN_DIR>/decomp -h
```

A number of files will be output corresponding to the number of processors you plan to run on. In the case above, 8 files, namely:

meshfile.g.8.0 meshfile.g.8.1 meshfile.g.8.2 meshfile.g.8.3 meshfile.g.8.4 meshfile.g.8.5 meshfile.g.8.6 meshfile.g.8.7

USim will automatically detect these files upon running with 8 cores. In your <Creator> block, you simply need to set the file to 'meshfile.g':

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

Gmsh

Gmsh, on the other hand, can partition your grid prior to saving. In Gmsh, select 'Mesh->Partition' and set the 'Number of Partitions' to however many cores you would like to run on. Then hit 'Partition'. The colors in your mesh should change to show the decomposition of the mesh onto your chosen number of partitions.

Saving this mesh will result in 1 file being written. We recommend you save the file with the number of partitions in the file name.:

```
meshfile16.msh
```

USim will NOT automatically detect which file to use when running in parallel. In your <Creator> block, be sure to set the correct meshfile.:

```
<Creator ctor>
  kind = gmsh
  ndim = 3
  file = meshfile16.msh
</Creator>
```

3.5.5 Initializing a Simulation

One of the great strengths of USim is that the underlying algorithms are able to work on both structured and unstructured meshes. This means that the worflow to setup a simulation on a unstructured mesh is very similar to that seen previously:

```
# Are we solving the MHD equations?
$ MHD = False

# Import macros to setup simulation
$ import fluidsBase.mac
$ if MHD
$ import idealmhd.mac
$ else
$ import euler.mac
$ endif

# Specify parameters for the specific physics problem
$ PARAM_1 = <value>
$ PARAM_2 = <value>
$ PARAM_N = <value>

# Initialize a USim simulation
$ if MHD
initializeFluidSimulation(NDIM,0.0,TEND,NUMDUMPS,CFL,GAS_GAMMA,MU0,WRITE_RESTART,DEBUG)
$ else
initializeFluidSimulation(NDIM,0.0,TEND,NUMDUMPS,CFL,GAS_GAMMA,WRITE_RESTART,DEBUG)
$ endif
```

For the *Forward Facing Step* example with the Euler equations, the parameters that we specify for the physics problem are as follows:

```
# grid for simulation
$ GRIDFILE = "forwardFacingStep"
# format of grid
$ GRIDTYPE = "gmsh"
# adiabatic index
$ GAS_GAMMA = 1.4
# Magnetic field strength
$ BETA = 1.0e3
# end time for simulation
$ TEND = 4.0
# number of frames
$ NUMDUMPS = 20
# Riemann solver
$ DIFFUSIVE = True
# Order in time
$ TIME_ORDER = "second"
# Write data for restarting the simulation
$ WRITE_RESTART = False
# Output info for debugging purposes
$ DEBUG = False
#  Dimensionality
$ NDIM = 2
# Permeability of free space
$ MU0 = 1.0
# CFL condition
$ CFL = 0.4
```

3.5.6 Adding a Simulation Grid

The use of an unstructured mesh in USim is specified through the use of one of the two grids:

```
addExodusGrid(GRIDFILE)
```

or:

```
addGmshGrid(GRIDFILE)
```

The choice of which block to use corresponds to the format of the mesh; either GMSH or ExodusII format. The *GRIDFILE* is the name of the file containing the mesh **without the file extension**.

Our simulation input file now looks like:

```
# Are we solving the MHD equations?
$ MHD = False

# Import macros to setup simulation
$ import fluidsBase.mac
$ if MHD
$ import idealmhd.mac
$ else
$ import euler.mac
$ endif

# Specify parameters for the specific physics problem
# grid for simulation
$ GRIDFILE = "forwardFacingStep"
# format of grid
$ GRIDTYPE = "gmsh"
# adiabatic index
$ GAS_GAMMA = 1.4
# Magnetic field strength
$ BETA = 1.0e3
# end time for simulation
$ TEND = 4.0
# number of frames
$ NUMDUMPS = 20
# Riemann solver
$ DIFFUSIVE = True
# Order in time
$ TIME_ORDER = "second"
# Write data for restarting the simulation
$ WRITE_RESTART = False
# Output info for debugging purposes
$ DEBUG = False
#  Dimensionality
$ NDIM = 2
# Permeability of free space
$ MU0 = 1.0
# CFL condition
$ CFL = 0.4

# Initialize a USim simulation
$ if MHD
initializeFluidSimulation(NDIM,0.0,TEND,NUMDUMPS,CFL,GAS_GAMMA,MU0,WRITE_RESTART,DEBUG)
$ else
initializeFluidSimulation(NDIM,0.0,TEND,NUMDUMPS,CFL,GAS_GAMMA,WRITE_RESTART,DEBUG)
$ endif
```

```
$ if isEqualString(GRIDTYPE,ExodusII)
addExodusGrid(GRIDFILE)
$ else
addGmshGrid(GRIDFILE)
$ endif
```

3.5.7 Creating a Fluid Simulation

As in the *Initializing a Simulation* step, the procedure for setting up a fluid simulation on an unstructured mesh is **identical** to that on a structured mesh. For the *Flow Over a Forward Facing Step* example, this proceeds as follows. First, we create the variables needed to simulate the fluid:

```
createFluidSimulation()
```

We then proceed through the three-step process to specify the distribution of the fluid on the grid. Step 1: Add Variables:

```
addVariable(gasGamma,GAS_GAMMA)
```

Step 2: Add Pre Expression's:

```
addPreExpression(rho = gasGamma)
addPreExpression(vx = 3.0)
addPreExpression(vy = 0.0)
addPreExpression(vz = 0.0)
addPreExpression(pr = 1.0)
```

Step 3: Add Expressions for density, momentum and total energy:

```
addExpression(rho)
addExpression(rho*vx)
addExpression(rho*vy)
addExpression(rho*vz)
addExpression((pr/(gas_gamma-1.0))+0.5*rho*(vx*vx+vy*vy+vz*vz))
```

Note that Step 3 is **identical** to that used in *Using USim to solve the Euler Equations*. Our simulation input file now looks like:

```
# Are we solving the MHD equations?
$ MHD = False

# Import macros to setup simulation
$ import fluidsBase.mac
$ if MHD
$ import idealmhd.mac
$ else
$ import euler.mac
$ endif

# Specify parameters for the specific physics problem
# grid for simulation
$ GRIDFILE = "forwardFacingStep"
# format of grid
$ GRIDTYPE = "gmsh"
# adiabatic index
$ GAS_GAMMA = 1.4
# Magnetic field strength
```

```
$ BETA = 1.0e3
# end time for simulation
$ TEND = 4.0
# number of frames
$ NUMDUMPS = 20
# Riemann solver
$ DIFFUSIVE = True
# Order in time
$ TIME_ORDER = "second"
# Write data for restarting the simulation
$ WRITE_RESTART = False
# Output info for debugging purposes
$ DEBUG = False
#  Dimensionality
$ NDIM = 2
# Permeability of free space
$ MU0 = 1.0
# CFL condition
$ CFL = 0.4

# Initialize a USim simulation
$ if MHD
initializeFluidSimulation(NDIM,0.0,TEND,NUMDUMPS,CFL,GAS_GAMMA,MU0,WRITE_RESTART,DEBUG)
$ else
initializeFluidSimulation(NDIM,0.0,TEND,NUMDUMPS,CFL,GAS_GAMMA,WRITE_RESTART,DEBUG)
$ endif

$ if isEqualString(GRIDTYPE,ExodusII)
addExodusGrid(GRIDFILE)
$ else
addGmshGrid(GRIDFILE)
$ endif

# Create data structures needed for the simulation
createFluidSimulation()

# Step 1: Add Variables
addVariable(gasGamma,GAS_GAMMA)

# Step 2: Add Pre-Expressions
addPreExpression(rho = gasGamma)
addPreExpression(vx = 3.0)
addPreExpression(vy = 0.0)
addPreExpression(vz = 0.0)
addPreExpression(pr = 1.0)

# Step 3: Add expressions specifying initial condition on density,
# momentum, total energy
addExpression(rho)
addExpression(rho*vx)
addExpression(rho*vy)
addExpression(rho*vz)
addExpression((pr/(gas_gamma-1))+0.5*rho*(vx*vx+vy*vy+vz*vz))
```

3.5.8 Evolving the Fluid

For multi-dimensional physics problems on structured meshes, our general pattern for evolving the fluid took the form:

```
# Add the spatial discretization of the fluxes
finiteVolumeScheme(DIFFUSIVE)

# Add (optional) physics to the finite volume scheme
< physics macros >

# Boundary conditions
boundaryCondition(<boundaryCondition,entity>)

# Time integration
timeAdvance(TIME_ORDER)
```

3.5.9 Boundary Conditions on Unstructured Meshes

The complexity of performing calculations on an unstructured mesh in USim is associated with application of boundary conditions. USim's approach to applying boundary conditions on an unstructured mesh is a two-step process:

1. The user defines the regions of the mesh (*entity*) that will be used to apply boundary conditions.
2. The user specifies the boundary conditions to apply on each region of the mesh.

For the specific case of the *Flow Over a Forward Facing Step* example, there are three boundaries that we need to define, which are delimited by position in the streamwise direction, x:

1. Inflow boundary: defined for $x < 0.0$
2. Wall boundary: defined for $0.0 < x < 3.0$
3. Outflow boundary: defined for $x > 3.0$

We can generate *boundary* entities that exist on the exterior of the mesh using a combination of the *createNewEntityFromMask* and *addEntityMaskExpression* macros:

```
createNewEntityFromMask(<entityName>)
addEntityMaskExpression(<entityName>,<logicalExpression>)
```

Using this combination of macros will result in an entity with name <*entityName*> being generated when <*logicalExpression*> expression evaluates to 1. For the three entities needed for the *Flow Over a Forward Facing Step*, these operations take the form:

```
# Inflow for x < 0.0
createNewEntityFromMask(inflowEntity)
addEntityMaskExpression(inflowEntity,if(x<0.0,1.0,-1.0))

# Wall for 0.0 < x < 3.0
createNewEntityFromMask(wallEntity)
addEntityMaskExpression(wallEntity,if( (x>0.0) and (x<3.0),1.0,-1.0))

# Outflow for x > 3.0
createNewEntityFromMask(outflowEntity)
addEntityMaskExpression(outflowEntity,if(x>3.0,1.0,-1.0))
```

Now that we have created the *inflowEntity*, *wallEntity* and *outflowEntity*, we can specify boundary conditions on them. For the *wallEntity* and the *outflowEntity*, the boundary conditions are familiar from our previous tutorials:

```
boundaryCondition(wall,wallEntity)
boundaryCondition(copy,outflowEntity)
```

For the *inflowEntity*, we specify a new type of boundary condition *userSpecified* to determine the inflow properties:

```
boundaryCondition(userSpecified,inflowEntity)
```

The *userSpecified* boundary condition allows the user to specify the properties of the flow on the boundary entity (here *inflowEntity*) using what should now be a familiar three-step process:

1. Use *addBoundaryConditionVariable* to add variables that are independent of grid position.
2. Use *addBoundaryConditionPreExpression* to add quantities that are functions of grid position, variables and any previously defined PreExpression in this block. Evaluated before expressions and the result is not accessible outside of this block. Any number of PreExpressions can be added.
3. Use *addBoundaryConditionExpression* to define each boundarycondition for the fluid. There is one expression for density, each component of momentum and the total energy. The order of the exprssions correspond to the order in the state vector and there can only be one expression per entry in the state vector.

Note that the macros for performing each of these steps take the form:

```
addBoundaryConditionVariable(<boundaryCondition>,<entityName>,<variableName>,<variableValue>)
addBoundaryConditionPreExpression(<boundaryCondition>,<entityName>,<preExpression>)
addBoundaryConditionExpression(<boundaryCondition>,<entityName>,<Expression>)
```

These macros are documented at euler-macro and idealmhd-macro. For the boundary condition on the *inflowEntity*, the steps are the same as we specified for the initial condition (i.e. the inflow boundary is held in the same state as at time $t = 0$):

```
# Step 1: Add Variables
addBoundaryConditionVariable(userSpecified,inflowEntity,gasGamma,GAS_GAMMA)

# Step 2: Add Pre-Expressions
addBoundaryConditionPreExpression(userSpecified,inflowEntity,rho = gasGamma)
addBoundaryConditionPreExpression(userSpecified,inflowEntity,vx = 3.0)
addBoundaryConditionPreExpression(userSpecified,inflowEntity,vy = 0.0)
addBoundaryConditionPreExpression(userSpecified,inflowEntity,vz = 0.0)
addBoundaryConditionPreExpression(userSpecified,inflowEntity,pr = 1.0)

# Step 3: Add expressions specifying boundary condition on density,
# momentum, total energy
addBoundaryConditionExpression(userSpecified,inflowEntity,rho)
addBoundaryConditionExpression(userSpecified,inflowEntity,rho*vx)
addBoundaryConditionExpression(userSpecified,inflowEntity,rho*vy)
addBoundaryConditionExpression(userSpecified,inflowEntity,rho*vz)
addBoundaryConditionExpression(userSpecified,inflowEntity,(pr/(gasGamma-1.0))+0.5*rho*(vx*vx))
```

3.5.10 Adding Output Diagnostics

USim can compute additional quantities during the simulation that are of interest to the user. This is accomplised by the use of the macro:

```
addOutputDiagnostic(<outputDiagnosticName>)
```

This macro is documented at euler-macro and idealmhd-macro. Once an output diagnostic has been defined, USim allows the user to compute the diagnostic using the familiar three-step process:

1. Use *addOutputDiagnosticVariable* to add variables that are independent of grid position.
2. Use *addOutputDiagnosticPreExpression* to add quantities that are functions of grid position, variables and any previously defined PreExpression in this block. Evaluated before expressions and the result is not accessible outside of this block. Any number of PreExpressions can be added.

3. Use *addOutputDiagnosticExpression* to compute the diagnostic.

Note that the macros for performing each of these steps take the form:

```
addOutputDiagnosticVariable(<outputDiagnosticName>,<variableName>,<variableValue>)
addOutputDiagnosticPreExpression(<outputDiagnosticName>,<preExpression>)
addOutputDiagnosticExpression(<outputDiagnosticName>,<Expression>)
```

These macros are documented at euler-macro and idealmhd-macro. In the *Flow Over a Forward Facing Step* example, we compute the mach number ($M = \frac{|u|}{c_s}$) as follows:

```
#  Add a diagnostic to compute the Mach number of the flow
addOutputDiagnostic(machNumber)
addOutputDiagnosticParameter(machNumber,gasGamma,GAS_GAMMA)
addOutputDiagnosticPreExpression(machNumber,"V=sqrt(Vx^2+Vy^2+Vz^2)")
addOutputDiagnosticPreExpression(machNumber,"Cs=sqrt(gasGamma*P/rho)")
addOutputDiagnosticExpression(machNumber,"V/Cs")
```

Note: For the Euler equations, the following variables are pre-defined when computing an output diagnostic:

- rho
- rhoVx
- rhoVy
- rhoVz
- En
- Vx
- Vy
- Vz
- P

For the MHD equations, the following variables are pre-defined when computing an output diagnostic:

- rho
- rhoVx
- rhoVy
- rhoVz
- En
- Vx
- Vy
- Vz
- P
- Pb
- divB
- Jx
- Jy
- Jz

3.5.11 Putting it all Together

The final step in the USim simulation is to add:

```
runFluidSimulation()
```

This tells USim that we're done specifying the simulation and that it can be run. So, our simulation now looks like:

```
# Are we solving the MHD equations?
$ MHD = False

# Import macros to setup simulation
$ import fluidsBase.mac
$ if MHD
$ import idealmhd.mac
$ else
$ import euler.mac
$ endif

# Specify parameters for the specific physics problem
# grid for simulation
$ GRIDFILE = "forwardFacingStep"
# format of grid
$ GRIDTYPE = "gmsh"
# adiabatic index
$ GAS_GAMMA = 1.4
# Magnetic field strength
$ BETA = 1.0e3
# end time for simulation
$ TEND = 4.0
# number of frames
$ NUMDUMPS = 20
# Riemann solver
$ DIFFUSIVE = True
# Order in time
$ TIME_ORDER = "second"
# Write data for restarting the simulation
$ WRITE_RESTART = False
# Output info for debugging purposes
$ DEBUG = False
#  Dimensionality
$ NDIM = 2
# Permeability of free space
$ MU0 = 1.0
# CFL condition
$ CFL = 0.4

# Initialize a USim simulation
$ if MHD
initializeFluidSimulation(NDIM,0.0,TEND,NUMDUMPS,CFL,GAS_GAMMA,MU0,WRITE_RESTART,DEBUG)
$ else
initializeFluidSimulation(NDIM,0.0,TEND,NUMDUMPS,CFL,GAS_GAMMA,WRITE_RESTART,DEBUG)
$ endif

$ if isEqualString(GRIDTYPE,ExodusII)
addExodusGrid(GRIDFILE)
$ else
addGmshGrid(GRIDFILE)
$ endif
```

```
# Create data structures needed for the simulation
createFluidSimulation()

# Step 1: Add Variables
addVariable(gasGamma,GAS_GAMMA)

# Step 2: Add Pre-Expressions
addPreExpression(rho = gasGamma)
addPreExpression(vx = 3.0)
addPreExpression(vy = 0.0)
addPreExpression(vz = 0.0)
addPreExpression(pr = 1.0)

# Step 3: Add expressions specifying initial condition on density,
# momentum, total energy
addExpression(rho)
addExpression(rho*vx)
addExpression(rho*vy)
addExpression(rho*vz)
addExpression((pr/(gas_gamma-1))+0.5*rho*(vx*vx+vy*vy+vz*vz))

# Create boundary entities
# Inflow for x < 0.0
createNewEntityFromMask(inflowEntity)
addEntityMaskExpression(inflowEntity,if(x<0.0,1.0,-1.0))

# Wall for 0.0 < x < 3.0
createNewEntityFromMask(wallEntity)
addEntityMaskExpression(wallEntity,if( (x>0.0) and (x<3.0),1.0,-1.0))

# Outflow for x > 3.0
createNewEntityFromMask(outflowEntity)
addEntityMaskExpression(outflowEntity,if(x>3.0,1.0,-1.0))

 # Add the spatial discretization of the fluxes
finiteVolumeScheme(DIFFUSIVE)

# Boundary conditions
boundaryCondition(wall,wallEntity)
boundaryCondition(userSpecified,inflowEntity)
boundaryCondition(copy,outflowEntity)

# Specify the inflow boundary condition according to the problem initial conditions
addBoundaryConditionVariable(userSpecified,inflowEntity,gasGamma,GAS_GAMMA)
addBoundaryConditionPreExpression(userSpecified,inflowEntity,rho = gasGamma)
addBoundaryConditionPreExpression(userSpecified,inflowEntity,vx = 3.0)
addBoundaryConditionPreExpression(userSpecified,inflowEntity,vy = 0.0)
addBoundaryConditionPreExpression(userSpecified,inflowEntity,vz = 0.0)
addBoundaryConditionPreExpression(userSpecified,inflowEntity,pr = 1.0)

#  Expressions to specify the inflow boundary
addBoundaryConditionExpression(userSpecified,inflowEntity,rho)
addBoundaryConditionExpression(userSpecified,inflowEntity,rho*vx)
addBoundaryConditionExpression(userSpecified,inflowEntity,rho*vy)
addBoundaryConditionExpression(userSpecified,inflowEntity,rho*vz)
addBoundaryConditionExpression(userSpecified,inflowEntity,(pr/(gasGamma-1.0))+0.5*rho*(vx*vx))

# Add user-specfied diagnostics
```

```
#  Add a diagnostic to compute the Mach number of the flow
addOutputDiagnostic(machNumber)
addOutputDiagnosticParameter(machNumber,gasGamma,GAS_GAMMA)
addOutputDiagnosticPreExpression(machNumber,"V=sqrt(Vx^2+Vy^2+Vz^2)")
addOutputDiagnosticPreExpression(machNumber,"Cs=sqrt(gasGamma*P/rho)")
addOutputDiagnosticExpression(machNumber,"V/Cs")

# Time integration
timeAdvance(TIME_ORDER)

# Run the simulation!
runFluidSimulation()
```

We can now apply our same approach to generalize this to performing simulations on an unstructured mesh in USim:

```
# Are we solving the MHD equations?
$ MHD = True

# Import macros to setup simulation
$ import fluidsBase.mac
$ if MHD
$ import idealmhd.mac
$ else
$ import euler.mac
$ endif

# Specify parameters for the specific physics problem
$ PARAM_1 = <value>
$ PARAM_2 = <value>
$ PARAM_N = <value>

# Initialize a USim simulation
$ if MHD
initializeFluidSimulation(NDIM,0.0,TEND,NUMDUMPS,CFL,GAS_GAMMA,MU0,WRITE_RESTART,DEBUG)
$ else
initializeFluidSimulation(NDIM,0.0,TEND,NUMDUMPS,CFL,GAS_GAMMA,WRITE_RESTART,DEBUG)
$ endif

# Setup the grid
$ if isEqualString(GRIDTYPE,ExodusII)
addExodusGrid(GRIDFILE)
$ else
addGmshGrid(GRIDFILE)
$ endif

# Create data structures needed for the simulation
createFluidSimulation()

# Specify initial condition
# Step 1: Add Variables
addVariable(NAME,<value>)

# Step 2: Add Pre-Expressions
addPreExpression(<PreExpression>)

# Step 3: a) Add expressions specifying initial condition on density,
# momentum
addExpression(<expression>)
$ if MHD
```

```
# Step 3: b) Add expression specifying initial conditions on total
# energy, magnetic field, correction potential
addExpression(<expression>)
$ else
# Step 3: b) Add expression specifying initial conditions on total
# energy
addExpression(<expression>)
$ endif

# Define boundary entities
createNewEntityFromMask(<entityName>)
addEntityMaskExpression(<entityName>,<logicalExpression>)

# Add the spatial discretization of the fluxes
finiteVolumeScheme(DIFFUSIVE)

# Boundary conditions
boundaryCondition(<boundaryCondition,entity>)

# User-specified boundary conditions
boundaryCondition(userSpecified, <entityName>)
addBoundaryConditionVariable(<boundaryCondition>,<entityName>,<variableName>,<variableValue>)
addBoundaryConditionPreExpression(<boundaryCondition>,<entityName>,<preExpression>)
addBoundaryConditionExpression(<boundaryCondition>,<entityName>,<Expression>)

# User-specified output diagnostics
addOutputDiagnostic(<outputDiagnosticName>)
addOutputDiagnosticVariable(<outputDiagnosticName>,<variableName>,<variableValue>)
addOutputDiagnosticPreExpression(<outputDiagnosticName>,<preExpression>)
addOutputDiagnosticExpression(<outputDiagnosticName>,<Expression>)

# Time integration
timeAdvance(TIME_ORDER)

# Run the simulation!
runFluidSimulation()
```

3.5.12 An Example Simulation

The input file for the problem *Flow over a forward facing step* in the USimBase package demonstrates each of the concepts described above. Executing this input file within USimComposer and switching to the *Visualize* tab yields the plots shown in Fig. 3.6.

Note: For more depth, you can view the actual input blocks to Ulixes in the *Setup* window by choosing *Save And Process Setup* and then clicking on the *forwardFacingStep.in* file. In the *.in* file all macros are expanded to produce input blocks.

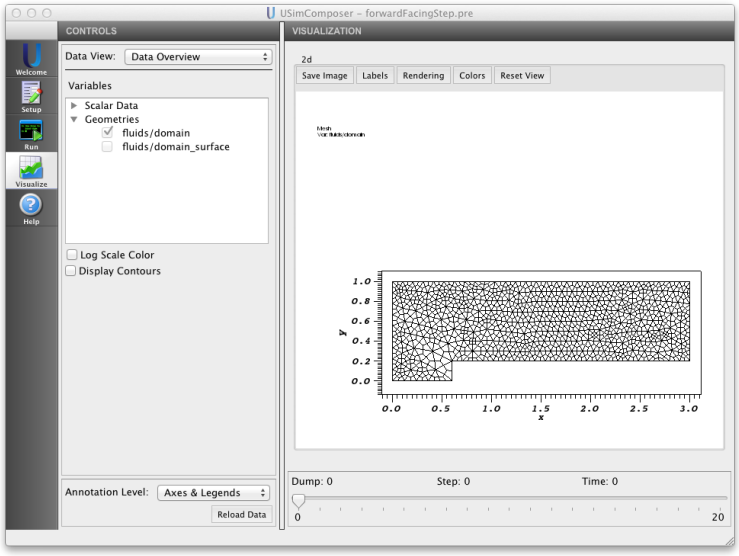


Fig. 3.6: Visualization of mesh geometry in USimComposer after executing the *Flow over a forward facing step* input file for the tutorial.

CHAPTER

FOUR

ADVANCED USIM SIMULATIONS

The following tutorials can be worked through with a *USimBase* license and described how to perform simulations using USim by working directly with the input file.

4.1 Advanced USim Simulation Concepts

In *Using USim to solve the Euler Equations*, *Solving Problems on Unstructured Meshes in USim*, we saw how to solve the Euler and MHD equations for a range of problems using USim. The algorithms that we used to do this were created using a range of macros. USim simulations can also be created **without** the use of these macros, allowing the user much greater control over the simulation design process and enabling simulations with greater complexity. The next set of tutorials describes how to create USim simulations without the use of macros. It does so by making use of the examples in USimBase again, but this time we examine the actual input blocks generated from the macros. This is done in the *Setup* window by choosing *Save And Process Setup* and then clicking on the <*exampleName*>*.in* tab. In the *.in* tab all macros are expanded to produce input blocks.

We start this set of tutorials by presenting advanced concepts about USim that you should understand before creating and running USim simulations without using the macros described in earlier tutorials. Taking the time to examine these concepts before reading about the simulation process will make the simulation procedures simple to understand and tutorial lessons straightforward to follow.

Contents

4.1.1 The Fluids Component

USim input files without macros look like XML files. Input files can contain nested blocks, each block can contain double, integer and string values and vectors (arrays) of these types. Remember: if you want to specify a double in the input file you *must not* omit the decimal point. Otherwise, the number will be interpreted as an integer. Similarly, if you want an array of doubles, the *first element* must have a decimal point. Hence, what follows are examples of a double scalar and a double vector:

```
value = 1.0
listOfNumbers = [1.0, 2.0, 3.0]
```

The simplest example of an input file given below. It basically does nothing but starts USim and quits:

```
# start and end times
tStart = 0.0
tEnd = 0.2
# number of frames to write
numFrames = 2
# initial time-step to use
initDt = 0.01
verbosity = debug
defaultParallelSync = false

# top level component
<Component fluids>
  kind = updaterComponent
</Component>
```

All input files need a start and end time for the simulation, specified as *tStart* and *tEnd*. The *numFrames* variable tells USim how many frames of data to write. USim will always write the initial frame and then will write additional *numFrames* frames. Each frame is an HDF5 file named same as the input file with numbers appended to indicate frame number. For example, *euler_0.h5*, *euler_1.h5* and *euler_2.h5* will be created by the above simulation. More information on USim data output is given in *Writing Out Data*.

The initial time-step is specified in *initDt*. This can be set to whatever you want as USim will simply adjust this depending on the physics included in the simulation.

The option *defaultParallelSync=false* tells USim that the user will specify the parallel synchronization of the data in the simulation. *defaultParallelSync* is optional, but defaults to *false*. If *defaultParallelSync=true* then parallel data synchronization is done automatically. Automatic synchronization is less efficient than user defined synchronization, but avoids many of the pitfalls. If you are having problems with a parallel simulations, a simple check of your synchronization is to set *defaultParallelSync=true*.

In each simulation there must be *exactly one* top-level *Component* block. This block is called *fluids*. You can name it anything you want and this name will be reflected in the output HDF5 files. Many blocks take a special field called the *kind* field. This field tells USim what kind of *Component* (in this case) to create. For now, we will use the standard *updaterComponent* component kind.

4.1.2 Defining the Simulation Grid

Every object in a USim simulation interacts with a grid in one, two or three dimensions, in either Cartesian or axisymmetric Cylindrical coordinates. Grids in USim can either be rectangular, body-fitted or unstructured. The simplest example is a one-dimensional rectangular grid, which here we name *domain*:

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

This will create a 512 zone, 1D Cartesian grid spanning the physical space $(0.0, 1.0)$. Instructions for creating multi-dimensional Cartesian grids, body-fitted grids and unstructured meshes can be found in other sections of this manual.

4.1.3 Allocating Memory

Data can be allocated on the grid defined in *Defining the Simulation Grid* through the code block such as:

```
<DataStruct q>
  kind = nodalArray
  onGrid = domain
  numComponents = 5
</DataStruct>
```

This block will create a *nodalArray* array called "q". A nodal array is one that interacts with rectangular, body-fitted and unstructured meshes and is decomposed using MPI when run in parallel. Note we have to specify the name of the grid the data lives on using the *onGrid* field. The field *numComponents* tells USim that we wish to store 5 components in this array.

Now our input file should look like:

```
# start and end times
 tStart = 0.0
 tEnd = 0.2
# number of frames to write
 numFrames = 2
# initial time-step to use
 initDt = 0.01
 verbosity = debug

# top level component
 <Component fluids>
   kind = updaterComponent

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

   <DataStruct q>
     kind = nodalArray
     onGrid = domain
     numComponents = 5
   </DataStruct>

 </Component>
```

This input file can be executed within USimComposer.

4.1.4 Setting Initial Conditions

Now that we have allocated an array we will initialize it. USim provides a block called *Updater* to perform action on data. There are many Updaters and we will use them extensively in constructing a simulation. Updaters are very powerful and can be used to create very complex simulations by combining them carefully. Think of an updater as a generalization of a subroutine or function.

Let's create an updater, named "init" to initialize the "q" array. This updater will use a expressions to initialize all the components of "q". As a simple example, we will use the initial condition for a one-dimensional Sod Shock tube, which consists of a left-state and a right-state separated by a discontinuity:

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

# initial condition to use
   <Function func>
     kind = exprFunc

# location of discontinuity
     sloc = 0.5

# gas density in left state
     rhol = 3.0
# gas pressure in left state
     prl = 3.0
# gas density in right state
     rhor = 1.0
# gas pressure in right state
     prr = 1.0

# Adiabatic index, or ratio of specific heats
     gasGamma = $GAMMA$

     preExprs = [ \
         "rho = if (x>sloc, rhor, rhol)", \
         "pr = if (x>sloc, prr, prl)"]
     exprs = ["rho", "0.0", "0.0", "0.0", "pr/(gasGamma-1)"]

   </Function>

 </Updater>
```

Each updater must have a *kind* field and an *onGrid* field. This updater is of kind "initialize1d" and runs on the grid "domain" that we created previously. Updaters usually have *in* and *out* fields that specify which datastructures are input and which are output. Other fields and blocks depend on the updater kind. In this updater we are using a *Function* block to specify a function for use in the initial conditions. The kind of function we are using is the "exprFunction" that uses expressions. This particular expression sets the following initial condition

$$\rho = 3.0 \quad x < 0.5 \tag{4.1}$$

$$\rho = 1.0 \quad x > 0.5 \tag{4.2}$$

$$P_g = 3.0 \quad x < 0.5 \tag{4.3}$$

$$P_g = 1.0 \quad x > 0.5 \tag{4.4}$$

If we simply put the above block in the input file and run the simulation nothing will happen! I.e. the initial conditions will not be applied. The reason is that updaters do not run automatically. We have to specify when the updater will be run using *UpdateStep* blocks. To do this we add the following block in the input file:

```
<UpdateStep initStep>
  updaters = [init]
  syncVars = [q]
</UpdateStep>
```

This tells USim that when the *initStep* updater-step is run to call the updater "init". The variable *syncVars* specifies which variables should be synchronized AFTER the list of updaters is called. In this case the variable q is synchronized. If *defaultParallelSync=false* then syncVars can be dropped and USim will automatically synchronize all variables in the *out* list of the updaters.

The final step to have this updater-step run is to add a *single UpdateSequence* block:

```
<UpdateSequence sequence>
  startOnly = [initStep]
  loop = []
  writeOnly = []
</UpdateSequence>
```

Thats it! When we run the input file the "q" array will be initialized with the updater. Now our complete input file looks like:

```
# Adiabatic index, or ratio of specific heats
 $GAMMA = 1.4

# start and end times
 tStart = 0.0
 tEnd = 0.2
# number of frames to write
 numFrames = 2
# initial time-step to use
 initDt = 0.01
 verbosity = debug

# top level component
 <Component fluids>
   kind = updaterComponent

   <Grid domain>
     kind = cart1d
     numLayers = 2
     lower = [0.0]
     upper = [1.0]
     cells = [100]
     isRadial = false
   </Grid>

   <DataStruct q>
     kind = nodalArray
     onGrid = domain
     numComponents = 5
   </DataStruct>

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

# initial condition to use
     <Function func>
       kind = exprFunc

# location of discontinuity
       sloc = 0.5

# gas density in left state
       rhol = 3.0
# gas pressure in left state
       prl = 3.0
# gas density in right state
```

```
       rhor = 1.0
# gas pressure in right state
       prr = 1.0

# Adiabatic index, or ratio of specific heats
       gasGamma = $GAMMA$

       preExprs = [ \
           "rho = if (x>sloc, rhor, rhol)", \
           "pr = if (x>sloc, prr, prl)"]

       exprs = ["rho", "0.0", "0.0", "0.0", "pr/(gasGamma-1)"]

     </Function>

   </Updater>

   <UpdateStep initStep>
     updaters = [init]
   </UpdateStep>

   <UpdateSequence sequence>
     startOnly = [initStep]
     loop = []
     writeOnly = []
   </UpdateSequence>

 </Component>
```

4.1.5 Writing Out Data

All data produced by USim is output using the HDF5 data file format. These output files are dumped as defined by the user; they can be written at the end of a simulation, for example, or every n steps to create a time series that can be used to see how a system evolves over time. Additionally, these output files can be used to restart a run and continue from a given point; for example, if a user has run a simulation for 1000 time steps and wishes to see how the simulation progresses if run for another 1000.

Execution of USim leads to the generation of the data from the simulation in the form of HDF5 `.h5` files. There are separate executables for serial and parallel runs.

Writing out Additional Data

If we plot the *q_4* variable in Composer we will be plotting the total energy and not the gas pressure. To compute the pressure we can add another array called *pressure* to store the pressure:

```
<DataStruct pressure>
  kind = nodalArray
  onGrid = domain
  numComponents = 1
</DataStruct>
```

Next, we add an "combiner" updater to extract the pressure:

```
 <Updater pressCalc>
   kind = combiner1d
```

```
   onGrid = domain
# input array
   in = [q]

# ouput data-structures
   out = [pressure]

   indVars_q = ["rho", "rhou", "rhov", "rhow", "Er"]

   gamma = GAMMA
   preExprs = ["pr = (Er - 0.5*(rhou^2+rhov^2+rhow^2)/rho)*(gamma-1)"]
   exprs = ["pr"]

 </Updater>
```

In this updater we assign a name to each of the five components of "q" and use them in expressions to compute the pressure using the formula

$$p = (\gamma - 1)\left(E - 0.5\rho(u^2 + v^2 + w^2)\right)$$

where E is the total fluid energy and (u, v, w) are the fluid velocity components.

Finally, we need to call this updater in an updater-step and add it to the *writeOnly* list in the update-sequence block:

```
<UpdateStep pressureStep>
  updaters = [pressCalc]
</UpdateStep>

<UpdateSequence sequence>
  startOnly = [initStep]
  loop = []
  writeOnly = [pressureStep]
</UpdateSequence>
```

When we run this simulation an array called "pressure" will also be written out that will store the pressure in the simulation. Note that by putting the "pressCalcStep" in the *writeOnly* list we are running the "pressCalcStep" only before we write data to file. It is *not* run every time-step.

4.1.6 An Example Simulation File

The input file for the problem *Shock Tube* in the USimBase package demonstrates each of the concepts described above to evolve the classic Sod Shock tube problem in one-dimensional hydrodynamics. You can view the actual input blocks to Ulixes in the *Setup* window by choosing *Save And Process Setup* and then clicking on the *shockTube.in* file. In the *.in* file all macros are expanded to produce input blocks.

4.2 Advanced Methods for Solving the Euler Equations with USim

In *Advanced USim Simulation Concepts* we examined the basic ingredients of a USim input file: the simulation grid (see *Defining the Simulation Grid*); data structures (see *Allocating Memory*); how to assign initial conditions (see *Setting Initial Conditions*) and how to write out additional data (see *Writing Out Data*). In this tutorial, we build on these concepts and demonstrate the basic methods used by USim to solve the Euler equations without the use of macros.

This tutorial is based on the quickstart-shocktube example. The *Shock Tube* simulation is designed to set up a variety of tube simulations including those by Einfeldt, Sod, Liska & Wendroff, Brio & Wu, and Ryu & Jones. In this tutorial, we will look at the Sod Shock Tube based on the classic paper:

```
Sod, Gary A. "A survey of several finite difference methods for
systems of nonlinear hyperbolic conservation laws." ; Journal of Computational Physics 27.1 (1978): 1
```

and the use of equations for inviscid compressible hydrodynamics (the *Euler* equations), which were described in *Using USim to solve the Euler Equations*. The addition of a basic grid and how to setup an initial condition was covered in *Advanced USim Simulation Concepts*. In this tutorial, we discuss how to setup USim algorithms for solving the Euler equations directly without using macros.

Contents

4.2.1 Allocating Simulation Memory

The Euler equations consist of partial differential equations that describe the evolution of density, momentum and total energy. refmanual-eulerEqn documents the number of components that each data structure needs to contain to solve this set of equations; which is five:

```
<DataStruct q>
  kind = nodalArray
  onGrid = domain
  numComponents = 5
</DataStruct>
```

4.2.2 Computing the Fluxes

USim implements the well-known MUSCL scheme to compute the spatial discretization of a hyperbolic conservation law (see *Using USim to solve the Euler Equations* for a more detailed description). In *Using USim to solve the Euler Equations*, we added this algorithm through:

```
finiteVolumeScheme(DIFFUSIVE)
```

For the Euler equations, this macro expands to produce a :ref: *refmanual-classicMuscl*, shown below:

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

  # input data-structures
  in=[q]
   # output data-structures
  out=[qNew]
  # CFL number to use
  cfl=0.5
  # legacy time integration scheme, attribute; should be set to "none"
  timeIntegrationScheme=none
  # Riemann solver
```

```
  numericalFlux=hllcEulerFlux
   # Limiter to use
  limiter=[muscl]
  # Form of variable to limit
  variableForm=primitive
  # Whether to check variables for physical validitiy
  preservePositivity=0

  # Hyperbolic equation system
  <Equation euler>
    kind=eulerEqn
    # Adiabatic index
    gasGamma=1.6666666666666667
  </Equation>

   # Hyperbolic equation to solve
  equations=[euler]
</Updater>
```

Details of the block and the meaning of each component are given in refmanual-classicMuscl . The Updater computes the numerical flux for the hyperbolic system. This same Updater can be used for all hyperbolic equations available in USim.

4.2.3 Applying Boundary Conditions

At each time step, we have to apply boundary conditions at the left and right of the domain to ensure that at the next time step, physically-valid data is used to update the conserved state. Without this, the simulation will fail. It is possible to specify arbitrary boundary conditions in USim.

For the Sod shock tube example considered here, appropriate boundary conditions are outflow ("open") boundary conditions at both ends of the domain. This was done in *Using USim to solve the Euler Equations* using the macros:

```
boundaryCondition(copy,left)
boundaryCondition(copy,right)
```

These expand to yield:

```
<Updater copyBoundaryOnEntityleft>
  kind=copy1d
  onGrid=domain
  in=[q]
  out=[q]
  entity=left
</Updater>

<Updater copyBoundaryOnEntityright>
  kind=copy1d
  onGrid=domain
  in=[q]
  out=[q]
  entity=right
</Updater>
```

The copy boundary condition block is described in refmanual-copy. This boundary condition updater copies the values on the layer next to the ghost cells into the ghost cells - this is equivalent to a zero derivative boundary condition.

If we are evolving in more than one-dimension, we have to specify boundary conditions on the rest of the domain boundaries. This was done in *Using USim to solve the Euler Equations* using the macro:

```
boundaryCondition(periodic)
```

This expands to yield:

```
<Updater periodicBoundaryOnEntityghost>
  kind=periodicCartBc1d
  onGrid=domain
  in=[q]
  out=[q]
</Updater>
```

The periodic boundary condition updater is documented at refmanual-periodicCartBc).

4.2.4 Advancing By A Time Step

In order to solve the Euler equations, we have to advance the conserved quantities from time t to $t + \Delta t$. This is done by applying a time integration scheme. In *Using USim to solve the Euler Equations*, we did this using the macro:

```
timeAdvance(TIME_ORDER)
```

This expands to yield:

```
<Updater mainIntegrator>
  kind=multiUpdater1d
  onGrid=domain
  in=[q]
  out=[qNew]

  <TimeIntegrator timeStepper>
    kind=rungeKutta1d
    onGrid=domain
    scheme=second
  </TimeIntegrator>


  <UpdateStep boundaryStep>
    operation=boundary
    updaters=[copyBoundaryOnEntityleft copyBoundaryOnEntityright periodicBoundaryOnEntityghost]
    syncVars=[q]
  </UpdateStep>


  <UpdateStep integrationStep>
    operation=integrate
    updaters=[hyper]
    syncVars=[qNew]
  </UpdateStep>


  <UpdateSequence sequence>
    startOnly=[]
    restoreOnly=[]
    writeOnly=[]
    loop=[boundaryStep   integrationStep]
  </UpdateSequence>

</Updater>
```

This is the refmanual-multiUpdater, which is the most powerful updater available in USim. It enables the user to combine together almost any of the refmanual-updaters available in USim and apply explicit, super-time step or fully-implicit refmanual-timeIntegrationSchemes. Details of each block can be found in refmanual-multiUpdater, however here are a few key points. The multiUpdater integrates system of equations in time. The time integration method is specified in *TimeIntegrator*, after that a series of *UpdateSteps* are defined. Each *UpdateStep* contains an *operation* type (which is optional), a list of *updaters* that are evaluated during the step and an optional list of *syncVars* which specifies which variables should synchronized after the updateStep is called. Synchronization (*syncVars*) are important for parallel computation and tell USim to synchronize a DataStruct across a parallel domain. Default synchronization can be specified by setting *defaultParallelSync=true* at the top of the file - at that point all the *syncVars* can be left blank. This approach is less efficient than manual synchronization, but is less prone to user error.

The final element of advancing the conserved quantities from time t to $t + \Delta t$ is to copy the updated data in *qnew* to q. In USim, we accomplish this through use of a *Linear Combiner*:

```
<Updater copier>
  kind=linearCombiner1d
  onGrid=domain
  in=[qNew]
  out=[q]
  coeffs=[1.0]
</Updater>
```

This updater block is also generated when we expand the *timeAdvance(TIME_ORDER)* macros and is described in refmanual-linearCombiner This updater solves the equation $Out = Coeffs * In$, where in this case $In = q_{New}$, $Out = q$ and $Coeffs = 1.0$.

4.2.5 Putting it all Together

In *Using USim to solve the Euler Equations*, we told USim that we're done specifying the simulation and that it can be run by calling the macro:

```
runFluidSimulation()
```

This macro collects up all of the updaters that we have added so far and, based on the sequence that we have added them, figures out how to call them to run a USim simulation. For the Sod shock case in *ShockTube* example, the computations performed by this macro results in the following refmanual-updateSteps and refmanual-updateSequence being generated:

```
<UpdateStep startStep>
  updaters=[setVar copyBoundaryOnEntityleft copyBoundaryOnEntityright periodicBoundaryOnEntityghost]
  syncVars=[q]
</UpdateStep>

<UpdateStep restoreStep>
  updaters=[]
  syncVars=[]
</UpdateStep>

<UpdateStep bcStep>
  updaters=[copyBoundaryOnEntityleft copyBoundaryOnEntityright periodicBoundaryOnEntityghost]
  syncVars=[q]
</UpdateStep>

<UpdateStep hyperStep>
  updaters=[mainIntegrator copier]
  syncVars=[]
</UpdateStep>
```

```
<UpdateStep writeStep>
  updaters=[computePressure computeVelocity computeDensity]
  syncVars=[]
</UpdateStep>
```

Each *UpdateStep* block calls a number of updaters in in the order given in the *updaters* string list and when USim is executed in parallel, handle synchronization of data across processors through the *syncVar* string list. Note that if these lists are empty and the Update Step is called, then it simply returns without doing anything.

As part of the expansion of the *runSimulation* macro, the *UpdateStep* blocks are added to appropriate string lists in the refmanual-UpdateSequence:

```
<UpdateSequence simulation>
  startOnly=[generateStep   startStep]
  restoreOnly=[generateStep   restoreStep   bcStep]
  writeOnly=[bcStep   writeStep]
  loop=[hyperStep]
</UpdateSequence>
```

UpdateSteps listed in the *startOnly* string list are executed at the beginning of the simulation and generally are used to define generate entities in the grid and definine iniital conditions. UpdateSteps listed in the *restoreOnly* string list are executed when a simulation is restartedand generally are used to define generate entities in the grid, ensure that data structures contain valid data and apply boundary conditions. UpdateSteps in the *writeOnly* string list are executed only when data is output to a file. Finally, UpdateSteps listed in the *loop* string list are executed at each time step.

An Example Simulation File

The input file for the problem *Shock Tube* in the USimBase package demonstrates each of the concepts described above to evolve the classic Sod Shock tube problem in one-dimensional hydrodynamics. You can view the actual input blocks to Ulixes in the *Setup* window by choosing *Save And Process Setup* and then clicking on the *shockTube.in* file. In the *.in* file all macros are expanded to produce input blocks.

4.2.6 Advanced USim Simulation Structure

In earlier tutorials, we developed a simple pattern that could be used to design USim simulations using macros. This pattern can be repeated when we don't use macros; however, we now need to add the grids, data structures, updaters, boundary conditions and time integration schemes by hand and then tell USim how to run them. An initial pattern looks like the following:

```
# Specify parameters for the specific physics problem
$ PARAM_1 = <value>
$ PARAM_2 = <value>
$ PARAM_N = <value>

# Initialize a USim simulation
# Simulation start and end times
tStart = <float>
tEnd = <float>
# Number of data files to write
numFrames = <integer>
# Initial time-step to use
initDt = <float>
# Level of feedback to give user
verbosity = <info/debug>
```

```
<Component fluids>
  kind = updaterComponent

   # Setup the grid
  <Grid Grid_Name (type=string)>
    <grid parameters>
  </Grid>

   # Create data structures needed for the simulation
  <DataStruct DataStruct_Name1 (type=string)>
    kind = nodalArray
    onGrid = <Grid_Name>
    numComponents = <int>
  </DataStruct>

  <DataStruct DataStruct_Name2 (type=string)>
    kind = nodalArray
    onGrid = <Grid_Name>
    numComponents = <int>
  </DataStruct>

   <DataStruct DataStruct_NameN (type=string)>
    kind = nodalArray
    onGrid = <Grid_Name>
    numComponents = <int>
  </DataStruct>

  # Specify initial condition
  <Updater Initialization_Updater_Name (type=string)>
    kind = initialize<NDIM>d
    onGrid = <Grid_Name>
    out = <DataStruct_Name for t = 0>

    # initial condition to use
    <Function func>
      kind = exprFunc

      # Step 1: Add Variables
      VARIABLE_1 = <float>
      VARIABLE_2 = <float>
      VARIABLE_N = <float>


      # Step 2: Add Pre-Expressions
      preExprs = [ \
          "<PreExpression_1>", \
          "<PreExpression_2>", \
          "<PreExpression_N>"
          ]

     # Step 3: Add expressions specifying initial condition on density,
     # momentum, total energy
     addExpression(<expression>)

      exprs = ["<density_expression>",  \
                    "<xMomentum_expression>",  \
                    "<yMomentum_expression>",  \
                    "<zMomentum_expression>", \
```

```
                    "<totalEnergy_expression>"]

  </Function>

</Updater>

# Add the spatial discretization of the fluxes
<Updater FiniteVolume_Updater_Name (type=string)>
  kind=classicMuscl<NDIM>d
  onGrid=<Grid_Name>

  # input data-structures
  in=<DataStruct_Name for t^n>
   # output data-structures
  out=<DataStruct_Name for \nabla \cdot F(<DataStruct_Name for t^n>)>
  # CFL number to use
  cfl=<float>
  # legacy time integration scheme, attribute; should be set to "none"
  timeIntegrationScheme=none
  # Riemann solver
  numericalFlux=<string>
   # Limiter to use
  limiter=[<string>]
  # Form of variable to limit
  variableForm=<string>
  # Whether to check variables for physical validitiy
  preservePositivity=<int>

  # Hyperbolic equation system
  <Equation euler>
    kind=eulerEqn
    # Adiabatic index
    gasGamma=<float>
  </Equation>

   # Hyperbolic equation to solve
  equations=[euler]
</Updater>

# Boundary conditions
<Updater BoundaryCondition_Name1 (type=string)>
  kind=<string><NDIM>d
  onGrid=<Grid_Name>
  in=<DataStruct_Name for t^n>
  out=<DataStruct_Name for t^n>
  entity=<Entity_Name (type=string)>
  <Boundary Condition Parameters>
</Updater>

# Time integration
<Updater TimeIntegrationUpdater_Name (type=string)>
  kind=multiUpdater<NDIM>d
  onGrid=<Grid_Name>
  in=<DataStruct_Name for t^n>
  out=<DataStruct_Name for t^n+1>

  <TimeIntegrator timeStepper>
    kind=rungeKutta<NDIM>d
```

```
    onGrid=<Grid_Name>
    scheme=<string>
  </TimeIntegrator>

  <UpdateStep boundaryStep>
    operation=boundary
    updaters=<string list of boundary conditions>
    syncVars=<DataStruct_Name at t^n>
  </UpdateStep>

  <UpdateStep integrationStep>
    operation=integrate
    updaters=<string list of integrators>
    syncVars=<DataStruct_Name for \nabla \cdot F(<DataStruct_Name for t^n>)>
  </UpdateStep>

  <UpdateSequence sequence>
    loop=[boundaryStep   integrationStep]
  </UpdateSequence>

</Updater>

<Updater CopierUpdater_Name (type=string)>
  kind=linearCombiner<NDIM>d
  onGrid=<Grid_Name>
  in=<DataStruct_Name for t^n+1>
  out=<DataStruct_Name for t^n>
  coeffs=[1.0]
</Updater>

<UpdateStep startStep>
  updaters=[Initialization_Updater_Name BoundaryCondition_Name1 ... BoundaryCondition_NameN]
  syncVars=[<DataStruct_Name for t^n>]
</UpdateStep>

<UpdateStep restoreStep>
  updaters=<String List of Updaters>
  syncVars=<String List of DataStructs>
</UpdateStep>

<UpdateStep bcStep>
  updaters=[BoundaryCondition_Name1 ... BoundaryCondition_NameN]
  syncVars=[<DataStruct_Name for t^n>]
</UpdateStep>

<UpdateStep hyperStep>
  updaters=[FiniteVolume_Updater_Name (type=string)]
  syncVars=<String List of DataStructs>
</UpdateStep>

<UpdateStep writeStep>
  updaters=<String List of Updaters>
  syncVars=<String List of DataStructs>
</UpdateStep>

<UpdateSequence simulation>
  startOnly=[generateStep   startStep]
  restoreOnly=[generateStep   restoreStep   bcStep]
```

```
    writeOnly=[bcStep   writeStep]
    loop=[hyperStep]
  </UpdateSequence>

</Component>
```

Note that we have not filled in some of the entries in the *UpdateSteps* above. How to use these *UpdateSteps* is discussed in later tutorials.

4.3 Advanced Methods for Solving the Magnetohydrodynamics Equations with USim

In *Advanced USim Simulation Concepts* we examined the basic ingredients of a USim input file: the simulation grid (see *Defining the Simulation Grid*); data structures (see *Allocating Memory*); how to assign initial conditions (see *Setting Initial Conditions*) and how to write out additional data (see *Writing Out Data*). In *Advanced Methods for Solving the Euler Equations with USim*, we built on these concepts and demonstrated the basic methods used by USim to solve the Euler equations without the use of macros.

This tutorial is based on the quickstart-shocktube example. The *Shock Tube* simulation is designed to set up a variety of tube simulations including those by Einfeldt, Sod, Liska & Wendroff, Brio & Wu, and Ryu & Jones. In this tutorial, we will look at the Brio & Wu shock Tube, described by:

```
Brio, M., & Wu, C.~C. (1988), Journal of Computational Physics, 75, 400
```

and the use of equations for ideal compressible magnetohydrodynamics (the *MHD* equations) in one-dimension, which were described in *Using USim to solve the Magnetohydrodynamic Equations*. In this tutorial, we discuss how to setup USim algorithms for solving the MHD equations directly without using macros.

Contents

4.3.1 Allocating Simulation Memory

The MHD equations consist of partial differential equations that describe the evolution of density, momentum. total energy and the magnetic field. refmanual-mhdDednerEqn documents the number of components that each data structure needs to contain to solve this set of equations. In addition to the quantities defined by the MHD equations, the refmanual-mhdDednerEqn evolves an additional partial differential equation that controls the divergence of the magnetic field (the *correction potential*). As a result, our data structures need nine components:

```
<DataStruct q>
  kind = nodalArray
  onGrid = domain
```

```
  numComponents = 9
</DataStruct>
```

i.e. modifying the *numComponents* input block to contain *9* entries. This must be applied to all data structures that contain the conserved state, fluxes or the primitive variables in the input file.

4.3.2 Initializing the Fluid

The initial condition for a MHD simulation follows the same basic three-step pattern outlined previously, but with the need to specify the additional componets associated with the magnetic field and the correction potential:

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

     <Function initFunc>
     kind=exprFunc

     # Step 1: Add Variables
     pi_value=3.141592653589793
     gas_gamma=1.6666666666666667
     densityL=1.0
     densityR=0.125
     pressureL=1.0
     pressureR=0.1
     normalVelocityL=0.0
     normalVelocityR=0.0
     perpendicularVelocityL=0.0
     perpendicularVelocityR=0.0
     tangentialVelocityL=0.0
     tangentialVelocityR=0.0
     mu0=1.0
     normalFieldL=0.75
     normalFieldR=0.75
     perpendicularFieldL=1.0
     perpendicularFieldR=-1.0
     tangentialFieldL=0.0
     tangentialFieldR=0.0

     # Step 2: Add Pre-Expressions
     preExprs=[ rho=if(x>0.0,densityR,densityL)   pr=if(x>0.0,pressureR,pressureL)   vx=if(x>0.0,norm

     # Step 3: Add Expressions
     exprs=[ rho   rho*vx   rho*vy   rho*vz   pr/(gas_gamma-1.0)+0.5*rho*(vx*vx+vy*vy+vz*vz)+0.5*((bx
   </Function>

 </Updater>
```

Notice how each of the three steps are added to the function block that defines the initial condition:

1. Step 1 creates a list of <*variablesName*> = <*value*> pairs; one per line.
2. Step 2 creates a string vector called *preExprs* (short for *preExpressions*). Each of the *addPreExpression* calls described in *Using USim to solve the Magnetohydrodynamic Equations* adds one entry to this vector, which contains the function specified in the *addPreExpression* call.
3. Step 3 creates a string vector called *exprs* (short for *expressions*). As with Step 2, each of the *addExpression*

call described in *Using USim to solve the Magnetohydrodynamic Equations* adds one entry to this vector, which contains the function specified in the *addExpression* call. Note that the order of the *addExpression* calls is preserved in the string vector; this is critical as these entries specify each component of the refmanual-nodalArray specified in the *[out]* attribute.

We will see this pattern repeated whenever we used this three step process in the simple input files.

4.3.3 Evolving the Fluid

USim implements the well-known MUSCL scheme to compute the spatial discretization of a hyperbolic conservation law (see *Using USim to solve the Euler Equations* for a more detailed description). In *Using USim to solve the Magnetohydrodynamic Equations*, we added this algorithm through:

```
finiteVolumeScheme(DIFFUSIVE)
```

For the MHD equations, this macro expands to produce a :ref: *refmanual-classicMuscl*, shown below:

```
<Updater hyper>
  kind=classicMuscl1d
  onGrid=domain
  # input data-structures
  in=[q]
  # output data-structures
  out=[qNew]
  # CFL number to use
  cfl=0.5
  # legacy time integration scheme attribute; should be set to "none"
  timeIntegrationScheme=none
  # Riemann solver
  numericalFlux=hlldFlux
  # Limiter to use
  limiter=[muscl]
  # Form of variable to limit
  variableForm=primitive
  # Whether to check variables for physical validitiy
  preservePositivity=0
  # Maximum wave speed in the grid at this time step
  waveSpeeds=[maxWaveSpeed]

  # Hyperbolic equation system
  <Equation idealMhd>
    kind=mhdDednerEqn
    # Adiabatic index
    gasGamma=1.6666666666666667
    # Permeability of free space; should always be set to unity
    mu0=1.0
  </Equation>

  # Hyperbolic equation to solve
  equations=[idealMhd]
</Updater>
```

Compare to the case for the Euler equations described in *Advanced Methods for Solving the Euler Equations with USim*, this example of the refmanual-classicMuscl updater has **two** key differences:

1. The use of the refmanual-mhdDednerEqn to describe the hyperbolic equation system.
2. The use of the *waveSpeeds = <string vector>* to specify a list of refmanual-dynVectors to provide the fastest wave speed in the grid at the current time step.

The *waveSpeeds* = <*string vector*> attribute is *required* for the refmanual-mhdDednerEqn in order to control the divergence of the magnetic field. These wave speeds are computed dynamically by USim using a refmanual-timeStepRestrictionUpdater, which in the shock tube example looks like:

```
<Updater getMaxWaveSpeed>
  kind=timeStepRestrictionUpdater1d
  onGrid=domain
  # input data-structures
  in=[q]
  # no output data-structures
  out=[]
  # CFL number to use
  courantCondition=0.5
  # DynVector to hold the maximum wave speed
  waveSpeeds=[maxWaveSpeed]

  # Time step restriction to compute
  <TimeStepRestriction idealMhdRestriction>
    # Compute a time step restriction for a hyperbolic equation
    kind=hyperbolic1d
    # Adiabatic index
    gasGamma=1.6666666666666667
    # Permeability of free-space, should always be 1.0
    mu0=1.0
    # Hyperbolic equation to use
    model=mhdDednerEqn
  </TimeStepRestriction>

  # List of time step restrictions to compute
  restrictions=[idealMhdRestriction]
</Updater>
```

This updater computes the global maximum wave speed over the entire simulation based on the refmanual-nodalArray specified in the *in* string vector. The wave speed is stored in the refmanual-dynVector specified in the *waveSpeeds* string vector.

4.3.4 Applying Boundary Conditions

At each time step, we have to apply boundary conditions at the left and right of the domain to ensure that at the next time step, physically-valid data is used to update the conserved state. Without this, the simulation will fail. It is possible to specify arbitrary boundary conditions in USim.

The boundary conditions used for the Brio & Wu shock tube are the same as those used in *Using USim to solve the Euler Equations*, *Using USim to solve the Magnetohydrodynamic Equations* and *Advanced Methods for Solving the Euler Equations with USim*: outflow ("open") boundary conditions at both ends of the domain. This was done in *Using USim to solve the Magnetohydrodynamic Equations* using the macros:

```
boundaryCondition(copy,left)
boundaryCondition(copy,right)
```

These expand to yield:

```
<Updater copyBoundaryOnEntityleft>
  kind=copy1d
  onGrid=domain
  in=[q]
  out=[q]
  entity=left
```

```
</Updater>

<Updater copyBoundaryOnEntityright>
  kind=copy1d
  onGrid=domain
  in=[q]
  out=[q]
  entity=right
</Updater>
```

The copy boundary condition block is described in refmanual-copy. This boundary condition updater copies the values on the layer next to the ghost cells into the ghost cells - this is equivalent to a zero derivative boundary condition. Note that the blocks that describe these boundary conditions are **identical** to those used for the Euler equations (described in *Advanced Methods for Solving the Euler Equations with USim*). This is because these boundary conditions simply copy the values in each component of the refmanual-nodalArray into the ghost cells; USim does this for all components of the supplied input refmanual-nodalArray.

As in *Using USim to solve the Euler Equations*, *Using USim to solve the Magnetohydrodynamic Equations* and *Advanced Methods for Solving the Euler Equations with USim*, If we are evolving in more than one-dimension, we have to specify boundary conditions on the rest of the domain boundaries. This was done in *Using USim to solve the Magnetohydrodynamic Equations* using the macro:

```
boundaryCondition(periodic)
```

This expands to yield:

```
<Updater periodicBoundaryOnEntityghost>
  kind=periodicCartBc1d
  onGrid=domain
  in=[q]
  out=[q]
</Updater>
```

The periodic boundary condition updater (documented at refmanual-periodicCartBc) is again identical to that used for the Euler equations (described in *Advanced Methods for Solving the Euler Equations with USim*), for the same reason as discussed above for the *copy* boundary condition. The *copy* and *periodicCartBc* boundary conditions are the two easiest boundary conditions to apply in USim.

4.3.5 Advancing By A Time Step

In order to solve the MHD equations, we have to advance the conserved quantities from time t to $t + \Delta t$. This is done by applying a time integration scheme. In *Using USim to solve the Magnetohydrodynamic Equations*, we did this using the macro:

```
timeAdvance(TIME_ORDER)
```

This expands to yield:

```
<Updater mainIntegrator>
  kind=multiUpdater1d
  onGrid=domain
  in=[q]
  out=[qNew]

  <TimeIntegrator timeStepper>
    kind=rungeKutta1d
    onGrid=domain
```

```
    scheme=second
  </TimeIntegrator>


  <UpdateStep boundaryStep>
    operation=boundary
    updaters=[copyBoundaryOnEntityleft copyBoundaryOnEntityright periodicBoundaryOnEntityghost]
    syncVars=[q]
  </UpdateStep>


  <UpdateStep integrationStep>
    operation=integrate
    updaters=[hyper]
    syncVars=[qNew]
  </UpdateStep>


  <UpdateSequence sequence>
    startOnly=[]
    restoreOnly=[]
    writeOnly=[]
    loop=[boundaryStep   integrationStep]
  </UpdateSequence>

</Updater>
```

Note that this time integration scheme is **identical** to that described in *Advanced Methods for Solving the Euler Equations with USim*; in USim, we can evolve many different systems of hyperbolic equations using the same basic algorithmic ingredients.

4.3.6 Simulation Diagnostics

The macro-based simulations described in *Basic USim Simulations* compute a range of quantities that are output from the simulation to aid the user in understanding simulation behavior. For the MHD equations, these quantities include the *primitive* variables:

```
primitiveVariables = [density Velocity Pressure magneticField magneticFieldEnergy]
```

These quantities are computed in the simulation through use of a refmanual-updater-combiner:

```
<Updater computeDensity>
  kind=combiner1d
  onGrid=domain
  in=[q]
  out=[density]
  indVars_q=[rho rhou rhov rhow Er bx by bz psi]
  exprs=[ rho  ]
</Updater>

<Updater computeVelocity>
  kind=combiner1d
  onGrid=domain
  in=[q]
  out=[velocity]
  indVars_q=[rho rhou rhov rhow Er bx by bz psi]
  preExprs=[ vx=rhou/rho   vy=rhov/rho   vz=rhow/rho  ]
```

```
  exprs=[ vx   vy   vz  ]
</Updater>

<Updater computePressure>
  kind=combiner1d
  onGrid=domain
  in=[q]
  out=[pressure]
  gas_gamma=1.6666666666666667
  mu0=1.0
  indVars_q=[rho rhou rhov rhow Er bx by bz psi]
  preExprs=[ pr=(Er-0.5*(rhou*rhou+rhov*rhov+rhow*rhow)/rho-0.5*(bx*bx+by*by+bz*bz))*(gas_gamma-1.0)
  exprs=[ pr  ]
</Updater>

<Updater computeMagneticField>
  kind=combiner1d
  onGrid=domain
  in=[q]
  out=[magneticField]
  indVars_q=[rho rhou rhov rhow Er bx by bz psi]
  exprs=[ bx   by   bz  ]
</Updater>

<Updater computeFieldEnergy>
  kind=combiner1d
  onGrid=domain
  in=[magneticField]
  out=[magneticFieldEnergy]
  mu0=1.0
  indVars_magneticField=[bx by bz]
  exprs=[ (bx*bx+by*by+bz*bz)*(0.5/mu0)  ]
</Updater>
```

Each of these updaters follow the pattern described in *Writing Out Data*. For the MHD equations, USim also computes a set of variables that describe the properties of the magnetic field:

```
magneticFieldProperties = [magneticFieldDivergence magneticFieldCurrent]
```

These quantities are computed through the use of USim capabilties to compute derivatives of quantities defined on the simulation mesh (described at refmanual-vector):

```
<Updater computeFieldDivergence>
  kind=vector1d
  onGrid=domain
  in=[magneticField]
  out=[magneticFieldDivergence]
  numberOfInterpolationPoints=8
  derivative=divergence
</Updater>

<Updater computeFieldCurrent>
  kind=vector1d
  onGrid=domain
  in=[magneticField]
  out=[magneticFieldCurrent]
  numberOfInterpolationPoints=8
  derivative=curl
</Updater>
```

USim capabilties for computing first- and second-order derivatives are discussed in later tutorials.

4.3.7 Putting it all Together

In *Solving Multi-Dimensional Problems in USim*, we told USim that we're done specifying the simulation and that it can be run by calling the macro:

```
runFluidSimulation()
```

This macro collects up all of the updaters that we have added so far and, based on the sequence that we have added them, figures out how to call them to run a USim simulation. For the Brio & Wu shock case in *ShockTube* example, the computations performed by this macro results in the following refmanual-updateSteps and refmanual-updateSequence being generated:

```
<UpdateStep startStep>
  updaters=[setVar copyBoundaryOnEntityleft copyBoundaryOnEntityright periodicBoundaryOnEntityghost g
  syncVars=[q]
</UpdateStep>


<UpdateStep restoreStep>
  updaters=[]
  syncVars=[]
</UpdateStep>


<UpdateStep bcStep>
  updaters=[copyBoundaryOnEntityleft copyBoundaryOnEntityright periodicBoundaryOnEntityghost]
  syncVars=[q]
</UpdateStep>


<UpdateStep hyperStep>
  updaters=[getMaxWaveSpeed mainIntegrator copier]
  syncVars=[]
</UpdateStep>


<UpdateStep writeStep>
  updaters=[computePressure computeMagneticField computeVelocity computeDensity computeFieldEnergy co
  syncVars=[]
</UpdateStep>


<UpdateSequence simulation>
  startOnly=[startStep]
  restoreOnly=[restoreStep   bcStep]
  writeOnly=[bcStep   writeStep]
  loop=[hyperStep]
</UpdateSequence>
```

Note the similarlity between these blocks and those discussed in *Advanced Methods for Solving the Euler Equations with USim*. There are two major differences:

1. The *updaters* specified in the *hyperStep* now calls the *getMaxWaveSpeed* updater prior to the *mainIntegrator*; this computes the maximum wave speed for the simulation needed for evolving the MHD equations by Δt.
2. The *updaters* specified in the *writeStep* now includes all of the updaters needed to compute additional quantities

for the MHD equations. These updaters are only run when USim writes out data; helping to minimize their impact on USim performance.

An Example Simulation File

The input file for the problem *Shock Tube* in the USimBase package demonstrates each of the concepts described above to evolve the classic Brio & Shock tube problem in one-dimensional magnetohydrodynamics. You can view the actual input blocks to Ulixes in the *Setup* window by choosing *Save And Process Setup* and then clicking on the *shockTube.in* file. In the *.in* file all macros are expanded to produce input blocks.

4.3.8 Advanced USim Simulation Structure

In earlier tutorials, we developed a simple pattern that could be used to design USim simulations using macros. This pattern can be repeated when we don't use macros; however, we now need to add the grids, data structures, updaters, boundary conditions and time integration schemes by hand and then tell USim how to run them. An initial for the MHD equations looks like the following:

```
# Specify parameters for the specific physics problem
$ PARAM_1 = <value>
$ PARAM_2 = <value>
$ PARAM_N = <value>

# Initialize a USim simulation
# Simulation start and end times
tStart = <float>
tEnd = <float>
# Number of data files to write
numFrames = <integer>
# Initial time-step to use
initDt = <float>
# Level of feedback to give user
verbosity = <info/debug>

<Component fluids>
  kind = updaterComponent

   # Setup the grid
  <Grid Grid_Name (type=string)>
    <grid parameters>
  </Grid>

   # Create data structures needed for the simulation
  <DataStruct DataStruct_Name1 (type=string)>
    kind = nodalArray
    onGrid = <Grid_Name>
    numComponents = <int>
  </DataStruct>

  <DataStruct DataStruct_Name2 (type=string)>
    kind = nodalArray
    onGrid = <Grid_Name>
    numComponents = <int>
  </DataStruct>

   <DataStruct DataStruct_NameN (type=string)>
    kind = nodalArray
```

```
  onGrid = <Grid_Name>
  numComponents = <int>
</DataStruct>

# Specify initial condition
<Updater Initialization_Updater_Name (type=string)>
  kind = initialize<NDIM>d
  onGrid = <Grid_Name>
  out = <DataStruct_Name for t = 0>

  # initial condition to use
  <Function func>
    kind = exprFunc

    # Step 1: Add Variables
    VARIABLE_1 = <float>
    VARIABLE_2 = <float>
    VARIABLE_N = <float>


    # Step 2: Add Pre-Expressions
    preExprs = [ \
        "<PreExpression_1>", \
        "<PreExpression_2>", \
        "<PreExpression_N>"
        ]

   # Step 3: Add expressions specifying initial condition on density,
   # momentum, total energy
   addExpression(<expression>)

    exprs = ["<density_expression>",  \
                  "<xMomentum_expression>",  \
                  "<yMomentum_expression>",  \
                  "<zMomentum_expression>", \
                  "<totalEnergy_expression>", \
                  "<xMagneticField_expression>",  \
                  "<yMagneticField_expression>",  \
                  "<zMagneticField_expression>", \
                  "<scalarPotential_expression>"
                  ]

  </Function>

</Updater>

# Add the spatial discretization of the fluxes
<Updater FiniteVolume_Updater_Name (type=string)>
  kind=classicMuscl<NDIM>d
  onGrid=<Grid_Name>

  # input data-structures
  in=<DataStruct_Name for t^n>
   # output data-structures
  out=<DataStruct_Name for \nabla \cdot F(<DataStruct_Name for t^n>)>
  # CFL number to use
  cfl=<float>
  # legacy time integration scheme, attribute; should be set to "none"
```

```
  timeIntegrationScheme=none
  # Riemann solver
  numericalFlux=<string>
   # Limiter to use
  limiter=[<string>]
  # Form of variable to limit
  variableForm=<string>
  # Whether to check variables for physical validitiy
  preservePositivity=<int>

  # Hyperbolic equation system
  <Equation idealMhd>
    kind=mhdDednerEqn
    # Adiabatic index
    gasGamma=<float>
    # Permeability
    mu0=<float>
  </Equation>

   # Hyperbolic equation to solve
  equations=[idealMhd]
</Updater>

# Boundary conditions
<Updater BoundaryCondition_Name1 (type=string)>
  kind=<string><NDIM>d
  onGrid=<Grid_Name>
  in=<DataStruct_Name for t^n>
  out=<DataStruct_Name for t^n>
  entity=<Entity_Name (type=string)>
  <Boundary Condition Parameters>
</Updater>

# Time integration
<Updater TimeIntegrationUpdater_Name (type=string)>
  kind=multiUpdater<NDIM>d
  onGrid=<Grid_Name>
  in=<DataStruct_Name for t^n>
  out=<DataStruct_Name for t^n+1>

  <TimeIntegrator timeStepper>
    kind=rungeKutta<NDIM>d
    onGrid=<Grid_Name>
    scheme=<string>
  </TimeIntegrator>

  <UpdateStep boundaryStep>
    operation=boundary
    updaters=<string list of boundary conditions>
    syncVars=<DataStruct_Name at t^n>
  </UpdateStep>

  <UpdateStep integrationStep>
    operation=integrate
    updaters=<string list of integrators>
    syncVars=<DataStruct_Name for \nabla \cdot F(<DataStruct_Name for t^n>)>
  </UpdateStep>
```

```
  <UpdateSequence sequence>
    loop=[boundaryStep   integrationStep]
  </UpdateSequence>

</Updater>

<Updater CopierUpdater_Name (type=string)>
  kind=linearCombiner<NDIM>d
  onGrid=<Grid_Name>
  in=<DataStruct_Name for t^n+1>
  out=<DataStruct_Name for t^n>
  coeffs=[1.0]
</Updater>

# Output Diagnostics
<Updater OutputDiagnosticUpdater_Name1 (type=string)>
  kind=<string><NDIM>d
  onGrid=<Grid_Name>
  in=<DataStruct_Name1, DataStruct_Name2, ..., DataStruct_NameN>
  out=<OutputDiagnosticDataStruct_Name1>

  # Step 0: Specify components of input data structures
  indVars_DataStruct_Name1 = <DataStruct_Name1_Component1, DataStruct_Name1_Component2, ..., DataSt
  indVars_DataStruct_Name2 = <DataStruct_Name2_Component1, DataStruct_Name2_Component2, ..., DataSt
  indVars_DataStruct_NameN = <DataStruct_NameN_Component1, DataStruct_NameN_Component2, ..., DataSt

  # Step 1: Add Variables
  VARIABLE_1 = <float>
  VARIABLE_2 = <float>
  VARIABLE_N = <float>

  # Step 2: Add Pre-Expressions
  preExprs = [ \
        "<PreExpression_1>", \
        "<PreExpression_2>", \
        "<PreExpression_N>"
        ]

   # Step 3: Add expressions to compute output diagnostic
   exprs = [ \
        "<expression_1>", \
        "<expression_2>", \
        "<expression_N>"
        ]
</Updater>

<UpdateStep startStep>
  updaters=[Initialization_Updater_Name BoundaryCondition_Name1 ... BoundaryCondition_NameN]
  syncVars=[<DataStruct_Name for t^n>]
</UpdateStep>

<UpdateStep restoreStep>
  updaters=<String List of Updaters>
  syncVars=<String List of DataStructs>
</UpdateStep>

<UpdateStep bcStep>
  updaters=[BoundaryCondition_Name1 ... BoundaryCondition_NameN]
```

```
    syncVars=[<DataStruct_Name for t^n>]
  </UpdateStep>

  <UpdateStep hyperStep>
    updaters=[FiniteVolume_Updater_Name (type=string)]
    syncVars=<String List of DataStructs>
  </UpdateStep>

  <UpdateStep writeStep>
    updaters=< OutputDiagnosticUpdater_Name1,  OutputDiagnosticUpdater_Name2,  OutputDiagnosticUpdate
    syncVars=<OutputDiagnosticDataStruct_Name1, OutputDiagnosticDataStruct_Name2, ..., OutputDiagnost
  </UpdateStep>

  <UpdateSequence simulation>
    startOnly=[generateStep   startStep]
    restoreOnly=[generateStep   restoreStep   bcStep]
    writeOnly=[bcStep   writeStep]
    loop=[hyperStep]
  </UpdateSequence>

</Component>
```

Note that we have not filled in some of the entries in the *UpdateSteps* above. How to use these *UpdateSteps* is discussed in later tutorials.

4.4 Advanced Methods for Solving for Solving Problems in Multi-Dimensions with Usim

In *Advanced USim Simulation Concepts* we examined the basic ingredients of a USim input file: the simulation grid (see *Defining the Simulation Grid*); data structures (see *Allocating Memory*); how to assign initial conditions (see *Setting Initial Conditions*) and how to write out additional data (see *Writing Out Data*). In *Advanced Methods for Solving the Euler Equations with USim*, we built on these concepts and demonstrated the basic methods used by USim to solve the Euler equations without the use of macros. Next, in *Advanced Methods for Solving the Magnetohydrodynamics Equations with USim*, we extended these ideas to demonstrate how to solve the MHD equations in USim, again without the use of macros.

This tutorial is based on the quickstart-rtInstability example in USimBase, which demonstrates the well-known Rayleigh-Taylor instability problem described by:

```
Jun, Norman, & Stone, ApJ 453, 332 (1995).
```

Using this example, we will examine how to solve the equations of inviscid compressible hydrodynamics and ideal compressible magnetohydrodynamics directly without the use of macros.

Contents

4.4.1 Defining the Simulation Grid

In order to execute the two-dimensional Rayleigh-Taylor, we must define a two-dimensional grid simulation grid. In USim, this is done by replacing the *Grid* input block defined in *Defining the Simulation Grid* with:

```
<Grid domain>
  kind=cart2d
  lower=[-0.25  -0.75]
  upper=[0.25  0.75]
  cells=[64  192]
  periodicDirs=[0]
  ghostLayers=2
  isRadial=0
  writeGeom=0
  writeConn=0
  writeHalos=0
</Grid>
```

Notice that the *lower*, *upper* and *cells* input blocks now take two entries, one for each dimension. A three-dimensional Cartesian grid would therefore be defined by:

```
<Grid domain>
  kind=cart3d
  lower=[-0.25  -0.75  -0.25]
  upper=[0.25  0.75  0.25]
  cells=[64  192  64]
  periodicDirs=[0 2]
  ghostLayers=2
  isRadial=0
  writeGeom=0
  writeConn=0
  writeHalos=0
</Grid>
```

Note the usage of the *periodicDirs* attribute in these blocks. This attribute species the directions in the computational mesh that are periodic. For this example in two-dimensions, the *0* (or x) direction is periodic; similarly, in three-dimensions, both the *0* and *2* (z) directions are periodic. Note that this attribute is necessary to setup information within the simulation grid regarding periodicity; however, periodic boundary conditions must still be applied in the simulation. This is discussed below.

4.4.2 Allocating Simulation Memory

USim data structures automatically account for the chosen dimensionality of the simulation. As a result, there is no need to change how data structures are added for the Euler equations compared to the discussion in *Advanced Methods for Solving the Euler Equations with USim* or the MHD equations compared to the discussion in *Advanced Methods for Solving the Magnetohydrodynamics Equations with USim*.

4.4.3 Initializing the Fluid

The initial condition for a multi-dimensional simulation follows the same basic three-step pattern outlined previously; however, the *kind* attribute for the initialization updater has to match the dimension of the mesh:

```
<Updater setVar>
  kind=initialize2d
  onGrid=domain
  in=none
  out=[q   qNew]

  <Function initFunc>
    kind=exprFunc
    # adiabatic index
    gas_gamma=1.4
    # Upper fluid density
    rhoTop=2.0
    # Lower fluid density
    rhoBottom=1.0
    # Perturbation amplitude
    perturb=0.01
    # Location of upper y boundary
    ytop=0.75
    # acceleration due to gravity
    gravity=0.1
    # X-extent of domain
    lx=0.5
    # Y-extent of domain
    ly=1.5
    preExprs=[ p0=0.01   pert=0.01   pi=3.14159   rho=if(y<0.0,rhoBottom,rhoTop)   pr=(1.0/gas_gamma)
    exprs=[ rho   rho*vx   rho*vy   rho*vz   (pr/(gas_gamma-1.0))+0.5*rho*vy*vy  ]
  </Function>

</Updater>
```

As in *Solving Multi-Dimensional Problems in USim*, this initialization block creates a hydrostatic equilibrium consisting of a heavier fluid supported by a lighter fluid, which is perturbed with a single mode. Compared to the initialization block for the Sod shock tube problem discussed in *Advanced Methods for Solving the Euler Equations with USim*, the key difference as far as problem dimensionality is concerned is that the *kind* block is specified as *initialize2d* for this problem. This pattern is repeated for all updaters within a USim input file, e.g. for the generic updater *update*:

```
<Updater Updater_Name (type=string)>
  kind = updateKind<NDIM>d
  onGrid = <Grid_Name>
</Updater>
```

Here, *updateKind* is the kind of the USim updater that is being utilized (e.g. for the initialization input block, this is *initialize*) and *<NDIM>* is the dimensionality of the problem, e.g. *1,2,3* for one-, two- and three-dimensions.

4.4.4 Evolving the Fluid

USim implements the well-known MUSCL scheme to compute the spatial discretization of a hyperbolic conservation law (see *Using USim to solve the Euler Equations* for a more detailed description). In *Solving Multi-Dimensional Problems in USim*, we added this algorithm through:

```
#
# Add the spatial discretization of the fluxes
finiteVolumeScheme(DIFFUSIVE)

#
# Add source term for gravitational acceleration to the equations
addGravitationalAcceleration(GRAVITY_ACCEL)
```

For the quickstart-rtinstability example in two-dimensions, this macro expands to give:

```
<Updater hyper>
  kind=classicMuscl2d
  onGrid=domain
  in=[q]
  out=[qNew]
  cfl=0.4
  timeIntegrationScheme=none
  numericalFlux=hllcEulerFlux
  limiter=[muscl]
  variableForm=primitive
  preservePositivity=0

  <Equation euler>
    kind=eulerEqn
    gasGamma=1.4
  </Equation>

  equations=[euler]

  <Source gravity>
    kind=exprHyperSrc
    gravity=0.1
    rhoSrc=0.0
    mxSrc=0.0
    mzSrc=0.0
    inpRange=[0  1  2  3  4]
    outRange=[0  1  2  3  4]
    indVars=[rho rhou rhov rhow Er]
    exprs=[ rhoSrc   mxSrc   -rho*gravity   mzSrc   -gravity*rhov  ]
  </Source>

  sources=[gravity]
</Updater>
```

Compared to the quickstart-shocktube example discussed in *Advanced Methods for Solving the Euler Equations with USim*, we see that this is following the pattern discussed above, i.e. *kind* = *classicMuscl1d* has been replaced by *kind* = *classicMuscl2d*. We also have added an additional *Source* block to this updater, corresponding to the use of the macro:

```
# Add source term for gravitational acceleration to the equations
addGravitationalAcceleration(GRAVITY_ACCEL)
```

The use of this macro resulted in the addition of refmanual-exprHyperSrc to the refmanual-classicMuscl updater:

```
<Source gravity>
  kind=exprHyperSrc
  gravity=0.1
  rhoSrc=0.0
  mxSrc=0.0
  mzSrc=0.0
  inpRange=[0  1  2  3  4]
  outRange=[0  1  2  3  4]
  indVars=[rho rhou rhov rhow Er]
  exprs=[ rhoSrc   mxSrc   -rho*gravity   mzSrc   -gravity*rhov  ]
</Source>
```

Note that this block is following the classic three step pattern that used throughout USim (in this case, no preExpressions have been defined). The additional attributes *inpRange* and *outRange* tell USim which components of the input and output variables for the refmanual-classicMuscl updater to work with.

4.4.5 Applying Boundary Conditions

The Rayleigh-Taylor instability problem requires more advanced boundary conditions that were used for the Sod Shock tube in *the previous example*. In two-dimensions, appropriate boundary conditions for the Rayleigh-Taylor instability are periodic boundaries in the direction transverse to the gravitational field and reflecting boundaries in the parallel direction, which are the x and y directions respectively.

In the quickstart-rtinstability example, we applied these boundary conditions through the use of the macros:

```
#
# Boundary conditions
boundaryCondition(wall,top)
boundaryCondition(wall,bottom)
boundaryCondition(periodic)
```

The first two of these macros expand to yield the blocks:

```
<Updater wallBoundaryOnEntitytop>
  kind=eulerBc2d
  onGrid=domain
  in=[q]
  out=[q]
  model=eulerEqn
  bcType=wall
  entity=top
  gasGamma=1.4
</Updater>

<Updater wallBoundaryOnEntitybottom>
  kind=eulerBc2d
  onGrid=domain
  in=[q]
  out=[q]
  model=eulerEqn
  bcType=wall
  entity=bottom
  gasGamma=1.4
</Updater>
```

These reflecting wall boundary conditions are described below. The third macro expands to yield a periodic boundary condition:

```
<Updater periodicBoundaryOnEntityghost>
  kind=periodicCartBc2d
  onGrid=domain
  in=[q]
  out=[q]
</Updater>
```

We discuss this in the next section.

Periodic Boundary Conditions

Periodic boundary conditions in USim are specified in two ways, both of which *must* be used simultaneously. First, an input block *peroidicDirs* is added to the *Grid* block:

```
<Grid domain>
  kind = cart2d
  ghostLayers = 2
  lower = [-0.25, -0.75]
  upper = [  0.25,  0.75]
  cells = [64, 192]
  periodicDirs = [0]
</Grid>
```

The dimensions in which periodic boundary conditions are applied are specified by a vector. In this case, we apply periodic boundary conditions in the x direction, so we set:

```
periodicDirs = [0]
```

If we wished (for example) to extend the Rayleigh-Taylor instability problem to three-dimensions, then it might be appropriate to apply periodic boundary conditions to the x and z directions. In this case, we would set:

```
periodicDirs = [0,2]
```

In addition to the *periodicDirs* input block that is added to *Grid*, we must also add an updater block to apply the periodic boundary conditions within the simulation loop. This is done through the updater:

```
<Updater periodicBoundaryOnEntityghost>
  kind=periodicCartBc2d
  onGrid=domain
  in=[q]
  out=[q]
</Updater>
```

where the meanings of the various input blocks are described as in *Evolving the Fluid*. Note that if we wished to apply periodic boundary conditions in a three-dimensional problem, then we would set:

```
kind = periodicCartBc3d
```

This follows the pattern for changing the dimension of a USim simulation discussed previously.

Reflecting Wall Boundary Conditions

In the transverse direction to the flow (here the y direction), we specify reflecting wall boundary conditions. In USim, this is done using the updater:

```
<Updater wallBoundaryOnEntitytop>
  kind=eulerBc2d
  onGrid=domain
  in=[q]
  out=[q]
  model=eulerEqn
  bcType=wall
  entity=top
  gasGamma=1.4
</Updater>

<Updater wallBoundaryOnEntitybottom>
  kind=eulerBc2d
  onGrid=domain
  in=[q]
  out=[q]
  model=eulerEqn
  bcType=wall
  entity=bottom
  gasGamma=1.4
</Updater>
```

The refmanual-eulerBc applies boundary condition appropriate for systems of equations that follow the pattern of the Euler equations, i.e. a conserved state vector similar to that described at refmanual-eulerEqn. The specific type of hydrodynamic equation that the boundary condition is applied to is specified by the attribute *modelEqn*. Valid options for this attribute include refmanual-eulerEqn, refmanual-realGasEqn and refmanual-realGasEosEqn. Note that attributes required by these equations should be specified in the block associated with the boundary condition. In the examples above, this includes the *gasGamma* parameter required for refmanual-eulerEqn.

A range of boundary conditions can be applied by refmanual-eulerBc; the specific choice is specified by the attribute *bcType*. Valid options for this attribute include *wall*, *noInflow* and *noSlip*.

Finally, we must specify the boundary that the boundary condition is to be applied on. For a structured mesh such as used in the quickstart-rtinstability example, a range of boundaries are predefined:

`left` The lower x-boundary.

`right` The upper x-boundary.

`bottom` The lower y-boundary.

`top` The upper y-boundary.

`back` The lower z-boundary.

`front` The upper z-boundary.

We specify the boundary to apply the boundary through the use of the *entityType* attribute, e.g. to apply the wall boundary condition on the upper y-boundary, we set:

```
entity=top
```

4.4.6 Advancing By A Time Step

In order to solve the MHD equations, we have to advance the conserved quantities from time t to $t + \Delta t$. This is done by applying a time integration scheme. In *Solving Multi-Dimensional Problems in USim*, we did this using the macro:

```
timeAdvance(TIME_ORDER)
```

This expands to yield:

```
<Updater mainIntegrator>
  kind=multiUpdater2d
  onGrid=domain
  in=[q]
  out=[qNew]

  <TimeIntegrator timeStepper>
    kind=rungeKutta2d
    onGrid=domain
    scheme=second
  </TimeIntegrator>


  <UpdateStep boundaryStep>
    operation=boundary
    updaters=[wallBoundaryOnEntitytop wallBoundaryOnEntitybottom periodicBoundaryOnEntityghost]
    syncVars=[q]
  </UpdateStep>


  <UpdateStep integrationStep>
    operation=integrate
    updaters=[hyper]
    syncVars=[qNew]
  </UpdateStep>


  <UpdateSequence sequence>
    startOnly=[]
    restoreOnly=[]
    writeOnly=[]
    loop=[boundaryStep   integrationStep]
  </UpdateSequence>

</Updater>
```

This time integration scheme is very similar to that described in *Advanced Methods for Solving the Euler Equations with USim*; we follow the pattern described above to switch the dimension of the *mainIntegrator* and the *timeStepper* blocks. To handle the changes to the boundary conditions compared to *Advanced Methods for Solving the Euler Equations with USim*, we change the *boundaryStep* block from:

```
<UpdateStep boundaryStep>
  operation=boundary
  updaters=[copyBoundaryOnEntityleft copyBoundaryOnEntityright periodicBoundaryOnEntityghost]
  syncVars=[q]
</UpdateStep>
```

to:

```
<UpdateStep boundaryStep>
  operation=boundary
  updaters=[wallBoundaryOnEntitytop wallBoundaryOnEntitybottom periodicBoundaryOnEntityghost]
  syncVars=[q]
</UpdateStep>
```

4.4.7 Putting it all Together

In *Solving Multi-Dimensional Problems in USim*, we told USim that we're done specifying the simulation and that it can be run by calling the macro:

```
runFluidSimulation()
```

This macro collects up all of the updaters that we have added so far and, based on the sequence that we have added them, figures out how to call them to run a USim simulation. For the quickstart-rtInstability example, the computations performed by this macro results in the following refmanual-updateSteps and refmanual-updateSequence being generated:

```
<UpdateStep generateStep>
  updaters=[]
  syncVars=[]
</UpdateStep>


<UpdateStep startStep>
  updaters=[setVar wallBoundaryOnEntitytop wallBoundaryOnEntitybottom periodicBoundaryOnEntityghost]
  syncVars=[q]
</UpdateStep>


<UpdateStep restoreStep>
  updaters=[]
  syncVars=[]
</UpdateStep>


<UpdateStep bcStep>
  updaters=[wallBoundaryOnEntitytop wallBoundaryOnEntitybottom periodicBoundaryOnEntityghost]
  syncVars=[q]
</UpdateStep>


<UpdateStep hyperStep>
  updaters=[mainIntegrator copier]
  syncVars=[]
</UpdateStep>


<UpdateStep writeStep>
  updaters=[computePressure computeVelocity computeDensity]
  syncVars=[]
</UpdateStep>


<UpdateSequence simulation>
  startOnly=[generateStep   startStep]
  restoreOnly=[generateStep   restoreStep   bcStep]
  writeOnly=[bcStep   writeStep]
  loop=[hyperStep]
</UpdateSequence>
</UpdateSequence>
```

Note the similarlity between these blocks and those discussed in *Advanced Methods for Solving the Euler Equations with USim*. In fact, the only difference is the change in the *bcStep* block, which is analogous to that discussed above, i.e. we have that:

```
<UpdateStep bcStep>
  updaters=[copyBoundaryOnEntityleft copyBoundaryOnEntityright periodicBoundaryOnEntityghost]
  syncVars=[q]
</UpdateStep>
```

becomes:

```
<UpdateStep bcStep>
  updaters=[wallBoundaryOnEntitytop wallBoundaryOnEntitybottom periodicBoundaryOnEntityghost]
  syncVars=[q]
</UpdateStep>
```

4.4.8 Advanced USim Simulation Structure

In earlier tutorials, we developed a simple pattern that could be used to design USim simulations using macros. This pattern can be repeated when we don't use macros; however, we now need to add the grids, data structures, updaters, boundary conditions and time integration schemes by hand and then tell USim how to run them. In multi-dimensions, we can extend the pattern to:

```
# Specify parameters for the specific physics problem
$ PARAM_1 = <value>
$ PARAM_2 = <value>
$ PARAM_N = <value>

# Initialize a USim simulation
# Simulation start and end times
tStart = <float>
tEnd = <float>
# Number of data files to write
numFrames = <integer>
# Initial time-step to use
initDt = <float>
# Level of feedback to give user
verbosity = <info/debug>

<Component fluids>
  kind = updaterComponent

   # Setup the grid
  <Grid Grid_Name (type=string)>
    <grid parameters>
  </Grid>

   # Create data structures needed for the simulation
  <DataStruct DataStruct_Name1 (type=string)>
    kind = nodalArray
    onGrid = <Grid_Name>
    numComponents = <int>
  </DataStruct>

  <DataStruct DataStruct_Name2 (type=string)>
    kind = nodalArray
    onGrid = <Grid_Name>
    numComponents = <int>
  </DataStruct>

   <DataStruct DataStruct_NameN (type=string)>
```

```
  kind = nodalArray
  onGrid = <Grid_Name>
  numComponents = <int>
</DataStruct>

# Specify initial condition
<Updater Initialization_Updater_Name (type=string)>
  kind = initialize<NDIM>d
  onGrid = <Grid_Name>
  out = <DataStruct_Name for t = 0>

  # initial condition to use
  <Function func>
    kind = exprFunc

    # Step 1: Add Variables
    VARIABLE_1 = <float>
    VARIABLE_2 = <float>
    VARIABLE_N = <float>


    # Step 2: Add Pre-Expressions
    preExprs = [ \
        "<PreExpression_1>", \
        "<PreExpression_2>", \
        "<PreExpression_N>"
        ]

   # Step 3: Add expressions specifying initial condition on density,
   # momentum, total energy
   addExpression(<expression>)

    exprs = ["<density_expression>",  \
                  "<xMomentum_expression>",  \
                  "<yMomentum_expression>",  \
                  "<zMomentum_expression>", \
                  "<totalEnergy_expression>"]

  </Function>

</Updater>

# Add the spatial discretization of the fluxes
<Updater FiniteVolume_Updater_Name (type=string)>
  kind=classicMuscl<NDIM>d
  onGrid=<Grid_Name>

  # input data-structures
  in=<DataStruct_Name for t^n>
   # output data-structures
  out=<DataStruct_Name for \nabla \cdot F(<DataStruct_Name for t^n>)>
  # CFL number to use
  cfl=<float>
  # legacy time integration scheme, attribute; should be set to "none"
  timeIntegrationScheme=none
  # Riemann solver
  numericalFlux=<string>
   # Limiter to use
```

```
  limiter=[<string>]
  # Form of variable to limit
  variableForm=<string>
  # Whether to check variables for physical validitiy
  preservePositivity=<int>

  # Hyperbolic equation system
  <Equation euler>
    kind=eulerEqn
    # Adiabatic index
    gasGamma=<float>
  </Equation>

   # Hyperbolic equation to solve
  equations=[euler]

<Source Souce_Name_1>
  kind=<string>
  <Source_Name_1 params>
</Source>

...

<Source Souce_Name_N>
  kind=<string>
  <Source_Name_N params>
</Source>

sources=[Souce_Name_1, ..., Souce_Name_N]
</Updater>

# Boundary conditions
<Updater BoundaryCondition_Name1 (type=string)>
  kind=<string><NDIM>d
  onGrid=<Grid_Name>
  in=<DataStruct_Name for t^n>
  out=<DataStruct_Name for t^n>
  entity=<Entity_Name (type=string)>
  <Boundary Condition Parameters>
</Updater>

...

<Updater BoundaryCondition_NameN (type=string)>
  kind=<string><NDIM>d
  onGrid=<Grid_Name>
  in=<DataStruct_Name for t^n>
  out=<DataStruct_Name for t^n>
  entity=<Entity_Name (type=string)>
  <Boundary Condition Parameters>
</Updater>

# Output Diagnostics
<Updater OutputDiagnosticUpdater_Name1 (type=string)>
  kind=<string><NDIM>d
  onGrid=<Grid_Name>
  in=<DataStruct_Name1, DataStruct_Name2, ..., DataStruct_NameN>
  out=<OutputDiagnosticDataStruct_Name1>
```

```
  # Step 0: Specify components of input data structures
  indVars_DataStruct_Name1 = <DataStruct_Name1_Component1, DataStruct_Name1_Component2, ..., DataSt
  indVars_DataStruct_Name2 = <DataStruct_Name2_Component1, DataStruct_Name2_Component2, ..., DataSt
  indVars_DataStruct_NameN = <DataStruct_NameN_Component1, DataStruct_NameN_Component2, ..., DataSt

  # Step 1: Add Variables
  VARIABLE_1 = <float>
  VARIABLE_2 = <float>
  VARIABLE_N = <float>

  # Step 2: Add Pre-Expressions
  preExprs = [ \
        "<PreExpression_1>", \
        "<PreExpression_2>", \
        "<PreExpression_N>"
        ]

   # Step 3: Add expressions to compute output diagnostic
   exprs = [ \
        "<expression_1>", \
        "<expression_2>", \
        "<expression_N>"
        ]
</Updater>

# Time integration
<Updater TimeIntegrationUpdater_Name (type=string)>
  kind=multiUpdater<NDIM>d
  onGrid=<Grid_Name>
  in=<DataStruct_Name for t^n>
  out=<DataStruct_Name for t^n+1>

  <TimeIntegrator timeStepper>
    kind=rungeKutta<NDIM>d
    onGrid=<Grid_Name>
    scheme=<string>
  </TimeIntegrator>

  <UpdateStep boundaryStep>
    operation=boundary
    updaters=<string list of boundary conditions>
    syncVars=<DataStruct_Name at t^n>
  </UpdateStep>

  <UpdateStep integrationStep>
    operation=integrate
    updaters=<string list of integrators>
    syncVars=<DataStruct_Name for \nabla \cdot F(<DataStruct_Name for t^n>)>
  </UpdateStep>

  <UpdateSequence sequence>
    loop=[boundaryStep   integrationStep]
  </UpdateSequence>

</Updater>

<Updater CopierUpdater_Name (type=string)>
  kind=linearCombiner<NDIM>d
```

```
    onGrid=<Grid_Name>
    in=<DataStruct_Name for t^n+1>
    out=<DataStruct_Name for t^n>
    coeffs=[1.0]
  </Updater>

  <UpdateStep startStep>
    updaters=[Initialization_Updater_Name BoundaryCondition_Name1 ... BoundaryCondition_NameN]
    syncVars=[<DataStruct_Name for t^n>]
  </UpdateStep>

  <UpdateStep restoreStep>
    updaters=<String List of Updaters>
    syncVars=<String List of DataStructs>
  </UpdateStep>

  <UpdateStep bcStep>
    updaters=[BoundaryCondition_Name1 ... BoundaryCondition_NameN]
    syncVars=[<DataStruct_Name for t^n>]
  </UpdateStep>

  <UpdateStep hyperStep>
    updaters=[FiniteVolume_Updater_Name (type=string)]
    syncVars=<String List of DataStructs>
  </UpdateStep>

  <UpdateStep writeStep>
    updaters=<String List of Updaters>
    syncVars=<String List of DataStructs>
  </UpdateStep>

  <UpdateSequence simulation>
    startOnly=[generateStep   startStep]
    restoreOnly=[generateStep   restoreStep   bcStep]
    writeOnly=[bcStep   writeStep]
    loop=[hyperStep]
  </UpdateSequence>

</Component>
```

Note that we have not filled in some of the entries in the *UpdateSteps* above. How to use these *UpdateSteps* is discussed in later tutorials.

4.4.9 Advanced Methods for Solving the MHD Equations in Multi-Dimensions

We can extend the approach above to solving the MHD equations in multi-dimensions in a straightforward fashion. The steps for *Creating a Fluid Simulation* is unchanged from *Using USim to solve the Magnetohydrodynamic Equations*, while the steps for *Adding a Simulation Grid* are identical to that given above. The first differences come in *Initializing the Fluid*, where the initial condition now has to specify the magnetic field, magnetic energy and the scalar potential:

```
<Updater setVar>
  kind=initialize2d
  onGrid=domain
  in=none
  out=[q   qNew]

  <Function initFunc>
```

```
    kind=exprFunc
    # adiabatic index
    gas_gamma=1.4
    # Upper fluid density
    rhoTop=2.0
    # Lower fluid density
    rhoBottom=1.0
    # Perturbation amplitude
    perturb=0.01
    # Location of upper y boundary
    ytop=0.75
    # acceleration due to gravity
    gravity=0.1
    # X-extent of domain
    lx=0.5
    # Y-extent of domain
    ly=1.5
    # permeability of free space
    mu0=1.0
    # ratio of gas to magnetic pressure
    beta=1000.0
    preExprs=[ p0=0.01   pert=0.01   pi=3.14159   rho=if(y<0.0,rhoBottom,rhoTop)   pr=(1.0/gas_gamma)
    exprs=[ rho   rho*vx   rho*vy   rho*vz   pr/(gas_gamma-1.0)+0.5*rho*(vy*vy)+0.5*((bx*bx)/mu0)   b
  </Function>

</Updater>
```

The finite volume scheme used to compute the spatial discretization is becomes:

```
<Updater hyper>
  kind=classicMuscl2d
  onGrid=domain
  in=[q]
  out=[qNew]
  cfl=0.4
  timeIntegrationScheme=none
  numericalFlux=hlldFlux
  limiter=[muscl]
  variableForm=primitive
  preservePositivity=0
  waveSpeeds=[maxWaveSpeed]

  <Equation idealMhd>
    kind=mhdDednerEqn
    gasGamma=1.4
    mu0=1.0
  </Equation>

  equations=[idealMhd]

  <Source gravity>
    kind=exprHyperSrc
    gravity=0.1
    rhoSrc=0.0
    mxSrc=0.0
    mzSrc=0.0
    bxSrc=0.0
    bySrc=0.0
```

```
    bzSrc=0.0
    psiSrc=0.0
    inpRange=[0  1  2  3  4  5  6  7  8]
    outRange=[0  1  2  3  4  5  6  7  8]
    indVars=[rho rhou rhov rhow Er Bx By Bz Psi]
    exprs=[ rhoSrc   mxSrc   -rho*gravity   mzSrc   -gravity*rhov   bxSrc   bySrc   bzSrc   psiSrc
  </Source>

  sources=[gravity]
</Updater>
```

Compared to the exampled discussed in *Advanced Methods for Solving the Magnetohydrodynamics Equations with USim*, the differences follow the same pattern as discussed above; the switch of the dimension and the addition of the *Source* block. Note that for the MHD equations, we have to specify the source terms for the magnetic field and the potential, even though these are zero.

Note: The algorithms in USim automatically preserve $\nabla \cdot B = 0$ (the solenoidal constraint on the magnetic field), so there is no need for the user to make changes to the algorithm at the input file level when running in multi-dimensions.

Finally, the wall boundary conditions become:

```
<Updater conductingWallBoundaryOnEntitytop>
  kind=mhdBc2d
  onGrid=domain
  in=[q]
  out=[q]
  model=mhdDednerEqn
  bcType=conductingWall
  entity=top
  gasGamma=1.4
</Updater>


<Updater conductingWallBoundaryOnEntitybottom>
  kind=mhdBc2d
  onGrid=domain
  in=[q]
  out=[q]
  model=mhdDednerEqn
  bcType=conductingWall
  entity=bottom
  gasGamma=1.4
</Updater>
```

The signficant difference compared to the hydrodynamic case discussed above, is that we have switch the *kind* attribute to refmanual-mhdBc. Valid *bcType* options for this attribute include *conductingWall*, *noInflow* and *noSlip*, while the options for the *modelType* attribute include refmanual-mhdDednerEqn, refmanual-gasDynamicMhdDednerEqn, refmanual-simpleTwoTemperatureMhdDednerEqn and refmanual-twoTemperatureMhdDednerEqn.

Advanced USim Simulation Structure for MHD in Multi-Dimensions

In earlier tutorials, we developed a simple pattern that could be used to design USim simulations using macros. This pattern can be repeated when we don't use macros; however, we now need to add the grids, data structures, updaters, boundary conditions and time integration schemes by hand and then tell USim how to run them. A multi-dimensional approach for the MHD equations, include sources resembles the following:

```
# Specify parameters for the specific physics problem
$ PARAM_1 = <value>
$ PARAM_2 = <value>
$ PARAM_N = <value>

# Initialize a USim simulation
# Simulation start and end times
tStart = <float>
tEnd = <float>
# Number of data files to write
numFrames = <integer>
# Initial time-step to use
initDt = <float>
# Level of feedback to give user
verbosity = <info/debug>

<Component fluids>
  kind = updaterComponent

   # Setup the grid
  <Grid Grid_Name (type=string)>
    <grid parameters>
  </Grid>

   # Create data structures needed for the simulation
  <DataStruct DataStruct_Name1 (type=string)>
    kind = nodalArray
    onGrid = <Grid_Name>
    numComponents = <int>
  </DataStruct>

  <DataStruct DataStruct_Name2 (type=string)>
    kind = nodalArray
    onGrid = <Grid_Name>
    numComponents = <int>
  </DataStruct>

   <DataStruct DataStruct_NameN (type=string)>
    kind = nodalArray
    onGrid = <Grid_Name>
    numComponents = <int>
  </DataStruct>

  # Specify initial condition
  <Updater Initialization_Updater_Name (type=string)>
    kind = initialize<NDIM>d
    onGrid = <Grid_Name>
    out = <DataStruct_Name for t = 0>

    # initial condition to use
    <Function func>
      kind = exprFunc

      # Step 1: Add Variables
      VARIABLE_1 = <float>
      VARIABLE_2 = <float>
      VARIABLE_N = <float>
```

```
     # Step 2: Add Pre-Expressions
     preExprs = [ \
         "<PreExpression_1>", \
         "<PreExpression_2>", \
         "<PreExpression_N>"
         ]

    # Step 3: Add expressions specifying initial condition on density,
    # momentum, total energy
    addExpression(<expression>)

     exprs = ["<density_expression>",  \
                   "<xMomentum_expression>",  \
                   "<yMomentum_expression>",  \
                   "<zMomentum_expression>", \
                   "<totalEnergy_expression>", \
                   "<xMagneticField_expression>",  \
                   "<yMagneticField_expression>",  \
                   "<zMagneticField_expression>", \
                   "<scalarPotential_expression>"
                   ]

  </Function>

</Updater>

# Add the spatial discretization of the fluxes
<Updater FiniteVolume_Updater_Name (type=string)>
  kind=classicMuscl<NDIM>d
  onGrid=<Grid_Name>

  # input data-structures
  in=<DataStruct_Name for t^n>
   # output data-structures
  out=<DataStruct_Name for \nabla \cdot F(<DataStruct_Name for t^n>)>
  # CFL number to use
  cfl=<float>
  # legacy time integration scheme, attribute; should be set to "none"
  timeIntegrationScheme=none
  # Riemann solver
  numericalFlux=<string>
   # Limiter to use
  limiter=[<string>]
  # Form of variable to limit
  variableForm=<string>
  # Whether to check variables for physical validitiy
  preservePositivity=<int>

  # Hyperbolic equation system
  <Equation idealMhd>
    kind=mhdDednerEqn
    # Adiabatic index
    gasGamma=<float>
    # Permeability
    mu0=<float>
  </Equation>

   # Hyperbolic equation to solve
```

```
  equations=[idealMhd]

<Source Souce_Name_1>
  kind=<string>
  <Source_Name_1 params>
</Source>

...

<Source Souce_Name_N>
  kind=<string>
  <Source_Name_N params>
</Source>

sources=[Souce_Name_1, ..., Souce_Name_N]
</Updater>

# Boundary conditions
<Updater BoundaryCondition_Name1 (type=string)>
  kind=<string><NDIM>d
  onGrid=<Grid_Name>
  in=<DataStruct_Name for t^n>
  out=<DataStruct_Name for t^n>
  entity=<Entity_Name (type=string)>
  <Boundary Condition Parameters>
</Updater>

...

<Updater BoundaryCondition_NameN (type=string)>
  kind=<string><NDIM>d
  onGrid=<Grid_Name>
  in=<DataStruct_Name for t^n>
  out=<DataStruct_Name for t^n>
  entity=<Entity_Name (type=string)>
  <Boundary Condition Parameters>
</Updater>

# Time integration
<Updater TimeIntegrationUpdater_Name (type=string)>
  kind=multiUpdater<NDIM>d
  onGrid=<Grid_Name>
  in=<DataStruct_Name for t^n>
  out=<DataStruct_Name for t^n+1>

  <TimeIntegrator timeStepper>
    kind=rungeKutta<NDIM>d
    onGrid=<Grid_Name>
    scheme=<string>
  </TimeIntegrator>

  <UpdateStep boundaryStep>
    operation=boundary
    updaters=<string list of boundary conditions>
    syncVars=<DataStruct_Name at t^n>
  </UpdateStep>

  <UpdateStep integrationStep>
```

```
    operation=integrate
    updaters=<string list of integrators>
    syncVars=<DataStruct_Name for \nabla \cdot F(<DataStruct_Name for t^n>)>
  </UpdateStep>

  <UpdateSequence sequence>
    loop=[boundaryStep   integrationStep]
  </UpdateSequence>

</Updater>

<Updater CopierUpdater_Name (type=string)>
  kind=linearCombiner<NDIM>d
  onGrid=<Grid_Name>
  in=<DataStruct_Name for t^n+1>
  out=<DataStruct_Name for t^n>
  coeffs=[1.0]
</Updater>

# Output Diagnostics
<Updater OutputDiagnosticUpdater_Name1 (type=string)>
  kind=<string><NDIM>d
  onGrid=<Grid_Name>
  in=<DataStruct_Name1, DataStruct_Name2, ..., DataStruct_NameN>
  out=<OutputDiagnosticDataStruct_Name1>

  # Step 0: Specify components of input data structures
  indVars_DataStruct_Name1 = <DataStruct_Name1_Component1, DataStruct_Name1_Component2, ..., DataSt
  indVars_DataStruct_Name2 = <DataStruct_Name2_Component1, DataStruct_Name2_Component2, ..., DataSt
  indVars_DataStruct_NameN = <DataStruct_NameN_Component1, DataStruct_NameN_Component2, ..., DataSt

  # Step 1: Add Variables
  VARIABLE_1 = <float>
  VARIABLE_2 = <float>
  VARIABLE_N = <float>

  # Step 2: Add Pre-Expressions
  preExprs = [ \
        "<PreExpression_1>", \
        "<PreExpression_2>", \
        "<PreExpression_N>"
        ]

   # Step 3: Add expressions to compute output diagnostic
   exprs = [ \
        "<expression_1>", \
        "<expression_2>", \
        "<expression_N>"
        ]
</Updater>

<UpdateStep startStep>
  updaters=[Initialization_Updater_Name BoundaryCondition_Name1 ... BoundaryCondition_NameN]
  syncVars=[<DataStruct_Name for t^n>]
</UpdateStep>

<UpdateStep restoreStep>
  updaters=<String List of Updaters>
```

```
    syncVars=<String List of DataStructs>
  </UpdateStep>

  <UpdateStep bcStep>
    updaters=[BoundaryCondition_Name1 ... BoundaryCondition_NameN]
    syncVars=[<DataStruct_Name for t^n>]
  </UpdateStep>

  <UpdateStep hyperStep>
    updaters=[FiniteVolume_Updater_Name (type=string)]
    syncVars=<String List of DataStructs>
  </UpdateStep>

  <UpdateStep writeStep>
    updaters=< OutputDiagnosticUpdater_Name1,  OutputDiagnosticUpdater_Name2,  OutputDiagnosticUpdate
    syncVars=<OutputDiagnosticDataStruct_Name1, OutputDiagnosticDataStruct_Name2, ..., OutputDiagnost
  </UpdateStep>

  <UpdateSequence simulation>
    startOnly=[generateStep   startStep]
    restoreOnly=[generateStep   restoreStep   bcStep]
    writeOnly=[bcStep   writeStep]
    loop=[hyperStep]
  </UpdateSequence>

</Component>
```

Note that we have not filled in some of the entries in the *UpdateSteps* above. How to use these *UpdateSteps* is discussed in later tutorials.

4.5 Advanced Methods for Solving Problems on Advanced Meshes with USim

In *Advanced USim Simulation Concepts* we examined the basic ingredients of a USim input file: the simulation grid (see *Defining the Simulation Grid*); data structures (see *Allocating Memory*); how to assign initial conditions (see *Setting Initial Conditions*) and how to write out additional data (see *Writing Out Data*). In *Advanced Methods for Solving the Euler Equations with USim*, we built on these concepts and demonstrated the basic methods used by USim to solve the Euler equations without the use of macros. Next, in *Advanced Methods for Solving the Magnetohydrodynamics Equations with USim*, we extended these ideas to demonstrate how to solve the MHD equations in USim, again without the use of macros. Then, in *Advanced Methods for Solving for Solving Problems in Multi-Dimensions with Usim*, we demonstrated how to extend these concepts to multi-dimensions for both the Euler and MHD without the use of macros.

In this tutorial, we show how USim can be used to solve problems on more advanced meshes, building on the concepts presented in *Solving Problems on Advanced Structured Meshes in USim* and *Solving Problems on Unstructured Meshes in USim* to demonstrate how to solve problems in USim on a range of advanced mesh types.

Contents

4.5.1 Solving Problems in Axi-Symmetric Curvilinear Coordinates

An example of using USim to solve the MHD equations in axi-symmetric curvilinear coordinates is found in quickstart-zpinch.

Adding a Simulation Grid

In *Solving Problems on Advanced Structured Meshes in USim*, the use of axisymmetric cylindrical coordinates in this problem is specified through the use of the macro:

```
addCylindricalGrid(lowerBounds, upperBounds, numCells, periodicDirections)
```

For the quickstart-zpinch example, this macro expans to yield:

```
<Grid domain>
  kind=cart2d
  lower=[0.001  -0.1]
  upper=[0.1  0.1]
  cells=[32  64]
  periodicDirs=[1]
  ghostLayers=2
  isRadial=1
  writeGeom=0
  writeConn=0
  writeHalos=0
</Grid>
```

The *isRadial* attribute in this *Grid* block specifies that the grid is utilizing curvilinear coordinates. When we work directly with the input file, we have to manually specify which Grids and updaters we want to use curvilinear coordinates, unlike for the macros described in *Solving Problems on Advanced Structured Meshes in USim*.

Note: Cylindrical grids in USim are designed with **axisymmetric** coordinates in mind, so a one-dimensional mesh simulates the R coordinate, a two-dimensional mesh simulates the (R, Z) coordinates and a three-dimensional mesh simulates the (R, Z, ϕ) coordinates.

Evolving the Fluid

The MUSCL scheme implemented in USim requires additional source blocks to correctly integrate a hyperbolic conservation law in curvilinear coordinates. In *Solving Problems on Advanced Structured Meshes in USim*, these terms were added automatically when we used the *addCylindricalGrid* macro. When we work directly with the input file,

we have to add these by hand. This is done through the addition of a *Source* block to the refmanual-classicMuscl Updater in a similar fashion to the gravitational acceleration block discussed in *Advanced Methods for Solving for Solving Problems in Multi-Dimensions with Usim*:

```
<Updater hyper>
  kind=classicMuscl2d
  onGrid=domain
  in=[q]
  out=[qNew]
  cfl=0.4
  timeIntegrationScheme=none
  numericalFlux=hlldFlux
  limiter=[muscl]
  variableForm=primitive
  preservePositivity=1
  waveSpeeds=[maxWaveSpeed]

  <Equation idealMhd>
    kind=mhdDednerEqn
    basementPressure=20047.379936313715
    basementDensity=8.000000000000001e-06
    gasGamma=1.6667
    mu0=1.0
    correctNans=1
    correct=1
  </Equation>

  equations=[idealMhd]

  <Source axiSymmetricSource>
    kind=mhdSym
    gasGamma=1.6667
    mu0=1.0
    symmetryType=cylindrical
    model=mhdDednerEqn
  </Source>

  sources=[axiSymmetricSource]
</Updater>
```

The *axiSymmetricSource* applied here is documented at refmanual-mhdSym. This is the only changed needed for the refmanual-classicMuscl updater to work in axisymmetric cylindrical coordinates.

Simulation Diagnostics

The macro-based simulations described in *Basic USim Simulations* compute a range of quantities that are output from the simulation to aid the user in understanding simulation behavior. For the MHD equations, USim also computes a set of variables that describe the properties of the magnetic field:

```
magneticFieldProperties = [magneticFieldDivergence magneticFieldCurrent]
```

These quantities are computed through the use of USim capabilties to compute derivatives of quantities defined on the simulation mesh (described at refmanual-vector):

```
<Updater computeFieldDivergence>
  kind=vector2d
  onGrid=domain
  in=[magneticField]
```

```
  out=[magneticFieldDivergence]
  numberOfInterpolationPoints=8
  isRadial=1
  derivative=divergence
</Updater>


<Updater computeFieldCurrent>
  kind=vector2d
  onGrid=domain
  in=[magneticField]
  out=[magneticFieldCurrent]
  numberOfInterpolationPoints=8
  isRadial=1
  derivative=curl
</Updater>
```

Compared to the *Advanced Methods for Solving the Magnetohydrodynamics Equations with USim*, these blocks have an additional attribute, *isRadial=1*, which tells USim to calculate these derivatives in a form consistent with axisymmetric cylindrical coordinates.

4.5.2 Advanced Methods for Solving Problems on Body-Fitted Meshes in USim

An example of applying USim to a problem on a body-fitted mesh is found in quickstart-rampflow. This simulation uses a refmanual-bodyFitted grid, which is added to the simulation through the *addBodyFittedGrid* macro:

```
addBodyFittedGrid(lowerBounds, upperBounds, numCells, periodicDirections)
```

This macro expands to yield:

```
<Grid domain>
  kind=bodyFitted2d
  lower=[0.0  0.0  0.0]
  upper=[1.0  1.0  1.0]
  cells=[37  25  25]
  periodicDirs=[]
  ghostLayers=2
  isRadial=0
  writeGeom=0
  writeConn=0
  writeHalos=0

  <Vertices vertices>
    kind=funcVertCalc

    <Function myGrid>
      kind=exprFunc

      # Step 1: Add Variables
      xmin=0.1
      xmax=0.8
      ymax=0.45
      slope=0.17632698070846498
      inletCells=12.0
      rampCells=25.0
      yCells=25.0
      dxc=0.02702702702702703
```

```
      dyc=0.04

      # Step 2: Add Pre-Expressions
      preExprs=[ ix=rint(x/dxc)   iy=rint(y/dyc)   dxi=xmin/inletCells   dxr=(xmax-xmin)/rampCells

      # Step 3: Add Expressions
      exprs=[ xp   yp  ]
    </Function>

  </Vertices>

</Grid>
```

Notice how each of the three steps are added to the function block that defines the grid condition:

1. Step 1 creates a list of <*variablesName*> = <*value*> pairs; one per line.
2. Step 2 creates a string vector called *preExprs* (short for *preExpressions*). Each of the *addGridPreExpression* calls described in *Solving Problems on Advanced Structured Meshes in USim* adds one entry to this vector, which contains the function specified in the *addGridPreExpression* call.
3. Step 3 creates a string vector called *exprs* (short for *expressions*). As with Step 2, each of the *addGridExpression* call described in *Solving Problems on Advanced Structured Meshes in USim* adds one entry to this vector, which contains the function specified in the *addExpression* call. Note that the order of the *addGridExpression* calls is preserved in the string vector; this is critical as these entries specify each of the real space coordinates for the computational grid.

4.5.3 Advanced Methods for Solving Problems on Unstructured Meshes with USim

An example of applying USim to a problem on an unstructured mesh is found in quickstart-forwardFacingStep. In *Solving Problems on Unstructured Meshes in USim*, we added an unstructured mesh to a USim simulation through the use of one of the two macros:

```
addExodusGrid(GRIDFILE)
```

or:

```
addGmshGrid(GRIDFILE)
```

The choice of which block to use corresponds to the format of the mesh; either GMSH or ExodusII format. The *GRIDFILE* is the name of the file containing the mesh **without the file extension**.

These macros expand to yield a grid block that follows the pattern:

```
<Grid GRIDFILE>
  kind=unstructured
  ghostLayers=2
  isRadial=0
  writeGeom=0
  writeConn=0
  writeHalos=0
  decomposeHalos=0

  <Creator ctor>
    kind=<Mesh_Format>
    ndim=<NDIM>
    file=GRIDFILE.<Mesh_Extension>
  </Creator>
```

```
</Grid>
```

Here, <*Mesh_Format*> = *gmsh,exodus* and <*Mesh_Extension*> = *msh,g* correspondigly, while <*NDIM*> specifies the dimension of the simulation (as usual); here, it additionally corresponds to the highest dimensionality element in the mesh, i.e. for quads <*NDIM*> = *2*, while <*NDIM*> = *3* for hexes.

Note: Unstructured meshes in USim can only have dimension 2,3. One-dimensional meshes should make use of a cartesian or a body fitted mesh.

Advanced Methods for Boundary Conditions on Unstructured Meshes

As was discussed in *Solving Problems on Unstructured Meshes in USim*, the complexity of performing calculations on an unstructured mesh in USim is associated with application of boundary conditions. USim's approach to applying boundary conditions on an unstructured mesh is a two-step process:

1. The user defines the regions of the mesh (*entity*) that will be used to apply boundary conditions.
2. The user specifies the boundary conditions to apply on each region of the mesh.

For the specific case of quickstart-forwardFacingStep, there are three boundaries that we need to define, which are delimited by position in the streamwise direction, x:

1. Inflow boundary: defined for $x < 0.0$
2. Wall boundary: defined for $0.0 < x < 3.0$
3. Outflow boundary: defined for $x > 3.0$

In *Solving Problems on Unstructured Meshes in USim*, we generated *boundary* entities that exist on the exterior of the mesh using a combination of the *createNewEntityFromMask* and *addEntityMaskExpression* using the macros:

```
createNewEntityFromMask(<entityName>)
addEntityMaskExpression(<entityName>,<logicalExpression>)
```

These expand to yield a block that follows the pattern:

```
<Updater generate<entityName> >
  kind=entityGenerator<NDIM>d
  onGrid=<gridName>
  in=none
  out=none
  newEntityName=<entityName>
  onEntity=ghost

  <Function <entityName>Mask>
    kind=exprFunc
    exprs=[ <logicalExpression> ]
  </Function>

</Updater>
```

Of particular note here is the attribute *onEntity* = *ghost*. This attribute specifies an existing entity in the grid which is will be subdivided by <*logicalExpression*>. All USim grids include an entity named *ghost* which corresponds to the exterior surface of the mesh; as a result, our new entities only include mesh elements on the exterior surface of the mesh that obey <*logicalExpression*>. In this fashion, it is possible to construct complex boundary regions for unstructured meshes in USim using only simple logical conditions.

For the three entities needed for quickstart-forwardFacingStep, these operations take the form:

```
# Inflow for x < 0.0
<Updater generateinflowEntity>
  kind=entityGenerator2d
  onGrid=forwardFacingStep
  in=none
  out=none
  newEntityName=inflowEntity
  onEntity=ghost

  <Function inflowEntityMask>
    kind=exprFunc
    exprs=[ if(x<0.0,1.0,-1.0)  ]
  </Function>

</Updater>

# Wall for 0.0 < x < 3.0
<Updater generatewallEntity>
  kind=entityGenerator2d
  onGrid=forwardFacingStep
  in=none
  out=none
  newEntityName=wallEntity
  onEntity=ghost

  <Function wallEntityMask>
    kind=exprFunc
    exprs=[ if((x>0.0)and(x<3.0),1.0,-1.0)  ]
  </Function>

</Updater>

# Outflow for x > 3.0
<Updater generateoutflowEntity>
  kind=entityGenerator2d
  onGrid=forwardFacingStep
  in=none
  out=none
  newEntityName=outflowEntity
  onEntity=ghost

  <Function outflowEntityMask>
    kind=exprFunc
    exprs=[ if(x>3.0,1.0,-1.0)  ]
  </Function>

</Updater>
```

Now that we have created the *inflowEntity*, *wallEntity* and *outflowEntity*, we can specify boundary conditions on them. For the *wallEntity* and the *outflowEntity*, the boundary conditions are familiar from our previous tutorials:

```
boundaryCondition(wall,wallEntity)
boundaryCondition(copy,outflowEntity)
```

The first of these boundary condition macros expands to yield:

```
<Updater wallBoundaryOnEntitywallEntity>
  kind=eulerBc2d
  onGrid=forwardFacingStep
```

```
  in=[q]
  out=[q]
  model=eulerEqn
  bcType=wall
  entity=wallEntity
  gasGamma=1.4
</Updater>
```

while the second of these expands to yield:

```
<Updater copyBoundaryOnEntityoutflowEntity>
  kind=copy2d
  onGrid=forwardFacingStep
  in=[q]
  out=[q]
  entity=outflowEntity
</Updater>
```

These boundary conditions should be familiar from previous tutorials; the difference here being that the *entityName* parameter is set to match the specific entity that the boundary condition is applied to.

For the *inflowEntity*, we specify a new type of boundary condition *userSpecified* to determine the inflow properties using the macros:

```
boundaryCondition(<boundaryCondition>,<entityName>,)
addBoundaryConditionVariable(<boundaryCondition>,<entityName>,<variableName>,<variableValue>)
addBoundaryConditionPreExpression(<boundaryCondition>,<entityName>,<preExpression>)
addBoundaryConditionExpression(<boundaryCondition>,<entityName>,<Expression>)
```

For quickstart-forwardFacingStep, the inflow boundary was specified according to:

```
# Add a user specified boundary condition
boundaryCondition(userSpecified,inflowEntity)

# Step 1: Add Variables
addBoundaryConditionVariable(userSpecified,inflowEntity,gasGamma,GAS_GAMMA)

# Step 2: Add Pre-Expressions
addBoundaryConditionPreExpression(userSpecified,inflowEntity,rho = gasGamma)
addBoundaryConditionPreExpression(userSpecified,inflowEntity,vx = 3.0)
addBoundaryConditionPreExpression(userSpecified,inflowEntity,vy = 0.0)
addBoundaryConditionPreExpression(userSpecified,inflowEntity,vz = 0.0)
addBoundaryConditionPreExpression(userSpecified,inflowEntity,pr = 1.0)

# Step 3: Add expressions specifying boundary condition on density,
# momentum, total energy
addBoundaryConditionExpression(userSpecified,inflowEntity,rho)
addBoundaryConditionExpression(userSpecified,inflowEntity,rho*vx)
addBoundaryConditionExpression(userSpecified,inflowEntity,rho*vy)
addBoundaryConditionExpression(userSpecified,inflowEntity,rho*vz)
addBoundaryConditionExpression(userSpecified,inflowEntity,(pr/(gasGamma-1.0))+0.5*rho*(vx*vx))
```

These macros expand to yield the block:

```
<Updater userSpecifiedBoundaryOnEntityinflowEntity>
  kind=functionBc2d
  onGrid=forwardFacingStep
  in=[q]
  out=[q]
  entity=inflowEntity
```

```
  <Function userSpecifiedBoundaryOnEntityinflowEntityFunc>
    kind=exprFunc
    # Step 1: Add Variables
    gasGamma=1.4
    # Step 2: Add preExpressions
    preExprs=[ rho=gasGamma   vx=3.0   vy=0.0   vz=0.0   pr=1.0  ]
    # Step 3: Add expressions
    exprs=[ rho   rho*vx   rho*vy   rho*vz   (pr/(gasGamma-1.0))+0.5*rho*(vx*vx)  ]
  </Function>

</Updater>
```

This block follows the standard three step pattern:

1. Step 1 creates a list of <*variablesName*> = <*value*> pairs; one per line.
2. Step 2 creates a string vector called *preExprs* (short for *preExpressions*). Each of the *addBoundaryCondition-PreExpression* calls described in *Solving Problems on Unstructured Meshes in USim* adds one entry to this vector, which contains the function specified in the *addBoundaryConditionPreExpression* call.
3. Step 3 creates a string vector called *exprs* (short for *expressions*). As with Step 2, each of the *addBoundaryConditionExpression* call described in *Solving Problems on Unstructured Meshes in USim* adds one entry to this vector, which contains the function specified in the *addBoundaryConditionExpression* call. Note that the order of the *addBoundaryConditionExpression* calls is preserved in the string vector; this is critical as these entries specify each of the entries in the output data structure.

Putting it all Together

In *Solving Problems on Unstructured Meshes in USim*, we told USim that we're done specifying the simulation and that it can be run by calling the macro:

```
runFluidSimulation()
```

This macro collects up all of the updaters that we have added so far and, based on the sequence that we have added them, figures out how to call them to run a USim simulation. For the quickstart-forwardFacingStep example, the computations performed by this macro results in the following refmanual-updateSteps and refmanual-updateSequence being generated:

```
<UpdateStep generateStep>
  updaters=[generatewallEntity generateinflowEntity generateoutflowEntity]
  syncVars=[]
</UpdateStep>


<UpdateStep startStep>
  updaters=[setVar wallBoundaryOnEntitywallEntity userSpecifiedBoundaryOnEntityinflowEntity copyBound
  syncVars=[q]
</UpdateStep>


<UpdateStep restoreStep>
  updaters=[]
  syncVars=[]
</UpdateStep>


<UpdateStep bcStep>
```

```
  updaters=[wallBoundaryOnEntitywallEntity userSpecifiedBoundaryOnEntityinflowEntity copyBoundaryOnEn
  syncVars=[q]
</UpdateStep>


<UpdateStep hyperStep>
  updaters=[mainIntegrator copier]
  syncVars=[]
</UpdateStep>


<UpdateStep writeStep>
  updaters=[computePressure computeVelocity computeDensity computemachNumber]
  syncVars=[]
</UpdateStep>


<UpdateSequence simulation>
  startOnly=[generateStep   startStep]
  restoreOnly=[generateStep   restoreStep   bcStep]
  writeOnly=[bcStep   writeStep]
  loop=[hyperStep]
</UpdateSequence>
```

Notice that the *Updaters* that generate the *wall*, *inflow* and *outflow* entities are added to the the *generateStep*. This step is only called during the *startOnly* and *restoreOnly* operations in the *UpdateSequence*: an entity only needs to be created once per simulation startup.

Advanced USim Simulation Structure

In earlier tutorials, we developed a simple pattern that could be used to design USim simulations using macros. This pattern can be repeated when we don't use macros; however, we now need to add the grids, data structures, updaters, boundary conditions and time integration schemes by hand and then tell USim how to run them. For unstructured meshes, we can extend the pattern to:

```
# Specify parameters for the specific physics problem
$ PARAM_1 = <value>
$ PARAM_2 = <value>
$ PARAM_N = <value>

# Initialize a USim simulation
# Simulation start and end times
tStart = <float>
tEnd = <float>
# Number of data files to write
numFrames = <integer>
# Initial time-step to use
initDt = <float>
# Level of feedback to give user
verbosity = <info/debug>

<Component fluids>
  kind = updaterComponent

   # Setup the grid
  <Grid Grid_Name (type=string)>
    kind=unstructured
```

```
  <Grid Parameters>

  <Creator ctor>
    kind=<Mesh_Format>
    ndim=<NDIM>
    file=GRIDFILE.<Mesh_Extension>
  </Creator>
</Grid>

 # Create data structures needed for the simulation
<DataStruct DataStruct_Name1 (type=string)>
  kind = nodalArray
  onGrid = <Grid_Name>
  numComponents = <int>
</DataStruct>

<DataStruct DataStruct_Name2 (type=string)>
  kind = nodalArray
  onGrid = <Grid_Name>
  numComponents = <int>
</DataStruct>

 <DataStruct DataStruct_NameN (type=string)>
  kind = nodalArray
  onGrid = <Grid_Name>
  numComponents = <int>
</DataStruct>

<Updater generate<Entity_Name1> >
  kind=entityGenerator<NDIM>d
  onGrid=<Grid_Name>
  in=none
  out=none
  newEntityName=<Entity_Name1>
  onEntity=ghost

  <Function <Entity_Name1>Mask>
    kind=exprFunc
    exprs=[ <logicalExpression> ]
  </Function>
</Updater>

...

<Updater generate<Entity_NameN> >
  kind=entityGenerator<NDIM>d
  onGrid=<Grid_Name>
  in=none
  out=none
  newEntityName=<Entity_NameN>
  onEntity=ghost

  <Function <Entity_NameN>Mask>
    kind=exprFunc
    exprs=[ <logicalExpression> ]
  </Function>
</Updater>
```

```
# Specify initial condition
<Updater Initialization_Updater_Name (type=string)>
  kind = initialize<NDIM>d
  onGrid = <Grid_Name>
  out = <DataStruct_Name for t = 0>

  # initial condition to use
  <Function func>
    kind = exprFunc

    # Step 1: Add Variables
    VARIABLE_1 = <float>
    VARIABLE_2 = <float>
    VARIABLE_N = <float>


    # Step 2: Add Pre-Expressions
    preExprs = [ \
        "<PreExpression_1>", \
        "<PreExpression_2>", \
        "<PreExpression_N>"
        ]

   # Step 3: Add expressions specifying initial condition on density,
   # momentum, total energy
   addExpression(<expression>)

    exprs = ["<density_expression>",  \
                  "<xMomentum_expression>",  \
                  "<yMomentum_expression>",  \
                  "<zMomentum_expression>", \
                  "<totalEnergy_expression>"]

  </Function>

</Updater>

# Add the spatial discretization of the fluxes
<Updater FiniteVolume_Updater_Name (type=string)>
  kind=classicMuscl<NDIM>d
  onGrid=<Grid_Name>

  # input data-structures
  in=<DataStruct_Name for t^n>
   # output data-structures
  out=<DataStruct_Name for \nabla \cdot F(<DataStruct_Name for t^n>)>
  # CFL number to use
  cfl=<float>
  # legacy time integration scheme, attribute; should be set to "none"
  timeIntegrationScheme=none
  # Riemann solver
  numericalFlux=<string>
   # Limiter to use
  limiter=[<string>]
  # Form of variable to limit
  variableForm=<string>
  # Whether to check variables for physical validitiy
  preservePositivity=<int>
```

```
  # Hyperbolic equation system
  <Equation euler>
    kind=eulerEqn
    # Adiabatic index
    gasGamma=<float>
  </Equation>

   # Hyperbolic equation to solve
  equations=[euler]

<Source Souce_Name_1>
  kind=<string>
  <Source_Name_1 params>
</Source>

...

<Source Souce_Name_N>
  kind=<string>
  <Source_Name_N params>
</Source>

sources=[Souce_Name_1, ..., Souce_Name_N]
</Updater>

# Boundary conditions
<Updater BoundaryCondition_Name1 (type=string)>
  kind=<string><NDIM>d
  onGrid=<Grid_Name>
  in=<DataStruct_Name for t^n>
  out=<DataStruct_Name for t^n>
  entity=<Entity_Name (type=string)>
  <Boundary Condition Parameters>
</Updater>

...

<Updater BoundaryCondition_NameN (type=string)>
  kind=<string><NDIM>d
  onGrid=<Grid_Name>
  in=<DataStruct_Name for t^n>
  out=<DataStruct_Name for t^n>
  entity=<Entity_Name (type=string)>
  <Boundary Condition Parameters>
</Updater>

# Output Diagnostics
<Updater OutputDiagnosticUpdater_Name1 (type=string)>
  kind=<string><NDIM>d
  onGrid=<Grid_Name>
  in=<DataStruct_Name1, DataStruct_Name2, ..., DataStruct_NameN>
  out=<OutputDiagnosticDataStruct_Name1>

  # Step 0: Specify components of input data structures
  indVars_DataStruct_Name1 = <DataStruct_Name1_Component1, DataStruct_Name1_Component2, ..., DataSt
  indVars_DataStruct_Name2 = <DataStruct_Name2_Component1, DataStruct_Name2_Component2, ..., DataSt
  indVars_DataStruct_NameN = <DataStruct_NameN_Component1, DataStruct_NameN_Component2, ..., DataSt
```

```
  # Step 1: Add Variables
  VARIABLE_1 = <float>
  VARIABLE_2 = <float>
  VARIABLE_N = <float>

  # Step 2: Add Pre-Expressions
  preExprs = [ \
        "<PreExpression_1>", \
        "<PreExpression_2>", \
        "<PreExpression_N>"
        ]

   # Step 3: Add expressions to compute output diagnostic
   exprs = [ \
        "<expression_1>", \
        "<expression_2>", \
        "<expression_N>"
        ]
</Updater>

# Time integration
<Updater TimeIntegrationUpdater_Name (type=string)>
  kind=multiUpdater<NDIM>d
  onGrid=<Grid_Name>
  in=<DataStruct_Name for t^n>
  out=<DataStruct_Name for t^n+1>

  <TimeIntegrator timeStepper>
    kind=rungeKutta<NDIM>d
    onGrid=<Grid_Name>
    scheme=<string>
  </TimeIntegrator>

  <UpdateStep boundaryStep>
    operation=boundary
    updaters=<string list of boundary conditions>
    syncVars=<DataStruct_Name at t^n>
  </UpdateStep>

  <UpdateStep integrationStep>
    operation=integrate
    updaters=<string list of integrators>
    syncVars=<DataStruct_Name for \nabla \cdot F(<DataStruct_Name for t^n>)>
  </UpdateStep>

  <UpdateSequence sequence>
    loop=[boundaryStep   integrationStep]
  </UpdateSequence>

</Updater>

<Updater CopierUpdater_Name (type=string)>
  kind=linearCombiner<NDIM>d
  onGrid=<Grid_Name>
  in=<DataStruct_Name for t^n+1>
  out=<DataStruct_Name for t^n>
  coeffs=[1.0]
</Updater>
```

```
  <UpdateStep generateStep>
    updaters=[generate<Entity_Name1> ... generate<Entity_NameN>]
  </UpdateStep>

  <UpdateStep startStep>
    updaters=[Initialization_Updater_Name BoundaryCondition_Name1 ... BoundaryCondition_NameN]
    syncVars=[<DataStruct_Name for t^n>]
  </UpdateStep>

  <UpdateStep restoreStep>
    updaters=<String List of Updaters>
    syncVars=<String List of DataStructs>
  </UpdateStep>

  <UpdateStep bcStep>
    updaters=[BoundaryCondition_Name1 ... BoundaryCondition_NameN]
    syncVars=[<DataStruct_Name for t^n>]
  </UpdateStep>

  <UpdateStep hyperStep>
    updaters=[FiniteVolume_Updater_Name (type=string)]
    syncVars=<String List of DataStructs>
  </UpdateStep>

  <UpdateStep writeStep>
    updaters=<String List of Updaters>
    syncVars=<String List of DataStructs>
  </UpdateStep>

  <UpdateSequence simulation>
    startOnly=[generateStep   startStep]
    restoreOnly=[generateStep   restoreStep   bcStep]
    writeOnly=[bcStep   writeStep]
    loop=[hyperStep]
  </UpdateSequence>

</Component>
```

CHAPTER

FIVE

USING USIM TO SOLVE ADVANCED PHYSICS PROBLEMS

The following tutorials can be worked through with either a *USimHS* license or a *USimHEDP* license:

5.1 Using USim to Solve a Diffusion Problem

In this tutorial we show how to use USim to solve a problem with diffusion using the updater refmanual-diffusion with *derivative* set to diffusion.

Contents

5.1.1 Required DataStructs

The variable that is diffusing is *q* and is a scalar

```
<DataStruct q>
  kind = nodalArray
  onGrid = domain
  numComponents = 3
  writeOut = 1
</DataStruct>
```

In addition we define the scalar diffusion coefficient as *diffCoeff*

```
<DataStruct diffCoeff>
  kind = nodalArray
  onGrid = domain
  numComponents = 1
  writeOut = 1
</DataStruct>
```

5.1.2 Solving problems using the derivatives with option *diffusion*

diffCoeff can be defined using a refmanual-updater-combiner. After *diffCoeff* is computed the diffusion operator can be applied. We use refmanual-diffusion with *derivative* set to diffusion to compute the diffusion term. The operator

take in *q* and *diffCoeff* and the result (in this case) is stored in *qnew*

```
<Updater computeDiffusion>
  kind = diffusion2d
  onGrid = domain
  derivative = diffusion
  numScalars = 1
  coefficient = 1.0
  numberOfInterpolationPoints = 8

  in = [q,diffCoeff]
  out = [qnew]
</Updater>
```

Note that the complete list of options available in refmanual-diffusion are diffusion, anisotropicDiffusion and gradientOfDivergence.

5.1.3 Computing the time step for the diffusion operator

Time integration is performed using super time stepping. Super time stepping is a variable stage Runge-Kutta approach that is much faster (by the number of stages) than standard Runge-Kutta methods for solving diffusion problems. The approach requires two time steps. The desired (actual) time step is computed using a refmanual-timeStepRestrictionUpdater. The key here is that *STS_CFL=40.0* so it is much higher than the explicitly stable time step for a diffusive system

```
<Updater timeStepRestriction>

 kind = timeStepRestrictionUpdater2d
 in = [diffCoeff]

 onGrid = domain
 restrictions = [quadratic]

 <TimeStepRestriction quadratic>
   kind = quadratic
   cfl = STS_CFL
 </TimeStepRestriction>

</Updater>
```

Next the time step for the super time stepping method is computed where an explicitly stable *CFL=0.25* is used. The time step is stored in *stsDt*

```
<Updater getSTSdt>
 kind = getTimeStepUpdater2d
 in = [diffCoeff]
 out = [stsDt]
 onGrid = domain
 restrictions = [quadratic]

 <TimeStepRestriction quadratic>
   kind = quadratic
   cfl = EXPLICIT_CFL
 </TimeStepRestriction>
</Updater>
```

The Super Time Stepping integrator then knows to take the ratio of the desired time step and the explicitly stable time step to compute the number of stages used in the STS Updater.

5.1.4 An Example Simulation

The input file for the problem *Diffusion* in the USimHS package demonstrates each of the concepts described above.

5.2 Using USim to Solve the Two-Fluid Plasma Model

In the USimBase tutorials the basic concepts of USim were described. In this first HEDP tutorial we describe how to solve the fully electromagnetic two-fluid plasma equations using a semi-implicit operator to step over the plasma frequency and cyclotron frequency and electric field diffusion to minimize errors in in the electric field divergence relation.

Contents

5.2.1 Semi-Implicit Solution for the Lorentz Forces and Current Sources

To demonstrate how to use USim to solve a problem using the two-fluid plasma system, we will use the well known GEM (geomagnetic environmental modeling) reconnection challenge setup to solve fast reconnection of a current layer. The GEM challenge was originally described in

```
Birn, J., et al. "Geospace Environmental Modeling (GEM) magnetic reconnection challenge."
Journal of Geophysical Research: Space Physics (1978–2012) 106.A3 (2001): 3715-3719.
```

This tutorial is based on the *GEM Challenge* template.

The first thing we need to model two-fluids, is data structures for the electrons, ions and the electromagnetic field. In this case the electrons will be represented by a 5-moment compressible fluid:

```
<DataStruct electrons>
  kind = nodalArray
  onGrid = domain
  numComponents = 5
</DataStruct>
```

The ions are also represented by a 5-moment compressible fluid:

```
<DataStruct ions>
  kind = nodalArray
  onGrid = domain
  numComponents = 5
</DataStruct>
```

The electromagnetic field variable contains the full Field vector [Ex,Ey,Ez,Bx,By,Bz,Ep,Bp] with the variables Ep and Bp the error correction potentials. As such the electromagnetic field data structure is defined as:

```
<DataStruct em>
  kind = nodalArray
  onGrid = domain
```

```
  numComponents = 8
</DataStruct>
```

Separate initialization using an refmanual-initialize updater is performed for each variable, electrons, ions and em:

```
<Updater initElectrons>
  kind = initialize2d
  onGrid = domain
  out = [electrons]

  <Function func>
    kind = exprFunc

   .
   .
   .

    preExprs = [ \
        "rhoe = n0*me*(1.0/(cosh(y/lambda)*cosh(y/lambda))+0.2)", \
    "mze = -(me/qe)*b0*(1.0/lambda)*1.0/(cosh(y/lambda)*cosh(y/lambda))", \
    "ee = (1.0/12.0)*(1./(gamma-1.0))*b0*b0*(rhoe/me)+0.5*(mze*mze/rhoe)"]

    exprs = [ \
        "rhoe", "0.0", "0.0", "mze", "ee"]

  </Function>

</Updater>

<Updater initIons>
  kind = initialize2d
  onGrid = domain
  out = [ions]

  <Function func>
    kind = exprFunc

    .
    .
    .

    preExprs = [ \
        "rhoi = n0*mi*(1.0/(cosh(y/lambda)*cosh(y/lambda))+0.2)", \
    "ei = (5.0/12.0)*(1.0/(gamma-1.0))*b0*b0*(rhoi/mi)"]

    exprs = ["rhoi", "0.0", "0.0", "0.0", "ei"]

  </Function>

</Updater>

<Updater initEm>
  kind = initialize2d
  onGrid = domain
  out = [em]

  <Function func>
    kind = exprFunc
```

```
    .
    .
    .

    preExprs = ["bx = b0*tanh(y/lambda)-psi*(pi/Ly)*cos(2.0*pi*x/Lx)*sin(pi*y/Ly)", \
          "by = psi*(2.0*pi/Lx)*sin(2.0*pi*x/Lx)*cos(pi*y/Ly)"]

    exprs = ["0.0", "0.0", "0.0", "bx", "by", "0.0", "0.0", "0.0"]

  </Function>

</Updater>
```

In the above initialization some variables have been eliminated for conciseness. In addition, every variable, electrons, ions, em, must have their own refmanual-classicMuscl

```
<Updater hyperElectrons>
  kind = classicMuscl2d
  onGrid = domain
  timeIntegrationScheme = none

  numericalFlux = FLUID_NUMERICAL_FLUX
  preservePositivity = true
  limiter = [LIMITER, none]
  limiterType = characteristic
  variableForm = conservative

  in = [electrons]
  out = [electronsNew]

  cfl = CFL
  equations = [euler]

  <Equation euler>
    kind = eulerEqn
    gasGamma = GAS_GAMMA
    basementDensity = BASEMENT_DENSITY
    basementPressure = BASEMENT_PRESSURE
  </Equation>
</Updater>

<Updater hyperIons>
  kind = classicMuscl2d
  onGrid = domain
  timeIntegrationScheme = none
  numericalFlux = FLUID_NUMERICAL_FLUX
  preservePositivity = true
  limiter = [LIMITER]
  limiterType = characteristic
  variableForm = conservative

  in = [ions]
  out = [ionsNew]

  cfl = CFL
  equations = [euler]

  <Equation euler>
    kind = eulerEqn
```

```
    basementDensity = BASEMENT_DENSITY
    basementPressure = BASEMENT_PRESSURE
    gasGamma = GAS_GAMMA
  </Equation>
</Updater>

<Updater hyperEm>
  kind = classicMuscl2d
  onGrid = domain
  timeIntegrationScheme = none
  numericalFlux = fWaveFlux
  limiterType = characteristic
  limiter = [LIMITER]
  variableForm = conservative

  in = [em]
  out = [emNew]

  cfl = CFL
  equations = [maxwell]

  <Equation maxwell>
    kind = maxwellEqn
    c0 = SPEED_OF_LIGHT
    gamma = BP
    chi = 0.0
  </Equation>
</Updater>
```

The coupling between the fields and fluids is provided by Lorentz forces (for the fluid equations) and current sources (for the electromagnetic field). One option is to simply add these to the right hand side of the flux calculation and then integrate, however, this leads to an explicit scheme where the plasma frequency and cyclotron frequency must be resolved. Instead we use a semi-implicit operator as defined in

Harish Kumar and Siddhartha Mishra. "Entropy Stable Numerical Schemes for Two-Fluid Plasma Equations." Journal of Scientific Computing (2012): 1-25.

The implicit operator refmanual-twoFluidSrc is a source in USim, it's applied cell by cell and does not require a global implicit solve. The refmanual-twoFluidSrc is evaluated using the explicit solution to the electron, ion and em variables and the resulting matrix is multiplied by those same variables to produce the implicit source evaluation with explicit hyperbolic terms. The semi-implicit operator is given below:

```
<Updater twoFluidLorentz>
   kind = equation2d

   onGrid = domain
   in =  [electronsNew, ionsNew, emNew]
   out = [electronsNew, ionsNew, emNew]

  <Equation twofluidLorentz>
    kind = twoFluidSrc
    type = 5Moment
    electronCharge = ELECTRON_CHARGE
    electronMass = ELECTRON_MASS
    ionCharge = ION_CHARGE
    ionMass = ION_MASS
    epsilon0 = EPSILON0
  </Equation>
</Updater>
```

The semi-implicit operator is applied in a special location in the refmanual-multiUpdater. The list of *updaters* in the refmanual-multiUpdater define the explicit steps in the refmanual-multiUpdater. A second list of updaters is in the *operator* list. Updaters in the *operator* list are applied to integrationVariablesOut after a complete update. In the block below *operators = [twoFluidLorentz]* applies the operator after the explicit right hand side has been calculated, for example

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.

$$Q^{n+1} = Q^n + \Delta t \nabla f^n + \Delta t \psi^{n+1} \tag{5.0}$$

and therefore

$$Q^{n+1} = \left(1 - \Delta t A^{n+1}\right)^{-1} \Delta t \nabla f^n \tag{5.0}$$

The term in parentheses is the *twoFluidLorentz* operator defined in the *multiUpdater* below. As stated previously, the updaters in *operate* are applied to the right hand side which is computed in the *updaters* list

```
 <Updater rkUpdaterMain>
  kind = multiUpdater2d
  onGrid = domain

  in = [em, ions, electrons]
  out = [emNew, ionsNew, electronsNew]

  <TimeIntegrator rkIntegrator>
    kind = rungeKutta2d
    ongrid = domain
    scheme = RK_SCHEME
  </TimeIntegrator>

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

  <UpdateStep boundaries>
     updaters = [periodicEm, periodicIons, periodicElectrons, electronBcTop, electronBcBottom, \
      ionBcTop, ionBcBottom, emBcTop, emBcBottom]
  </UpdateStep>

  <UpdateStep hyper>
     operation = "integrate"
     updaters = [hyperIons, hyperElectrons, hyperEm, addSource]
  </UpdateStep>

  <UpdateStep implicit>
     operation = "operate"
     updaters = [twoFluidLorentz]
  </UpdateStep>

</Updater>
```

In the refmanual-multiUpdater above we include 3 sets of *in* variables, 3 sets of *out* variables for each of the integration variables.

The results described above are sufficient to create an algorithm that steps over plasma frequency and cyclotron frequency, but this does not show us how to minimze errors in the divergence equations.

5.2.2 Electric and Magnetic Field Divergence Cleaning

The standard approach to divergence preservation in USim is to use hyperbolic divergence cleaning. Hyperbolic divergence cleaning is described for the MHD equation in

Andreas Dedner, et al. "Hyperbolic divergence cleaning for the MHD equations." Journal of Computational Physics 175.2 (2002): 645-673.

And for Maxwell's equation in

Munz, C-D., et al. "Divergence correction techniques for Maxwell solvers based on a hyperbolic model." Journal of Computational Physics 161.2 (2000): 484-511.

Unfortunately, for the two-fluid system hyperbolic cleaning of the electric field is often inadequate, or results in larger errors than we started with. Instead we use electric field diffusion. In this section we describe how we perform the divergence cleaning in the GEM challenge problem.

First off, the magnetic field can be cleaned using the hyperbolic approach. In the hyperbolic updated of the electromagnetic field the refmanual-Equation block defines the speed of light *c0* as well as the wave speeds of the correction potentials. *gamma* is the magnetic field correction potential factor and *chi* is the electric field correction factor. The speed of the magnetic field correction potential is *gamma*c0* and of the electric field correction potential *chi*c0*. In the case below we give *gamma* a finite value (typically 1.0) and we set *chi* to 0 so that we can use a different correction method for the electric field:

```
<Updater hyperEm>
  kind = classicMuscl2d
  onGrid = domain
  timeIntegrationScheme = none
  numericalFlux = fWaveFlux
  limiterType = characteristic
  limiter = [LIMITER]
  variableForm = conservative

  in = [em]
  out = [emNew]

  cfl = CFL
  equations = [maxwell]

  <Equation maxwell>
    kind = maxwellEqn
    c0 = SPEED_OF_LIGHT
    gamma = BP
    chi = 0.0
  </Equation>
</Updater>
```

Electric field diffusion is quite simple. We add a diffusion term to the electric field given as

$$\frac{\partial E}{\partial t} - c^2 \nabla \times B = -\frac{J}{\epsilon_0} + \gamma \nabla \left(\nabla \cdot E - \frac{\rho_c}{\epsilon_0} \right) \tag{5.0}$$

where γ is the electric field diffusion coefficient. You can see that the diffusion term never kicks in unless there is a numerical error gauss' law. How do we go about implementing this in USim? First of all we define a set of data structures just for electric field divergence cleaning

The first data structure stores the characteristic cell length for the diffusion coefficient:

```
<DataStruct cellDx>
   kind = nodalArray
```

```
   onGrid = domain
   numComponents = 1
   writeOut = 0
 </DataStruct>
```

We define a data structure for storing the error computed from $\nabla \cdot E - \frac{\rho_c}{\epsilon_0}$:

```
<DataStruct residual>
  kind = nodalArray
  onGrid = domain
  numComponents = 1
  writeOut = 0
</DataStruct>
```

We define a data structure for storing the divergence of E:

```
<DataStruct divE>
  kind = nodalArray
  onGrid = domain
  numComponents = 1
  writeOut = 0
</DataStruct>
```

We then store the actual diffusion term in the last data structure:

```
<DataStruct gradDiv>
  kind = nodalArray
  onGrid = domain
  numComponents = 3
  writeOut = 0
</DataStruct>
```

Along with the data structures, we have a series of updaters that are used to fill up the data structures. The first updater refmanual-characteristicCellLength simply computes a characteristic length for each cell. This updater only needs to be called at startup since the cell length does not change.

```
<Updater computeCellDx>
  kind = characteristicCellLength2d
  onGrid = domain
  out = [cellDx]
  coefficient = 1.0
</Updater>
```

The next updater computes the $\nabla \cdot E$ from the electric field. It takes as in the array *em*. The vectorDivergence operator assumes the 3 vector of interest occurs in the 3 components of *em* which happens to be correct in this case as those components correspond to *Ex*,'Ey','Ez'. In addition, a parameter *coeffs* is provided which multiplies the resulting divergence by the factor 1.0. This is a simple way to reverse the sign of the divergence or multiply by some other factor:

```
<Updater computeDivE>
  kind = vectorDivergence2d
  onGrid = domain
  in = [em]
  out = [divE]
  coeffs = [1.0]
</Updater>
```

The next updater uses an refmanual-equation to compute the residual $\nabla \cdot E - \rho_c/\epsilon_0$. The source *computeChargeError* expects as input $\nabla \cdot E$ and then expectes the remaining variables to be fluid mass densities, any number of species can

be added. Along with the species mass densities we provide a list indicating the species mass and species charge for each of these species densities. We also specify the permittivity *epsilon0* so that the residual can be calculated.

```
<Updater computeResidual>
  kind = equation2d
  onGrid = domain

  in = [divE, electrons, ions]
  out = [residual]

  <Equation>
    kind = computeChargeError
    speciesCharge = [ELECTRON_CHARGE, ION_CHARGE]
    speciesMass = [ELECTRON_MASS, ION_MASS]
    epsilon0 = EPSILON0
  </Equation>

</Updater>
```

The final diffusion term including *gamma* factor is computed using a scalar gradient calculator. The scalar gradient takes 2 inputs. It takes the gradient of the first input (*residual* in this case) and multiplies that gradient by the second input *cellDx*. The result is the multiplied by *coefficient* which is constant for all space. This particular form of diffusion is such that the diffusion is stable to explicit time stepping. If super time stepping is used or subcycling, the diffusion coefficient can be increased to do a better job of error cleaning.

```
<Updater gradient>
  kind = scalarGradient2d
  gradientType = leastSquares
  onGrid = domain
  in = [residual, cellDx]
  out = [gradDiv]
  coefficient = 0.5
</Updater>
```

Once the diffusion term is computed it needs to be added to the right hand side of the update equation. The term is added after the hyperbolic explicit terms are computed. We use a *combiner2d* to add the diffusion term to the equation. In this example the updater takes two input data structures, *emNew* and *gradDiv* and dumps the output into *emNew*. Every input variable requires an *indVars_inputName* block which provides a way to access each component of the input variable. These names can then be used in the output expression *exprs*. In the refmanual-multiUpdater, the *addSource* block is called after the hyperbolic terms so that it is not overwritten by updaters that are called earlier.

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

  in = [emNew, gradDiv]
  out = [emNew]

  indVars_emNew = ["ex","ey","ez","bx","by","bz","phiE","phiB"]
  indVars_gradDiv = ["dx","dy","dz"]
  c = SPEED_OF_LIGHT
  exprs = ["ex+c*dx","ey+c*dy","ez+c*dz","bx","by","bz","phiE","phiB"]
</Updater>
```

Finally, boundary conditions must be provided to the residual. In this problem the residual on the boundary should be 0 so we use a refmanual-functionBc to explicitly set the residual on the boundary to zero. The variable *out* specifies the data that the boundary condition will be applied to, while *entity* tells the boundary condition which boundary it should be applied to. In this case *entity=ghost* means that the boundary is applied to all boundary cells.

```
<Updater residualBc>
  kind = functionBc2d

  onGrid = domain

  <Function func>
    kind = exprFunc
    exprs = ["0.0"]
  </Function>

  out = [residual]
  entity = ghost
</Updater>
```

This simulation has periodic boundaries in the x direction so we also apply a periodic boundary condition to the residual. This boundary condition is called after *residualBc* so it overwrites the boundry conditions in the X direction set by *residualBc*

```
<Updater periodicResidual>
  kind = periodicCartBc2d
  onGrid = domain
  in = [residual]
  out = [residual]
</Updater>
```

Now that we've added a bunch of new updaters we need to modify the multiUpdater to include the changes that have been made. The new updater added to the *updater* list include *computeDivE*, *computeResidual*, *residualBc*,'periodicResidual','gradient' in the *UpdateStep*'s *compute* and *clean*. These updaters are evaluated after the boundary conditions for *electrons*,'ions' and *em* are applied, but before the hyperbolic updaters are called *hyperIons*, *hyperElectrons* and *hyperEm*. Once again, for parallel simulations it's very important to get the synchronization correct. We've added one synchronizations. The new synchronization occurs directly after *periodicResidual*. *periodicResidual* is the last updater that is applied before a derivative *gradient* is computed on *residual* so we need to synchronize *residual* at this point.

```
<Updater rkUpdaterMain>
  kind = multiUpdater2d
  onGrid = domain

  in = [em, ions, electrons]
  out = [emNew, ionsNew, electronsNew]

  <TimeIntegrator rkIntegrator>
    kind = rungeKutta2d
    ongrid = domain
    scheme = RK_SCHEME
  </TimeIntegrator>

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

  <UpdateStep boundaries>
     updaters = [periodicEm, periodicIons, periodicElectrons, electronBcTop, electronBcBottom, \
      ionBcTop, ionBcBottom, emBcTop, emBcBottom]
  </UpdateStep>

  <UpdateStep compute>
     updaters = [ computeDivE, computeResidual, residualBc, periodicResidual]
```

```
  </UpdateStep>

  <UpdateStep clean>
     updaters = [ gradient]
  </UpdateStep>

  <UpdateStep hyper>
     operation = "integrate"
     updaters = [hyperIons, hyperElectrons, hyperEm, addSource]
  </UpdateStep>

  <UpdateStep implicit>
     operation = "operate"
     updaters = [twoFluidLorentz]
  </UpdateStep>

</Updater>
```

Finally, recall that the characteristic cell lengths only need to be calculated once. As a result we calculate them during the initialization step.

```
<UpdateStep initStep>
   updaters = [initElectrons, initIons, initEm, computeCellDx]
   syncVars = [electrons, ions, em]
</UpdateStep>
```

5.2.3 Computing the Reconnected Magnetic Flux

In the GEM Challenge simulation we've added passive diagnostic to compute the line integral across the domain. In order to compute the line integral we add two data structures. The first data structure is called a refmanual-bin. The *bin* is a uniform grid that overlays the exiting grid. The extents of the *bin* match the extents of the grid, but a *bin* is rectangular regardless of the shape of the grid. The *bin* has two important parameters, the first is *onGrid* which specifies which grid the bin will use to define itself, the second is the *scale*. *scale* tells roughly how many bins there are per grid cell in *domain*. The bin is used for fast lookup so that USim can quickly tell which cell a particular point in space is in. The bin is given as

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

In addition we need to fill the bin with data. In this particular case we want to fill each bin with a a list of indexes corresponding to the cells that fall inside each cell of the bin. The refmanual-binCells updater does exactly that. This updater only needs to be called at startup since the grid does not change with time.

```
<Updater fillBin>
   kind = binCells2d
   onGrid = domain
   out = [cellBin]
 </Updater>
```

The second variable is a refmanual-dynVector. A dynVector is a dataStructure which is a vector and has the same value on all domains in parallel simulations. dynVectors store a time series of data, so the value of the dynVector is dumped at every time step so it can be used to store integrated quantities. The *dynVector* had *numComponents* to specify how long the vector is. In this case it stores only 1 value. *writeOut=true* is set so the dynVector is written.

```
<DataStruct integratedFlux>
  kind = dynVector
  numComponents = 1
  writeOut = 1
</DataStruct>
```

These two data structures are then used to compute the line integral with the refmanual-lineIntegral updater. The refmanual-lineIntegral behaves like a refmanual-updater-combiner except that it takes in refmanual-nodalArray's and fills up a refmanual-dynVector.

```
<Updater computeLineIntegral>
  kind = lineIntegral2d
  onGrid = domain
  startPosition = [XMIN, 0.0]
  endPosition = [XMAX, 0.0]
  numberOfSamples = 1000

  layout = [cellBin]
  in = [em]
  indVars_em = ["ex","ey","ez","bx","by","bz","phiE","phiB"]
  exprs = ["0.5*abs(by)"]
  out = [integratedFlux]
</Updater>
```

This updater should not be called within an rkUpdater otherwise it will store multiple values per time step. Instead we call this updater in its own refmanual-UpdateStep

```
<UpdateStep diagnosticStep>
  updaters = [computeLineIntegral]
</UpdateStep>
```

5.2.4 An Example Simulation

The input file for the problem *GEM Challenge* in the USim USimHEDP package demonstrates each of the concepts described above to solve fast magnetic reconnection.

5.3 Using USim to Solve MHD with General Equation of State

In the USimBase tutorials the basic concepts of USim were described. In this HEDP tutorial we show how to use the ideal MHD equations with general equation of state and a simple radiation model.

Contents

5.3.1 Solving Problems using the idealMhdEosEqn

The plasma jets problem uses an ideal MHD model with general equation of state refmanual-idealMhdEosEqn and a simple radiation loss model refmanual-bremsPowerSrc. More sophisticated radiation loss can be included with purchase of the PROPACEOS tables from prism computational sciences or use of the SESAME tables from Los Alamos National Laboratory. The key to this whole development is the use of the refmanual-idealMhdEosEqn in the hyperbolic updater given below:

```
<Updater hyper>
  kind = classicMuscl$DIM$d
  onGrid = domain
  numericalFlux = NUMERICAL_FLUX
  timeIntegrationScheme = none

  variableForm = conservative
  preservePositivity = true
  limiter = [LIMITER, LIMITER, none]

  in = [q, pressure, soundSpeed]
  out = [qnew]

  cfl = CFL

  equations = [mhd]

  <Equation mhd>
    kind = idealMhdEosEqn
    basementDensity = $0.1*BASEMENT_DENSITY$
    basementPressure = $0.1*BASEMENT_PRESSURE$
    mu0 = MU0
    correctionSpeed = 0.0
  </Equation>

</Updater>
```

A few key differences for this refmanual-classicMuscl updater are that it takes in multiple input variables. In this case the conserved variables *q*, the fluid pressure *pressure* and an estimate of the speed of sound *soundSpeed*. Since the pressure and sound speed are inputs the user can define these however they wish, with the caveat that the sound speed should be a good estimate in order to produce accurate (and stable) results.

For every variable input into the refmanual-classicMuscl updater a limiter must be provided *limiter=[LIMITER, LIMITER, none]. In this case we limit the conserved variable and the pressure, but don't limit the sound speed since it's only used to add diffusion to the system and compute a time step. In this case the 'correctionSpeed* is being set to 0.0 as we will use a separate updater to evolve the correction potential.

The key issue then becomes how to compute the pressure and sound speed outside the hyperbolic update? As an example we use an ideal equation of state and use a refmanual-updater-combiner to compute the pressure from the energy:

```
<Updater computePressure>
  kind = combiner$DIM$d
  onGrid = domain

  in = [q]
  out = [pressure]
  mu0 = MU0
  gamma = GAMMA
  bp = BASEMENT_PRESSURE
  indVars_q = ["rho","mx","my","mz","en","bx","by","bz","phi"]
```

```
  exprs = ["max(1.1*bp,(gamma-1)*(en-(0.5/mu0)*(bx*bx+by*by+bz*bz)-0.5*(mx*mx+my*my+mz*mz)/rho))"]
</Updater>
```

In this case a basement pressure is used so the maximum of the basement pressure and the computed pressure is taken. Alternatively, the refmanual-propaceosVariables or refmanual-sesameVariables updaters may be used to compute an equation of state for the pressure as a function of the temperature and density. Similarly, we use a combiner to compute the sound speed based on the pressure and the density:

```
 <Updater computeSoundSpeed>
   kind = combiner$DIM$d
   onGrid = domain

   in = [q, pressure]
   out = [soundSpeed]
   gamma = GAMMA
   indVars_q = ["rho","mx","my","mz","en","bx","by","bz","phi"]
   indVars_pressure = ["pressure"]

   exprs = ["sqrt(gamma*pressure/rho)"]
</Updater>
```

These results can be easily applied in a *multiUpdater* to produce an MHD model with a user specified equation of state. The refmanual-multiUpdater for this system follows where the pressure and density are computed before *hyper* updater is called:

```
<Updater rkUpdater>
  kind = multiUpdater$DIM$d
  onGrid = domain
  timeIntegrationScheme = RKSCHEME

  updaters = [bc, computePressure, computeSoundSpeed, hyper]

  syncVars_bcBack =[q]
  syncVars_hyper = [qnew]

  integrationVariablesIn = [q]
  integrationVariablesOut = [qnew]
  dummyVariables_q = [dummy1, dummy2]

  syncAfterSubStep = [qnew]
</Updater>
```

5.3.2 Adding an Analytic Energy Loss Term

Adding an energy loss term is quite simple, except the user needs to make sure that the energy loss does not lead negative energies in the MHD model. In this case we use radiated power as estimated by analytical estimates of Bremsstrahlung radiation. The refmanual-bremsPowerSrc computes radiated power density (W/m^3) given plasma density, temperature and the effective Z. The block below shows how to use the refmanual-bremsPowerSrc:

```
<Updater computeRadPower>
   kind = equation$DIM$d
   onGrid = domain
   in =  [density, temperature, zEffective]
   out = [radiationPower]

  <Equation Bremsstrahlung>
    kind = bremsPowerSrc
```

```
  </Equation>
</Updater>
```

The dataStruct *radiationPower* has a single component and this value must then be subtracted from the fluid energy equation as shown below using a refmanual-updater-combiner:

```
 <Updater addRadiation>
   kind = combiner$DIM$d
   onGrid = domain

   in = [qnew, radiationPower]
   out = [qnew]

   indVars_qnew = ["rho","mx","my","mz","en","bx","by","bz","phi"]
   indVars_radiationPower = ["radiation"]

   exprs = ["rho","mx","my","mz","en-radiation","bx","by","bz","phi"]
</Updater>
```

In certain cases the radiated power an be huge, so much so that if the time step is not reduced all the energy in the system will dissappear. To get around this problem we use a *positiveValue* time step restriction inside a refmanual-timeStepRestrictionUpdater. A refmanual-timeStepRestrictionUpdater takes a list of input variables, and a list of *restriction*'s. The restriction in this case is the refmanual-positiveValue restriction. The variable *positiveIndex* refers to the index of the first input variable (in this case pressure). The variable *sourceIndex* refers to the index of the second variable *sourceIndex*. In this case the maximum time step is computed using

$$\Delta t = \text{alpha*(pressure[positiveValue]-basementValue)/(-coefficient*radiationPower[sourceIndex])} \quad (5.0)$$

And what the refmanual-positiveValue restriction does is it choses a time step so that you only subtract off the fraction *alpha* of the remaining energy in the input *pressure*. This time step is then compared with the other time steps computed by other restrictions and updaters and the smallest time allowable time step is used for the update:

```
<Updater timeStepRestriction>
  kind = timeStepRestrictionUpdater$DIM$d
  in = [pressure, radiationPower]
  onGrid = domain
  restrictions = [radiation]

  <TimeStepRestriction radiation>
    kind = positiveValue
    positiveIndex = 0
    sourceIndex = 0
    basementValue = BASEMENT_PRESSURE
    alpha = 0.25
    coefficient = -1.0
  </TimeStepRestriction>
</Updater>
```

In this case *radiatedPower* is always positive. Since *radiatedPower* is actually subtracted from pressure (not added) we set *coefficient=-1.0*. If *radiatedPower* was defined to be negative then *coefficient=1.0* would be the correct value. With radiation added the refmanual-multiUpdater looks like this:

```
<Updater rkUpdater>
  kind = multiUpdater$DIM$d
  onGrid = domain
  timeIntegrationScheme = RKSCHEME

  updaters = [correctMore, bc, computeDensity, computeTemperature, computePressure, \
```

```
                      computeSoundSpeed, computeRadPower, timeStepRestriction, \
                      hyper, addRadiation]

  syncVars_bcBack =[q]
  syncVars_hyper = [qnew]

  integrationVariablesIn = [q]
  integrationVariablesOut = [qnew]
  dummyVariables_q = [dummy1, dummy2]

  syncAfterSubStep = [qnew]
</Updater>
```

The timeStepRestriction is included in the refmanual-multiUpdater, but could also be included in a separate *UpdateStep* depending on how rapidly the power loss is changing.

5.3.3 Divergence Cleaning the Magnetic Field as A Separate Step

In this problem divergence cleaning is computed as a separate step using refmanual-hyperbolicCleanEqn. This approach has benefits because a less diffusive method can be used for the cleaning update and for certain choices of numerical flux, it can reduce overall diffusion of the solution. We use a separate refmanual-classicMuscl updater as defined below:

```
<Updater hyperClean>
  kind = classicMuscl2d
  timeIntegrationScheme = none
  variableForm = conservative

  cfl = CFL

  numericalFlux = hlleFlux

  limiter = [muscl]

  onGrid = domain
  in = [qClean]
  out = [qCleanNew]

  equations = [clean]

  <Equation clean>
    kind = hyperbolicCleanEqn
    waveSpeed = CORRECTIONSPEED
  </Equation>
</Updater>
```

Which uses the refmanual-hyperbolicCleanEqn. The refmanual-hyperbolicCleanEqn takes one parameter *waveSpeed* which is the speed that the correction wave propagates at. As with the refmanual-multiUpdater for evolving the rest of the MHD model, we must also include boundary conditions for the cleaning update:

```
<Updater rkClean>
  kind = multiUpdater2d
  onGrid = domain
  timeIntegrationScheme = rk2
  updaters = [cleanCopyBc, hyperClean]

  integrationVariablesIn = [qClean]
```

```
  integrationVariablesOut = [qCleanNew]
  dummyVariables_qClean = [dummyClean1,dummyClean2]

  syncAfterSubStep = [qCleanNew]
</Updater>

<Updater cleanCopyBc>
  kind = copy2d
  onGrid = domain
  out = [qClean]
  entity = ghost
</Updater>

<Updater copyClean>
  kind = linearCombiner2d
  onGrid = domain
  in = [qCleanNew]
  out = [qClean]
  coeffs = [1.0]
</Updater>
```

In the above example we used the *DataStructAlias* which is a pointer to a *DataStruct*:

```
<DataStructAlias qClean>
  kind = nodalArray
  target = q
  componentRange = [5,9]
</DataStructAlias>

<DataStructAlias qCleanNew>
  kind = nodalArray
  target = qNew
  componentRange = [5,9]
</DataStructAlias>
```

In the *DataStructAlias* we only need to define *target* which is the *DataStruct* that the alias is pointing to, the *componentRange* which defines which components of *DataStruct* to use. In the case above *componentRange=[5,9]* means that *qClean* points to elements [5,6,7,8] of *q*.

Finally, we put all of this into its own update step:

```
<UpdateStep cleanStep>
   updaters = [rkClean, copyClean]
</UpdateStep>
```

Which can then be called before the main refmanual-multiUpdater.

5.3.4 Using the vertexJetUpdater

The refmanual-vertexJetUpdater was an updater specifically designed for modeling the Los Alamos Plasma Liner Experiment. It allows one to easily generate a series of directed jets with uniform initial conditions each aligned about their own axis. The refmanual-vertexJetUpdater works in both 2 and 3 dimensions. The variety of options provided by the updater are described in the reference manual, however we can point out a few key things here. An arbitrary number of jets can be defined by adding additional vertices, but they must be labeled in sequential order *vertex0,1,2,3,....* The vertex represents the tip of the jet and that jet will be directed towards the location specified by *origin*. The *width* of the jet defines the width about each jet will be evaluated perpendicular to the line from the vertex to the origin. The *length* specifies the distance from the vertex along the line from the vertex to the origin that the

jet will be evaluated. The normalizedDensityFunction is used to define jets with non-uniform density profiles. The density profile is rotated into the coordinate system where X is parallel to the vector from the origin to the vertex and Y and Z are perpendicular. An example block is given below where it is used to update q:

```
<Updater jetSet>
  kind = vertexJetUpdater$DIM$d

  origin = [0.0, 0.0, 0.0]
  width = JET_INIT_WIDTH
  length = JET_INIT_LENGTH

  radialVelocity = $-U$

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

  <normalizedDensityFunction>
       kind = exprFunc
       preExprs = JET_DENSITY_PRE_FUNCTION
       exprs = JET_DENSITY_FUNCTION
  </normalizedDensityFunction>

  vertex0 = [RAD, 0.0, 0.0]
  vertex1 = [0.0, RAD, 0.0]
  vertex2 = [-RAD, 0.0, 0.0]
  vertex3 = [0.0, -RAD, 0.0]

  vertex4 = [$(1.0/sqrt(2.0))*RAD$, $(1.0/sqrt(2.0))*RAD$, 0.0]
  vertex5 = [$-(1.0/sqrt(2.0))*RAD$, $(1.0/sqrt(2.0))*RAD$, 0.0]
  vertex6 = [$-(1.0/sqrt(2.0))*RAD$, $-(1.0/sqrt(2.0))*RAD$, 0.0]
  vertex7 = [$(1.0/sqrt(2.0))*RAD$, $-(1.0/sqrt(2.0))*RAD$, 0.0]


  includeElectronTemperature = false

  correctionSpeed = CORRECTIONSPEED

  gasGamma = GAMMA

  #We use the mhdMunzEqn to initialize energy based on a temperature
  model = mhdMunzEqn
  mu0 = MU0
  onGrid = domain

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

5.3.5 An Example Simulation

The input file for the problem *Plasma Jet Merging* in the USim USimHEDP package demonstrates each of the concepts described above to solve the plasma jet merging problem.

5.4 Using USim to Solve MHD with General Equation of State

In the USimBase tutorials the basic concepts of USim were described. In this HEDP tutorial we show how to use the two-temperature mhd equations with general equation of state on an unstructured grid.

Contents

5.4.1 Solving Problems Using The twoTemperatureMhdEosEqn

This problem is performed on an unstructured grid in axisymmetric geometry. The standard grid block is used making sure to set *isRadial=true* and using the unstructured gmsh grid file called *dpf1.msh*:

```
<Grid domain>
   kind = unstructured

   ghostLayers = 2
   isRadial = true
   decomposeHalos = false

   <Creator ctor>
     kind = gmsh
     ndim = 2
     file = dpf1.msh
   </Creator>
 </Grid>
```

The refmanual-twoTemperatureMhdEosEqn is considerably more complex than any of the ideal MHD models since the electric field is computed externally. The hyperbolic updater is given below using the refmanual-classicMuscl method:

```
<Updater hyper>
  kind = classicMuscl2d
  onGrid = domain
  variableForm = conservative
  gradientType = leastSquares

  timeIntegrationScheme = none
  preservePositivity = true
  numericalFlux = NUMERICALFLUX
  limiterType = component

  in = [q, pressure, electronPressure, soundSpeed, J, E, Z]
  out = [qnew]

  cfl = CFL
  limiter = [LIMITER, LIMITER, LIMITER, none, LIMITER, LIMITER, none]
  equations = [mhd]
```

```
  sources = [axisymmetricSource, mhdSrc]

  <Equation mhd>
    kind = twoTemperatureMhdEosEqn
    mu0 = MU0
    ionMass = MI
    gasGamma = GAMMA
    fundamentalCharge = CHARGE
    basementDensity = BASEMENTDENSITY
    basementPressure = BASEMENTPRESSURE
  </Equation>

   <Source axisymmetricSource>
     kind = mhdSym
     symmetryType = cylindrical
     model = twoTemperatureMhdEosEqn
     inputVariables = [q, pressure, electronPressure, J, E, Z]
     fundamentalCharge = CHARGE
     mu0 = MU0
     ionMass = MI
   </Source>

   <Source mhdSrc>
     kind = mhdSrc
     model = twoTemperatureMhdEosEqn
     inputVariables = [q, J, E, Z]
     fundamentalCharge = CHARGE
     mu0 = MU0
     ionMass = MI
   </Source>

</Updater>
```

The updater differs from previous MHD type schemes in that we have 7 input variables which are described in the reference manual for refmanual-twoTemperatureMhdEosEqn. The 7 variables are the conserved variable vector q, the pressure p the electron pressure *electronPressure* an estimate of the local sound speed *soundSpeed* the current density (computed from the curl of the magnetic field) J, the electric field E and the local charge state Z which can vary spatially

Limiters must be defined for each input variable, in this case we have

```
limiter = [LIMITER, LIMITER, LIMITER, none, LIMITER, LIMITER, none]
```

where each limiter corresponds to the corresponding input variable. Limiters need to be applied to q, p, *electronPressure*, E and J since derivatives of each of these terms are compute. No limiters are applied to the remaining variable. Limiters could also be applied to *soundSpeed* and Z however the stability of the scheme does not depend on limiting this terms so we reduce the work load by not limiting these variables.

The *Equation* block is *kind=twoTemperatureMhdEosEqn* and is given by

```
<Equation mhd>
  kind = twoTemperatureMhdEosEqn
  mu0 = MU0
  ionMass = MI
  gasGamma = GAMMA
  fundamentalCharge = CHARGE
  basementDensity = BASEMENTDENSITY
  basementPressure = BASEMENTPRESSURE
</Equation>
```

Where the variables are described in the reference manual. In addition to the *Equation* block we require two *Source* blocks. The following line indicates which sources will be called by the updater

```
sources = [axisymmetricSource, mhdSrc]
```

The first source adds the axisymmetric terms of the hyperbolic flux so that the derivative is correct in axisymmetric geometry. This block is defined in refmanual-mhdSym and some specifics outlined here

```
<Source axisymmetricSource>
  kind = mhdSym
  symmetryType = cylindrical
  model = twoTemperatureMhdEosEqn
  inputVariables = [q, pressure, electronPressure, J, E, Z]
  fundamentalCharge = CHARGE
  mu0 = MU0
  ionMass = MI
</Source>
```

We are interested in cylindrical symmetry so *symmetryType=cylindrical* is specified. The model being used is *model = twoTemperatureMhdEosEqn*. The key parameter in the *axisymmetricSource* block is the *inputVariable* block. In computing sources, if USim does not include *inputVariable* block it will assume that the inputVariables will be provided for the updaters *in* list in the order they are provided. In this case this is not what is desired so we must specify a *inputVariable* list so that USim uses the correct variables in evaluating the source.

First note that in the *classicaMuscl2d* updater the *in* list is given as

```
in = [q, pressure, electronPressure, soundSpeed, J, E, Z]
```

The axisymmetric source does not need the variable *soundSpeed*, so, if no *inputVariables* list is specified in the axisymmetric source then USim will try and use the first 6 variables *q*,'pressure','electronPressure' *soundSpeed*,'J' and *E* in evaluating the source. Fortunately the size of *soundSpeed* is wrong (size 1 instead of the expected size 3) so USim will throw an exception and won't run. In order to get around this problem, and to get the correct computed source we specify the *inputVariables* list. The *inputVariables* can only contain variables specified in the *in* list of the *classicMusclUpdater2d*. In this example the correct *inputVariables* is

```
inputVariables = [q, pressure, electronPressure, J, E, Z]
```

note that there is no *soundSpeed* variable in this list.

The next *Source* is the *mhdSrc*. This source is needed to add the $E \cdot J_e$ term to the electron energy equation since the two-temperature mhd system cannot be written entirely in flux form. The source is desribed in refmanual-mhdSrc and some details are given here

```
<Source mhdSrc>
  kind = mhdSrc
  model = twoTemperatureMhdEosEqn
  inputVariables = [q, J, E, Z]
  fundamentalCharge = CHARGE
  mu0 = MU0
  ionMass = MI
</Source>
```

The *kind=mhdSrc* is specified and the model is *twoTemperatureMhdEosEqn*. As with the axisymmetric source the user must specify *inputVariables* to override the default input variables that would be used based on the *in* list from the *classicaMuscl2d* updater. In this case the list is given as

```
inputVariables = [q, J, E, Z]
```

5.4.2 Computing the Electric Field

The electric field could be computed using a *combiner*, but USim also has a refmanual-generalizedOhmsLaw updater which can simplify things. The *generalizedOhmsLaw* updater is discussed in the reference manual. It allows the user to add electron pressure gradient, resistive hall and ideal terms to the electric field. All derivatives (such as the electron pressure gradient) are computed outside of the *generalizedOhmsLaw* updater. The *generalizedOhmsLaw* used in this case is specified below

```
 <Updater computeE>
   kind = generalizedOhmsLaw2d
   onGrid = domain

   in = [q, J, Z]
   out = [E]

   hallTerm = false

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

The updater takes in parameters conserved variable vector q, current density J and the charge state Z. These terms are sufficient to compute the ideal and hall terms. In order to add in resistivity and the electron pressure gradient refer to the manual. In this case *hallTerm* is set to false as it severly constrains the time step in many explicit simulations. The output is stored in the electric field E which has 3 components regardless of whether a refmanual-generalizedOhmsLaw 1d, 2d or 3d is used.

5.4.3 Computing J through $\nabla \times B$

Derivatives such as gradient, divergence and curl can be computed using two different types of algorithms in USim. The first approach is a finite difference approach which is fast, but becomes inaccurate on triangular and sufficiently skewed quadrilateral meshes. This approach currently works consistently in 1, 2 and 3 dimension. The following example shows how to compute the curl using a finite difference approach

```
<Updater computeJ>
  kind = curl2d
  onGrid = domain
  inIndex = [5, 6, 7]
  outIndex = [0, 1, 2]
  in = [q]
  out = [J]
  coefficient = $1.0/MU0$
</Updater>
```

The updater is described in the reference manual. In this case we want the curl of the magnetic field to compute the current density J. The input variable is the conserved variable vector q and the output is the current density J. The magnetic field component stored in the conserved variable vector q occur in components [5,6,7] of the vector q so we must specify *inIndex=[5,6,7]* so that USim takes the correct components when computing J.

The second approach produces better results on unstructured grids and uses a least squares approach to computing the curl. This method is more accurate than the finite difference approach and is robust in 1 and 2 dimensions, but is slower than the previous method. An example block using the least squares refmanual-vector, with the option curl is given below

```
<Updater computeJ>
  kind = vector2d
  onGrid = domain
  derivative = curl
  numScalars = 1
  orderAccuracy = 1
  coefficient = $1.0/MU0$
  numberOfInterpolationPoints = 5

  in = [BPhi]
  out = [J]
</Updater>
```

This updater expects an input vector of size 3 and produces and output vector of size 3. In this case, a second data struct *BPhi* was created and the magnetic field form *q* was copied into that variable. In addition, the parameter *numberOfInterpolationPoints* must be specified. The value chosen here is variable depending on the grid, but should generally be 5 or greater in 2d. Note that the complete list of options available in refmanual-vector are gradient, curl and divergence.

5.4.4 Computing Pressure

Discussion of the pressure calculation for this system is included for clarity using a refmanual-updater-combiner. Pressure in this case is total pressure or electron pressure + ion pressure. If we are using the ideal gas law (as is done in this case) the pressure must be broken into ion pressure and electron pressure separately. The combiner below does just that

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

  in = [q, electronPressure]
  out = [pressure]
  mi = MI
  mu0 = MU0
  gamma = GAMMA
  k=KB
  indVars_q = ["rho","mx","my","mz","en","bx","by","bz","ene"]
  indVars_electronPressure = ["ene"]
  exprs = ["(gamma-1)*(en-ene-(0.5/mu0)*(bx*bx+by*by+bz*bz)-0.5*(mx*mx+my*my+mz*mz)/rho)+(gamma-1.0)*
</Updater>
```

5.4.5 Current Boundary Condition

The boundary at the inlet is specified using a refmanual-functionBc. The *functionBc* allows you to specify the boundary condition as a function of time *t* and spatial variables *x*,'y','z'. This is to be contrasted with the refmanual-generalBc which also allows boundaries to be specified as functions of other *DataStructs*. The *functionBc* used in this example is specified below

```
<Updater bcSource>
  kind = functionBc2d
  onGrid = domain
  out = [q]

  entity = source
```

```
  <Function func>
     kind = exprFunc

     #insulator end
     iEnd = $0.027-0.81e-2$
     iRad = $1.5*0.27e-2$
     b0 = B0
     riseTime = 2.0e-6
     mu0 = MU0
     gamma = GAMMA
     p=$BETA*P0$
     rho0 = $2.0*BASEMENTDENSITY$

     preExprs = ["uz=1.0e3","by = (b0*iRad/x)*sin(6.28*t/riseTime)","en=(p/(gamma-1))+0.5*rho0*uz*uz-

     exprs = ["rho0","0.0","0.0","rho0*uz","en","0.0","by","0.0","0.5*p/(gamma-1.0)"]
  </Function>

</Updater>
```

As with all USim boundary conditions an *entity* must be defined to tell USim where to apply the boundary condition. In addition a *Function* block must be specified to define the initial conditions. The *kind* of this block is always *exprFunc*. Inside this block we specify a *preExprs* list to provide some work space for performing computations before the values of *q* are set in the boundary with *exprs*. The *preExprs* list is given below

```
preExprs = ["uz=1.0e3","by = (b0*iRad/x)*sin(6.28*t/riseTime)","en=(p/(gamma-1))+0.5*rho0*uz*uz+0.5*
```

We've specified *by* as a *sin* function to mimic current rise and fall. The values defined in the *preExprs* can then be used in *exprs*

```
exprs = ["rho0","0.0","0.0","rho0*uz","en","0.0","by","0.0","0.5*p/(gamma-1.0)"]
```

In this examples magnetic field is fed into the domain using a small velocity *uz* and low density *rho0*. Using this technique we can make sure the magnetic field gets into the domain without defining a "resistive" layer near the wall. Since the model is an MHD model a finite plasma density must exist so that the Alfven wave speed does not become infinite.

5.4.6 An Example Simulation

The input file for the problem *Dense Plasma Focus* in the USimHEDP package demonstrates each of the concepts described above to evolve the dense plasma focus problem in 2D axisymmetric geometry using two temperature MHD with a general equation of state.

5.5 Using USim to Solve a Magnetic Nozzle Problem

In this tutorial we show how to use USim to solve a problem with both an induced and imposed magnetic field using the gas dynamic form of the mhd equations.

Contents

5.5.1 Solving Problems Using The Gas Dynamic Form of the MHD Equations

This tutorial follows many of the same concepts as the previous tutorial *Using USim to Solve MHD with General Equation of State*. In this case we are solving the MHD system written in gas dynamic form. This form is significant since it is not conservative, but simplifies some considerations when there are both imposed and induces magnetic fields.

First of all we want to split the field into an imposed field generated by external coils and and induced field generated by the plasma motion. To do this we have 3 data structures, the first, *backgroundB* stores the field generated by a magnetic field coil

```
<DataStruct backgroundB>
   kind = nodalArray
   onGrid = domain
   numComponents = 3
 </DataStruct>
```

The second variable stores the conserved variables along with the total magnetic field, i.e., the induced field + the background field

```
<DataStruct qModified>
  kind = nodalArray
  onGrid = domain
  numComponents = NUMCOMP
</DataStruct>
```

The 3rd variable stores the conserved variables and only the induced magnetic field

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

The imposed magnetic field is generated at startup using a wire source refmanual-wireFieldEqn and a refmanual-equation updater. The refmanual-equation updater works with all USim refmanual-Source blocks. Since the problem is not axisymmetric we use a wire, but could use a refmanual-coilFieldEqn to model a current carrying loop. The refmanual-wireFieldEqn initialization of the background magnetic field is given below

```
<Updater initMagField>
  kind = equation2d
  onGrid = domain
  in = []
  out = [backgroundB]

  <Equation coil>
    kind = wireFieldEqn

    outRange = [0, 1, 2]
```

```
    point = [$COILRAD$, -0.001, 0.0]
    mu0 = MU0
    normal = [0.0, 0.0, 1.0]
    current = $COILCURRENT$
  </Equation>
</Updater>
```

Now the fields are split into induces q and imposed fields *backgroundB*. The imposed field *backgroundB* does not evolve in time, but does affect the flow of fluid through the $J \times B$ force which is treated as a source in this example. Taking this into account we apply the hyperbolic update to q (which ignores the imposed field) not *qModified* which includes the imposed field. The refmanual-classicMuscl updater is used with the input file seen below

```
<Updater hyper>
  kind = classicMuscl2d
  timeIntegrationScheme = none
  gradientType = leastSquares

  numericalFlux = hlleFlux
  variableForm = conservative
  preservePositivity = true

  limiter = [muscl, none, muscl, none]

  onGrid = domain
  in = [q, J, E, Z]
  out = [qnew]

  cfl = CFL

  equations = [mhd]

  <Equation mhd>
    kind = gasDynamicMhdEqn
    gasGamma = ION_GAMMA
    mu0 = MU0
    correctionSpeed = 0.0
    ionMass = ION_MASS
    chargeState = ZRATIO
    fundamentalCharge = FUNDAMENTAL_CHARGE
  </Equation>
</Updater>
```

Since this equation system does not use the general equation of state the inputs are somewhat simpler than in the case of the refmanual-twoTemperatureMhdEosEqn. The inputs are the conserved variables q, the total current J, the electric field E and the charge state Z. Also, note that we have not included the refmanual-mhdSrc which was included in the case of refmanual-twoTemperatureMhdEosEqn. Instead we add the refmanual-mhdSrc in a separate step. First we compute *qModified* which includes the conserved variables and both the induced and imposed field (we use a refmanual-updater-combiner)

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

   in = [q, backgroundB]
   out = [qModified]

   indVars_q = ["rho", "mx", "my", "mz", "en", "bx", "by", "bz", "phi", "ee"]
   indVars_backgroundB = ["B0x","B0y","B0z"]
```

```
   exprs = ["rho", "mx", "my", "mz", "en", "bx+B0x", "by+B0y", "bz+B0z", "phi", "ee"]
 </Updater>
```

The current density is computed from the perturbed magnetic field only $J = \frac{1}{\mu_0}\nabla \times B$ since it is already known that that the curl of the imposed field is zero everywhere inside the domain

```
<Updater computeJ>
    kind = vector2d
   onGrid = domain
   derivative = curl
   numScalars = 1
   orderAccuracy = 2
   coefficient = 1.0
   numberOfInterpolationPoints = 8

   in = [B]
   out = [J]
 </Updater>
```

Now that we've computed *qModified* we use this to compute the refmanual-mhdSrc through the use of a refmanual-equation so that both the induced and imposed fields provide a force to the fluids. The source term is stored in *src*.

```
<Updater computeSource>
   kind = equation2d
   in = [qModified, J, E, Z]
   out = [src]
   onGrid = domain

   <Equation gdSrc>
      kind = mhdSrc
      model = gasDynamicMhdEqn
      gasGamma = ION_GAMMA
      electronGasGamma = ELECTRON_GAMMA
      ionMass = ION_MASS
      chargeState = ZRATIO
      fundamentalCharge = FUNDAMENTAL_CHARGE
      correctionSpeed = 0.0
      mu0 = MU0
   </Equation>
 </Updater>
```

The source term is then added to the update *q*, *qnew* through the use of a refmanual-uniformCombiner. The refmanual-uniformCombiner acts just like a refmanual-updater-combiner except that it assumes all input variables and output variables are of the same size and that the same operation is being applied to all components. In the block below we have added *src* generated by the *computeSource* Updater above

```
<Updater updateQ>
  kind = uniformCombiner2d
  onGrid = domain

  in = [qnew, src]
  out = [qnew]

  indVars_qnew = ["qn"]
  indVars_src = ["s"]
  exprs = ["qn+s"]
</Updater>
```

As in the previous tutorial the electric field is computed using a refmanual-generalizedOhmsLaw updater. Notice

that the conserved variables including the total magnetic field *qModified* is used in computing the electric field. In addition, we've included a resistivity term by defining *resistivity=eta*. *eta* is a refmanual-nodalArray defined using a refmanual-updater-combiner

```
<Updater computeE>
  kind = generalizedOhmsLaw2d
  onGrid = domain

  in = [qModified, J, Z]
  out = [E]
  resistivity = eta

  hallTerm = false

  fundamentalCharge = FUNDAMENTAL_CHARGE
  ionMass = ION_MASS
  electronMass = ME
  boltzmannConstant = KB

  mu0 = MU0
</Updater>
```

5.5.2 Computing the Δt restriction for explicit resistive term

USim has a refmanual-TimeStepRestriction for explicit diffusion type terms. The restriction can be applied to viscous, thermal diffusion and resistive terms. In order to compute the restriction properly the diffusion coefficient needs to be computed properly. In the case of resistivity we want to compute the diffusion coefficient γ in the proper form

$$\frac{\partial B}{\partial t} = \gamma \nabla^2 B$$

It turns out *gamma=eta/mu_{0}* in the case of magnetic field diffusion. In this simulation we first compute the resistivity

```
<Updater initEta>
   kind = initialize2d
   onGrid = domain
   out = [eta]

   <Function func>
     kind = exprFunc
     eta0 = RESISTIVITY
     exprs = ["eta0"]
   </Function>
 </Updater>
```

and then compute *etaBymu0 = eta0/mu0* so that we can compute the explicit time step constraint

```
<Updater initEtaBymu0>
   kind = combiner2d
   onGrid = domain
   in = [eta]
   out = [etaBymu0]
   indVars_eta = ["eta0"]
   mu0 = $MU0$
   exprs = ["eta0/mu0"]
 </Updater>
```

etaByMu0 can then be used in the refmanual-quadratic time step restriction updater

```
<Updater timeStepRestriction>
  kind = timeStepRestrictionUpdater2d
  in = [etaBymu0]
  onGrid = domain
  restrictions = [quadratic]

  <TimeStepRestriction quadratic>
    kind = quadratic
    cfl = 0.5
  </TimeStepRestriction>
</Updater>
```

Note that diffusion terms have explicit time step restriction given by $\Delta t \approx \Delta x^2$ so that the time step reduces quadatically as the grid is refined. Currently we can get around this problem in USim by using super time stepping (see *Advanced Time-Stepping Methods in USim*).

5.5.3 Divergence Cleaning with an Imposed Field

Divergence cleaning with imposed fields is accomplished by cleaning the induced field only. It is already known that the imposed field is divergence free so that part can be ignored. Hyperbolic divergence cleaning in this case, but only applied to the perturbed field. A refmanual-DataStructAlias is used to point just to the magnetic field and correction potential components in the conserved variable vector

```
<DataStructAlias qClean>
   kind = nodalArray
   target = q
   componentRange = [5,9]
   writeOut = 0
 </DataStructAlias>
```

This refmanual-DataStructAlias is used in the hyperbolic cleaning step which consists of a refmanual-classicMuscl updater, a refmanual-multiUpdater and associated boundary conditions.

```
<Updater hyperClean>
  kind = classicMuscl2d
  timeIntegrationScheme = none
  gradientType = leastSquares
  variableForm = conservative

  cfl = CFL

  numericalFlux = hlleFlux

  limiter = [muscl]

  onGrid = domain
  in = [qClean]
  out = [qCleanNew]

  equations = [clean]

  syncAfterSubStep = [qCleanNew]
  <Equation clean>
    kind = hyperbolicCleanEqn
    waveSpeed = CORRECTION_SPEED
  </Equation>
```

```
</Updater>

<Updater hyperClean2>
  kind = multiUpdater2d
  onGrid = domain
  timeIntegrationScheme = rk2
  updaters = [cleanCopyBc, cleanInflow, hyperClean]

  integrationVariablesIn = [qClean]
  integrationVariablesOut = [qCleanNew]
  dummyVariables_qClean = [dummyClean1,dummyClean2]

  syncAfterSubStep = [qCleanNew]
</Updater>
```

In this case the boundary conditions consist of copying out the magnetic field and reversing the sign of the correction potential. In can be the case that field build up can occur at the boundary and in these cases simply setting the perturbed magnetic field to zero resolves the issue. Boundary conditions used for the hyperbolic cleaning step with the perturbed magnetic field are shown below using a refmanual-generalBc

```
<Updater cleanCopyBc>
  kind = generalBc2d
  onGrid = domain

  in = [qClean]
  dynVectors = []

  indVars_qClean = ["Bx","By","Bz","Phi"]

  exprs = ["Bx","By","Bz","-Phi"]
  out = [qClean]

  entity = ghost
</Updater>

<Updater cleanInflow>
  kind = generalBc2d
  onGrid = domain

  in = [qClean]
  dynVectors = []

  indVars_qClean = ["Bx","By","Bz","Phi"]

  exprs = ["Bx","By","Bz","-Phi"]
  out = [qClean]

  entity = ghost
</Updater>
```

5.5.4 An Example Simulation

The input file for the problem *Magnetic Nozzle* in the USimHEDP package demonstrates each of the concepts described above.

5.6 Using USim to Solve an Anisotropic Diffusion Problem

In this tutorial we show how to use USim to solve a problem with anisotropic diffusion using the updater refmanual-diffusion with option anisotropicDiffusion. The problem used an example is based off of the problem described in

```
Parrish, Ian J., and James M. Stone. "Nonlinear evolution of the magnetothermal instability in two di
```

but solved on an unstructured grid.

Contents

5.6.1 Required DataStructs

This tutorial follows many of the same concepts as prior tutorials

First of all we initialize a vector (in this case B) that defines the field that is used to construct the conductivity tensor

```
<DataStruct B>
  kind = nodalArray
  onGrid = domain
  numComponents = 3
  writeOut = 1
</DataStruct>
```

In addition we define the parallel conductivity (parallel to the vector field) *kParallel*

```
<DataStruct kParallel>
  kind = nodalArray
  onGrid = domain
  numComponents = 1
  writeOut = 1
</DataStruct>
```

And the conductivity perpendicular to the vector field *kPerpendicular*

```
<DataStruct kPerpendicular>
  kind = nodalArray
  onGrid = domain
  numComponents = 1
  writeOut = 1
</DataStruct>
```

Finally we define the conductivity tensor *conductivityTensor*. The conductivity tensor always has 9 components even if the simulation is 2 dimensional

```
<DataStruct conductivityTensor>
  kind = nodalArray
  onGrid = domain
  numComponents = 9
```

```
  writeOut = 1
</DataStruct>
```

5.6.2 Computing a Conductivity Tensor

In general the *conductivityTensor* could be filled up manually using a refmanual-updater-combiner, but thermal diffusion in a magnetic field is best split into parallel and perpendicular conductivities. A built in source computes this automatically given the parallel and perpendicular conductivities. The *conductivityTensor* takes a vector field *B* a parallel conductivity *kParallel* (scalar) and a perpendicular conductivity *kPerpendicular* (scalar). The use of a Conductivity Tensor is given below

```
<Updater initConductivityTensor>
  kind = equation2d
  onGrid = domain

  in = [B, kParallel, kPerpendicular]

  out = [conductivityTensor]

  <Equation a>
    kind = conductivityTensor
  </Equation>

</Updater>
```

5.6.3 Solving problems using Derivatives with option *anisotropicDiffusion*

Now that *conductivityTensor* is defined we can compute the diffusion. We use the refmanual-diffusion which supplies second order derivatives. In this case the we use anisotropicDiffusion which expects a scalar input (in this case *temperature*) and a 9 component conductivity tensor (in this case *conductivityTensor*)

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

  orderAccuracy = 1
  numberOfInterpolationPoints = 8

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

Note that the complete list of options available in refmanual-diffusion are diffusion, anisotropicDiffusion and gradientOfDivergence.

5.6.4 Computing the time step for the diffusion operator

Time integration is performed using super time stepping. Super time stepping is a variable stage Runge-Kutta approach that is much faster (by the number of stages) than standard Runge-Kutta methods for solving diffusion

problems. The approach requires two time steps. The desired (actual) time step is computed using a refmanual-timeStepRestrictionUpdater. The key here is that *CFLSTEP=100.0* so it is much higher than the explicitly stable time step for a diffisive system

```
 <Updater timeStepRestriction>

  kind = timeStepRestrictionUpdater2d
  in = [maxConductivity]

  onGrid = domain
  restrictions = [quadratic]

  <TimeStepRestriction quadratic>
    kind = quadratic
    cfl = CFLSTEP
  </TimeStepRestriction>

</Updater>
```

Next the time step for the super time stepping method is computed where an explicitly stable *CFL=0.1* is used. The time step is stored in *diffDT1*

```
 <Updater getDiffDT1>
  kind = getTimeStepUpdater2d
  in = [maxConductivity]
  out = [diffDT1]
  onGrid = domain
  restrictions = [quadratic]

  <TimeStepRestriction quadratic>
    kind = quadratic
    cfl = CFL
  </TimeStepRestriction>
</Updater>
```

The Super Time Stepping integrator then knows to take the ratio of the desired time step and the explicitly stable time step to compute the number of stages used in the STS Updater.

5.6.5 An Example Simulation

The input file for the problem *Anisotropic Diffusion* in the USimHEDP package demonstrates each of the concepts described above.

5.7 Using USim to Solve Multi-Fluid Problems with Collisions

In this tutorial we show how to use USim to solve a problem with collisions in a multi-fluid simulation. This tutorial uses refmanual-collisionFrequency, refmanual-momentumEnergyExchange, refmanual-temperatureRelaxation.

Contents

5.7.1 Required DataStructs

In this example we have 3 different fluids. The first thing we need to define is a variable to contain the collision matrix. Since there are 3 fluids the collision matrix will have 3X3=9 components

```
<DataStruct collisionMatrix>
  kind = nodalArray
  onGrid = domain
  numComponents = 9
  writeOut = 1
</DataStruct>
```

So that we can look at the total density, energy and momentum of the system we define *qTotal* for convenience (this variable is not necessary for this simulation)

```
<DataStruct qTotal>
  kind = nodalArray
  onGrid = domain
  numComponents = 5
</DataStruct>
```

For each of the 3 fluids we define a vector to contain the mass density, momentum density and energy density. The first fluid *DataStruct* is defined as *q1* along with its updated value *qnew1*

```
<DataStruct q1>
  kind = nodalArray
  onGrid = domain
  numComponents = 5
</DataStruct>

<DataStruct qnew1>
  kind = nodalArray
  onGrid = domain
  numComponents = 5
  writeOut = false
</DataStruct>
```

Also, we need a variable to hold the momentum and energy collisional source terms for each of the fluids. The length of the vector is 5 to match the number of conserved variables even though the collisional source for the first variable (density) is 0

```
 <DataStruct q1Mom>
  kind = nodalArray
  onGrid = domain
  numComponents = 5
```

```
  writeOut = false
</DataStruct>
```

In addition data for storing collisions due to thermal energy exchange for each species are also created. The length of this DataStruct is 1

```
<DataStruct tempRelax1>
  kind = nodalArray
  onGrid = domain
  numComponents = 1
  writeOut = false
</DataStruct>
```

Data for the species number density is required for each species

```
<DataStruct N1>
  kind = nodalArray
  onGrid = domain
  numComponents = 1
  writeOut = 1
</DataStruct>
```

Data for the temperature of each species is also required

```
<DataStruct T1>
  kind = nodalArray
  onGrid = domain
  numComponents = 1
</DataStruct>
```

Data is also required for the velocity of each species

```
<DataStruct V1>
  kind = nodalArray
  onGrid = domain
  numComponents = 3
</DataStruct>
```

5.7.2 Computing the collision tensor using Collision Frequency

In general the *collisionFrequency* could be filled up manually using a refmanual-updater-combiner, but it is much easier to use *collisionFrequency*. The *in* variable takes *temperature number density* and *velocity* for each species (repeated as below for any number of species). In the case where plasma collision are modeled *type=ionized* charge states *Z* are also required. In the example below *type=neutrals* is used so collision cross sections are computed from species diameters given by *speciesDia*. Notice that *inverse=false* which means collision frequencies are returned in *collisionMatrix*. If *inverse=true* then collision times are returned

```
<Updater collisionFrequency>
  kind = equation1d
  onGrid = domain

  in =  [T1, N1, V1, T2, N2, V2, T3, N3, V3]

  out = [collisionMatrix]

  <Equation momentum>
    kind = collisionFrequency
    inverse = false
```

```
    type = neutrals
    speciesMass = [MI1, MI2, MI3]
    speciesDia = [DI1, DI2, DI3]
  </Equation>
</Updater>
```

5.7.3 Computing the momentum and energy exchange terms using Momentum Energy Exchange

Once *collisionMatrix* is computed it can be used to compute the momentum and energy exchange term can be computed. Note that this term ignores thermal collisions which are computed using a different operator. refmanual-momentumEnergyExchange takes in the conserved variables for each of the fluids along with the collisionMatrix and produces momentum and energy exchange terms that can then be added to the fluid equations

```
<Updater momentumSource>
  kind = equation1d
  onGrid = domain
  in =  [q1, q2, q3, collisionMatrix]
  out = [q1Mom, q2Mom, q3Mom]

  <Equation momentum>
    kind = momentumEnergyExchange
    speciesMass = [MI1, MI2, MI3]
  </Equation>
</Updater>
```

5.7.4 Computing the temperature relaxation terms using Temperature Relaxation

collisionMatrix is also used to compute the temperature relaxation between fluids. *temperatureRelaxation* takes mass densities or number densities (depending on whether *isNumberDensity=1* (takes number densities) or *isNumberDensity=0* (takes mass densities). *q1*,‘q2‘ and *q3* are conserved variable vectors where the first component is mass density, the remaining 4 components of the vector are ignored by *temperatureRelaxation*. The resulting relaxation terms are stored in the output variables and then are added to energy term of the respective fluid equations to compute the effect of temperature relaxation

```
<Updater energySource>
  kind = equation1d
  onGrid = domain
  in =  [q1, q2, q3, T1, T2, T3, collisionMatrix]
  out = [tempRelax1, tempRelax2, tempRelax3]

  <Equation momentum>
    kind = temperatureRelaxation
    isNumberDensity = 0
    speciesMass = [MI1, MI2, MI3]
  </Equation>
</Updater>
```

5.7.5 Adding the momentum/energy sources to the fluid equations

Now that the momentum and energy exchange terms have been computed along with the temperature relaxation term have been computed these are added to the solution after the hyperbolic part is computed. The source terms are added using a combiner. Notice that the energy term contains contributions from both the momentum exchange and the

thermal relaxation term. The updater for the first fluid is given below, the updaters for the remaining fluids will be similar

```
<Updater addThermalRelaxation1>
  kind = combiner1d
  onGrid = domain

  in = [qnew1, tempRelax1, q1Mom]
  out = [qnew1]

  indVars_qnew1 = ["rho","mx","my","mz","en"]
  indVars_q1Mom = ["dRho","dMx","dMy","dMz","dEn"]
  indVars_tempRelax1 = ["dT"]

  exprs = ["rho","mx+dMx","my+dMy", "mz+dMz","en+dT+dEn"]

</Updater>
```

5.7.6 Computing the time step for collisions using Frequency

Collisions add new time scales that need to be applied when an explicit approach is used. A special updater is used to compute the smallest time scale introduced by *collisionMatrix*. We use refmanual-timestepRestriction which takes in *collisionMatrix* and then used the restriction kind given by *kind=frequency*. The frequency restriction takes *components* which tells USim how many of the components in *collisionMatrix* should be used. In this case all components are used. This restriction will ensure that the explicit solution is stable to the collisions

```
<Updater timestepRestriction>
  kind = timeStepRestrictionUpdater1d
  in = [collisionMatrix]
  onGrid = domain
  restrictions = [inverseTime]

  <TimeStepRestriction inverseTime>
    kind = frequency
    components = 9
    cfl = 0.5
  </TimeStepRestriction>
</Updater>
```

5.7.7 An Example Simulation

The input file for the problem *Multi-Fluids with collisions* in the USimHEDP package demonstrates each of the concepts described above.

5.8 Using USim to solve 10 moment ions with 5 moment electrons

In this tutorial we show how to solve the two-fluid system where the ions use the 10-moment model and the electrons use the 5 moment model. The simulation is based off of the result published in

```
Hakim, Ammar H. "Extended MHD modelling with the ten-moment equations." Journal of Fusion Energy 27.1
```

Readers should refer to *Using USim to Solve the Two-Fluid Plasma Model* since the setup is virtually identical. The difference in this case lays in the use of refmanual-tenMomentEqn as well as the options *type = 10MomentIonsStep1* and *type = 10MomentIonsStep2* in refmanual-twoFluidSrc.

Contents

5.8.1 DataStructs

The electrons use the 5 moment model so the electron conserved variables have 5 components refmanual-eulerEqn

```
<DataStruct electrons>
  kind = nodalArray
  onGrid = domain
  numComponents = 5
</DataStruct>
```

The ions use the 10 moment model so the ion conserved variables have 10 components refmanual-tenMomentEqn

```
<DataStruct ions>
  kind = nodalArray
  onGrid = domain
  numComponents = 10
</DataStruct>
```

5.8.2 Computing the semi-implicit operators

The first part of the semi-implicit operator is computed as shown below. The *type* must be set to *10MomentIonsStep* as this operator is applied first

```
<Updater twoFluidSrc1>
   kind = equation1d

   onGrid = domain
   in =  [electronsNew, ionsNew, emNew]
   out = [electronsNew, ionsNew, emNew]

  <Equation twofluidLorentz>
    kind = twoFluidSrc
    type = 10MomentIonsStep1
    useImposedField = false
    electronCharge = ELECTRON_CHARGE
    electronMass = ELECTRON_MASS
    ionCharge = ION_CHARGE
    ionMass = ION_MASS
    epsilon0 = EPSILON0
  </Equation>
</Updater>
```

The second operator can be applied after the conserved variables have been updated from the first operator. In this case the *type* is *10MomentIonsStep2* as shown below. Notice that the result is only applied to the updated ion values *ionsNew*

```
<Updater twoFluidSrc2>
   kind = equation1d

   onGrid = domain
   in =  [ionsNew, emNew]
   out = [ionsNew]

  <Equation twofluidLorentz>
    kind = twoFluidSrc
    type = 10MomentIonsStep2
    ionCharge = ION_CHARGE
    ionMass = ION_MASS
  </Equation>
</Updater>
```

5.8.3 Integrating the 10 moment ion 5 moment electron system

Time integration is performed in the standard way, but now two operators must be applied. Notice the block

```
<UpdateStep operators>
  updaters = [twoFluidSrc1, twoFluidSrc2]
  syncVars = [emNew, ionsNew, electronsNew]
</UpdateStep>
```

where both *twoFluidSrc1* and then *twoFluidScr2* are applied

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

  in = [em, ions, electrons]
  out = [emNew, ionsNew, electronsNew]

  <TimeIntegrator rkIntegrator>
    kind = rungeKutta1d
    ongrid = domain
    scheme = secondNonTVD
  </TimeIntegrator>

  <UpdateSequence sequence>
    fluxStep = [hyper]
    boundaryStep = [boundaries]
    operatorStep = [operators]
  </UpdateSequence>

  <UpdateStep boundaries>
     updaters = [bcIons, bcElectrons, bcEm]
     syncVars = [em, ions, electrons]
  </UpdateStep>

  <UpdateStep hyper>
     updaters = [hyperIons, hyperElectrons, hyperEm]
     syncVars = [emNew, ionsNew, electronsNew]
  </UpdateStep>

  <UpdateStep operators>
     updaters = [twoFluidSrc1, twoFluidSrc2]
```

```
    syncVars = [emNew, ionsNew, electronsNew]
  </UpdateStep>

</Updater>
```

5.8.4 An Example Simulation

The input file for the problem *Ten-Moment Two-Fluid Shock* in the USimHEDP package demonstrates each of the concepts described above.

5.9 Using USim to Solve Navier-Stokes Equations

Conservative form of Navier-Stokes equations are solved in USim. Convective and diffusion terms are de-coupled. Hence the diffusion terms are added as source in the Euler equations (see *Using USim to solve the Euler Equations*).

Contents

5.9.1 Viscous and Thermal Terms in Navier-Stokes Equations

Navier-Stokes equations are solved in *Supersonic cross flow over cylinder* example of the USimHS by adding the diffusion terms to regular Euler equations. The diffusion terms considered for the evaluation are found in refmanual-navierStokesViscousOperator. An example updater block that evaluates the diffusion terms in momentum and energy equations is shown below:

```
<Updater computeViscousFluxes>
  kind = navierStokesViscousOperator2d
  isRadial = false
  numberOfInterpolationPoints = 8
  onGrid = domain
  coefficient = 1.0
  enableThermal = true
  enableViscous = true
  temperatureIndex = 0
  in = [velocity, dynamicViscosity, temperature, thermalCoefficient]
  out = [source]
</Updater>
```

The meanings for the input blocks in this updater are as follows:

`kind` (string)

> Specifies the *navierStokesViscousOperator2d* updater, which tells USim to evaluate the *diffusion terms* on two-dimensional grid. Similarly *navierStokesViscousOperator1d* and *navierStokesViscousOperator3d* can be used for one and three dimensional space domains.

`isRadial` (boolean)

Specifies whether the evaluation is carried out in cylindrical or Cartesian coordinates.

`numberOfInterpolationPoints` (string)

Number of interpolation points for leastsquares fit.

`enableThermal` (boolean)

Option to include heat conduction term in energy equation. time t, here this is *qSpecies*.

`enableViscous` (boolean)

Option to include viscous terms in momentum and energy equations.

`temperatureIndex` (int)

Index of temperature in the array that stores temperature. Lets say the temperature array that is passed for diffusion source computation has three values translational, vibrational, and rotational temperature. Since translational temperature is used in the computation of heat conduction and viscous dissipation, we pass *temperatureIndex* = 0.

`in` (nodalArrays)

The variables required for the computation of diffusion terms. The names in the examples are self-explanatory and the order should not be changed.

`out` (nodalArray)

The variable to store the evluated diffusion terms.

5.9.2 Evaluation and Addition of the Diffusion Terms

The Updater block is evaluated in the time integration *multiUpdater* of the input file. Again referring to *Supersonic cross flow over cylinder* example of the USimHS package, the following updater block is used for time integration of fluxes using first order Runge-Kutta method. In this *multiUpdater*, *computeViscousFluxes* is added to the list of the updaters (updaters = [.,.,.,.,.]) to evaluate diffusion fluxes.:

```
<Updater rkUpdaterMain>
  kind = multiUpdater2d
  onGrid = domain

  in = [q]
  out = [qnew]

  <TimeIntegrator rkIntegrator>
    kind = rungeKutta2d
    ongrid = domain
    scheme = first
  </TimeIntegrator>

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

  <UpdateStep boundaries>
     updaters = [correct, bcBottom, bcTop, bcRight, bcWall, bcLeft, computeT, bcWallTemp]
     syncVars = [q]
  </UpdateStep>

  <UpdateStep compute>
  operation = "operate"
```

```
    updaters = [computeVelocity, computeViscosity, computeKinematicViscosity, computeThermalCoeffici
                computeThermalDiffusivity, computeViscousFluxes]
    syncVars = [source]
  </UpdateStep>

  <UpdateStep hyper>
    operation = "integrate"
    updaters = [hyper,addSource]
    syncVars = [qnew]
  </UpdateStep>

</Updater>
```

option *kind* (string)

name of of the this time integration updater *multiUpdater2d* (2d for two dimensional space).

option *in* (nodalArray)

variables to be integrated in time.

option *out* (nodalArray)

name of the output variable.

option *TimeIntegrator* (sub block)

type of integration scheme is *rungeKutta2d* with *first* order scheme.

option *UpdateSequence* (sub block)

specifies the suquence *loop* of *UpdateSteps*.

option *UpdateStep* (sub blocks)

Consists of the names of the user defined updater blocks required to be evaluated during the time integration steps. Note that, the names are user-given in the inputfile. And the order of evaluation is important.

correct, bcBottom, bcTop, bcRight, bcWall, bcLeft, computeT, bcWallTemp, computeVelocity, computeViscosity, computeKinematicViscosity, computeThermalCoefficient, computeThermalDiffusivity, are the other updater blocks to evaluate and apply boundary conditions and thermophysical properties. These updater blocks are found in the *Supersonic cross flow over cylinder* example. Their usefulness will be briefed here in this lesson. *correct* corrects the values of any variables of interest to user specified limits. *bcBottom, bcTop, bcRight, bcWall, bcLeft* are the boundary conditions for conservative variables (here in this example, they are mass, three components of momentum, and energy density). *computeT* is for obtaining temperature from the conservative variables. *bcWallTemp* applies user specified temperature on the wall. 'computeVelocity' obtains the velocity from the conservative variables. *computeViscosity, computeKinematicViscosity, computeThermalCoefficient, computeThermalDiffusivity* are updater blocks used to compute thermophysical properties varying with temperature. Note that these blocks can be eliminated from the input file, if the properties are constant. *computeViscousFluxes* evaluates the viscous and thermal fluxes using the updater block shown above. *hyper* is *classicMuscl* updater block used to evaluate the convective fluxes from the hyperbolic part of the NS equations. Now the diffusion source is added to the existing convective sources using the *combiner2d* updater as shown below:

```
<Updater addSource>
  kind = combiner2d
  onGrid = domain
  in = [qnew, source]
  out = [qnew]
  indVars_qnew = ["rho", "mx", "my", "mz", "en"]
  indVars_source = ["sx","sy","sz","sen"]
```

```
  exprs = ["rho","mx+sx","my+sy","mz+sz","en+sen"]
</Updater>
```

option *syncVars_computeViscousFluxes* (optional attribute) This option synchronizes the diffusion fluxes in the variable *source*. All of the computed variables which need information from the cells in the adjacent partition (used in parallel computing) have to be synchronized using this option. Differentiation operators are used in *computeViscousFluxes* and the result is stored in *source*. Hence *source* is synChronized after calling *computeViscousFluxes*.

option *syncVars_bcRight* (optional attribute) used for the same reason as specified above.

option *dummyVariables_q* The dummy variables *qDummy1, qDummy2, qDummy3* are required for Runge-Kutta integration. Each of the integration variable should have a different *dummyVariables_ = [.,.,.]* option.

option *syncAfterSubStep*

qnew which has the updated values of the integration is synchronized at the end of integration.

5.9.3 Time Step Restriction

Time step restriction has to be added separately for hyperbolic and elliptic terms. The following three restriction are evaluated and added in the loop.

Hyperbolic restriction:

```
 <Updater timeStepRestriction>
   kind = timeStepRestrictionUpdater2d
   in = [q]
   onGrid = domain
   restrictions = [hyperbolic]
   <TimeStepRestriction hyperbolic>
     kind = hyperbolic
     model = eulerEqn
     cfl = CFL
     gasGamma = GAS_GAMMA
   </TimeStepRestriction>
</Updater>
```

Viscous diffusion time step restriction:

```
<Updater timeStepRestriction2>
  kind = timeStepRestrictionUpdater2d
  in = [kinematicViscosity]
  onGrid = domain
  restrictions = [quadratic]

  <TimeStepRestriction quadratic>
    kind = quadratic
    cfl = 0.5
  </TimeStepRestriction>
</Updater>
```

Thermal diffusion time step restriction:

```
 <Updater timeStepRestriction3>
   kind = timeStepRestrictionUpdater2d
   in = [thermalDiffusivity]
   onGrid = domain
```

```
   restrictions = [quadratic]
  <TimeStepRestriction quadratic>
    kind = quadratic
    cfl = 0.5
  </TimeStepRestriction>
</Updater>
```

In all of the above time step restrictions, cfl can be varied according to the problem.

5.9.4 An Example Simulation

The *Supersonic cross flow over cylinder* example of the USimHS demonstrates each of the concepts described above. Executing the *Supersonic cross flow over cylinder* input file within USimComposer and switching to the *Visualize* tab yields the plot shown in Fig. 5.1.

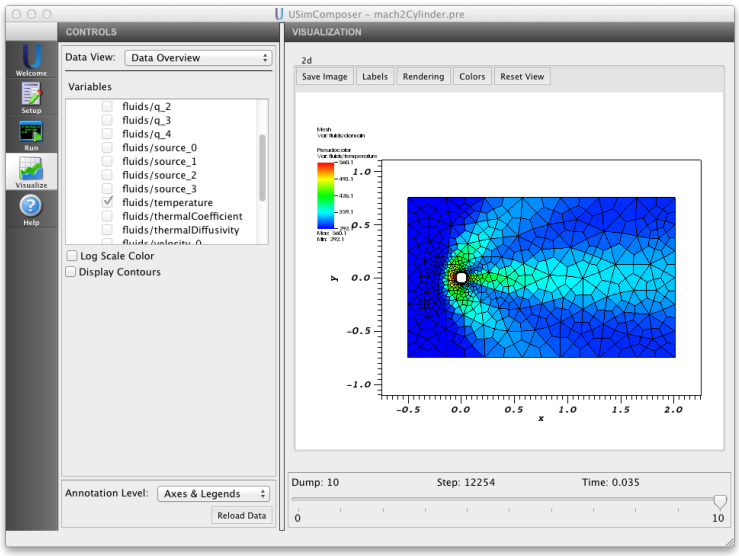


Fig. 5.1: Visualization tab in USimComposer after executing the input file for the tutorial.

5.10 Using USim to Solve Multi-Species Reactive Flows

Multi-species transport in cylindrical coordinates along with reactions is demonstrated in this example. A hypothetical gas consisting of three species *N2,N,O2* is considered here. Mass transport of individual species is solved along with Euler equations. Rate equations are solved in USim to obtain the change in concentration of the species due to chemical reactions. The rate of change of species is obtained using reaction rates and then added to the species transport as sources. Similarly the the change in energy is incorporated using energy of formation of each of the species. The related input file can be found in quickstart-blunt-body example of USimHS.

Contents

5.10.1 Multi-Species Mass Transport

Species mass transport fluxes are evaluated using the *classicMuscl* updater and Eigenvalues values of the Euler equation (see *Using USim to solve the Euler Equations*). The updater block is as given below.

```
<Updater hyperSpecies>
  kind = classicMuscl2d
  timeIntegrationScheme = none
  numericalFlux = localLaxFlux
  limiter = [minmod, minmod, minmod, none]
        variableForm = conservative
  cfl = CFL
  onGrid = domain
  in = [speciesDens, q, p, a]
  out = [speciesDensNew]
  equations = [speciesContinuity]
  sources = [multiSpeciesAxiSrc]

  <Equation speciesContinuity>
    useParentEigenvalues = true
    inputVariables = [speciesDens, q]
    kind = multiSpeciesSingleVelocityEqn
    numberOfSpecies = NSPECIES

    <Equation euler>
      kind = realGasEosEqn
      inputVariables = [q, p, a]
      numSpecies = NSPECIES
    </Equation>

  </Equation>

  <Source multiSpeciesAxiSrc>
     kind = multiSpeciesSym
     symmetryType = cylindrical
     numberOfSpecies = NSPECIES
  </Source>
</Updater>
```

This block uses *Equation* sub-block and a *Source* block. The mass fluxes are computed in the *Equation* block using the Eigenvalues of conservative variables q. Here q contains the conservative variables of *realGasEosEqn* equation. The equation of state is user specified, hence it requires pressure p and speed of sound a as inputs. The *Source* block computes the sources due to additional terms in cylindrical coordinates. The fluxes evaluated in both of the sub-blocks are added to the *out* variable *speciesDensNew*.

5.10.2 Mass Diffusion

Mass diffusion source for the multi-species can computed using the following block. This block uses refmanual-diffusion operator to compute the diffusion source, with the derivative specified as diffusion, *numScalars* is the number of species, isRadial is true for cylindrical coordinates, and the input variables are species density and diffusion coefficient *D*. *out* variable *diffSrc* contains the output.

```
<Updater computeDiffSrc>
  kind = diffusion2d
  onGrid = domain
  derivative = diffusion
  numScalars = NSPECIES
  coefficient = 1.0
  numberOfInterpolationPoints = 8
  isRadial = true
  in = [speciesDens,D]
  out = [diffSrc]

</Updater>
```

5.10.3 Rate of Change of Density

The following equation block shows the computation of rate of change of density three species due to two reactions. In order to use this block, a nodalArray variable consisting of the species densities *speciesDens* has to be initialized, another array to add the density change rates *speciesDensNew* should be declared, and an nodalArray with average temperature of the species. The two reactions are *N2 + N2 -> N + N + N2* and *N2 + O2 -> N + N + O2*. The equation block follow.

```
<Updater sourceUpdater>
  kind = equation2d
  onGrid = domain

  in = [speciesDens, temperature]
  out = [speciesDensNew]

  equations = [reactionSrc]

   <Equation reactionSrc>
     kind = reactionTableRhs
     outputEnergyRate = 0
     maxRate = 1.0e28
     species = [N2, N, O2]
     fileName = airReaction.txt
   </Equation>
</Updater>
```

The attributes used in the above block are

option *kind* (string)

Specifies the type of updater. Here it is *equation2d*

option *in* (nodalArray)

Names of the required nodalArray inputs. In this example, those are *speciesDens, temperature*

option *out*

Name of the output variable.

option *equations* (string)

Name of the equation. Here the name is *reactionSrc*

Attributes within the *Equation* sub blocks

option *kind* (string)

Type of equation. In this example, it is refmanual-reactionTableRhs

option *outputEnergyRate* (boolean)

option to compute reaction energy. it is chosen to be *false* in this demo.

option *maxRate* (real number)

An option to introduce artificial limit on the maximum value of reaction rate. It is useful in stabilizing the solution at reasonably small time steps.

option *species* (list of strings)

Names of the species considered.

option *fileName* (string)

Name of the refmanual-MultiSpeciesDataFiles.

Within the *Equation* Any number of reactions can be included by simply adding those to refmanual-MultiSpeciesDataFiles.

5.10.4 Chemical Energy

The energy of formation is computed using the following Updater block. The total energy of formation of all the species is added to get the mixture's energy. This *energyOfFormation* is added to internal energy in the energy equation. Hence during the initialization, the initial value of *energyOfFormation* should be computed using the initial densities of species and added to the energy density variable.

```
<Updater computeChemEn>
  kind = equation2d
  onGrid = domain

  in = [speciesDens,cpR,temperature]
  out = [chemEn]

  <Equation cp>
    kind = transportCoeffSrc
    coeff = chemicalEnergy
    numSpecies = NSPECIES
    fileName = airReaction.txt
  </Equation>
</Updater>
```

The attributes used in the above block are

option *kind* (string)

Specifies the type of updater. Here it is *equation2d*

option *in* (nodalArray)

Names of the required nodalArray inputs. In this example, those are *speciesDens, cpR, temperature*. specisDens is the number densities of all *species*, *cpR* is specific heats. *temperature* is average temperature of species. *speciesDens* and *cpR* are arrays with size equal to the number of species.

option *out*

Name of the output variable. contains the energy of formation. This variable will have only one component.

5.10.5 Addition of sources in time integrator

Two updaters are here, one for integrating the conservative variables *q* and species density *speciesDens*. The other is to integrate time rate of change of species due to reactions in using detailed explanation of the attributes is found in refmanual-multiUpdater.

```
<Updater rkUpdaterFluid>
  kind = multiUpdater2d
  onGrid = domain

  in = [q,speciesDens]
  out = [qnew,speciesDensNew]

  <TimeIntegrator rkIntegrator>
    kind = TIMEINTEGRATION_METHOD
    ongrid = domain
    scheme = TIMEINTEGRATION_SCHEME
  </TimeIntegrator>

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

  <UpdateStep boundaries>
     updaters = [bcOutflowSpecies, bcInflowSpecies, bcAbWallSpecies1, bcAbWallSpecies2, bcAbWallSpeci
     syncVars = [speciesDens,chemEn, temperature, surfTemp, q]
  </UpdateStep>

 <UpdateStep hyper>
 operation = "integrate"
  updaters = [computeChemEn, computeTemperature, temperatureCorrector, computeCpAvg, computeMwAvg, co
  syncVars = [temperature, p,a,velocity,viscousSource,qnew,speciesDensNew]
</UpdateStep>
```

Boundary conditions are applied on species using *bcOutflowSpecies, bcInflowSpecies, bcAbWallSpecies1, bcAbWallSpecies2, bcAbWallSpecies3*. Energy of formation and temperature are computed using *computeChemEn, computeTemperature*. Remember that, energy of formation is required to compute temperature. Boundary conditions on temperature are then applied using *bcFluidTempAxis, bcFluidTempWall, bcFluidTempInflow, bcFluidTempCopy, bcSurfTemp* updaters. Finally boundary conditions are applied to *bcOutflow, bcInflow, bcAxis, bcAbWall1, bcAbWall2, bcAbWall3* conservative variables.

Energy of formation, *computeChemEn* is evaluated and then temperature is computed using the updater *computeTemperature*. The properties are evaluated using *computeCpAvg* for average value of constant pressure specific heat of species, *computeMwAvg* for the average molecular weight, and *computeGammaAvg* for average gamma of the mixture. Viscosity an thermal conductivity are evaluated using *computeViscosity*, *computeThermalCoefficient*. The total pressure of the gas, electron pressure and heavy particle pressure are computed using *computeGasPressure*, *computeElectronPressure*, *computeHvpPressure* updaters respectively. The pressure is copied into the ghost layers using the boundary condition updaters *bcPressureWall1*, *bcPressureWall2*, *bcPressureWall3*. Pressure boundary condition is applied for refmanual-realGasEosEqn. Then *computeSoundSpeed* is evaluated. *computeKinematicViscosity*, *computeThermalDiffusivityFluid* are evaluated to obtain kinematic viscosity and thermal diffusivity. These are required for time step restriction based on diffusion. Velocity of fluid is evaluated using *computeVelocity* and then viscous source is computed *computeViscousSource*. Convective fluxes of conservative variable are computed in 'hyper' and

stored in *qnew*. Convective fluxes of species are stored in *speciesDensNew*. Viscous source is added to *qnew* using *addViscousSource*.

The change in density due to reactions is added to species density and integrated in the following Updater. The updater *sourceUpdater* evaluates and adds the rate of change of density to the species equations. The resulting equations are integrated using refmanual-localOdeIntegrator method.

```
<Updater sourceUpdater>
    kind = localOdeIntegrator2d

    onGrid = domain

    in = [speciesDens, temperature]

    out = [speciesDensNew]
    integrationScheme = bulirschStoer
    relativeErrorTolerance = 1.0e9
    equations = [reactionSrc]
     <Equation reactionSrc>
       kind = reactionTableRhs
       outputEnergyRate = 0
       maxRate = MAXRATE
       species = [N2, N, O2, O, NO, NO_p1, e, Ca, Na, K]
       fileName = airReaction.txt
    </Equation>

  </Updater>
```

5.10.6 An Example Simulation

The *Blunt body reentry* example of the USimHS demonstrates each of the concepts described above using 7 species model of air. The considered 7 species are $N2, N, O2, O, NO, NO+, e$. Executing the *Blunt body reentry* input file within USimComposer and switching to the *Visualize* tab yields the plot shown in Fig. 5.2.

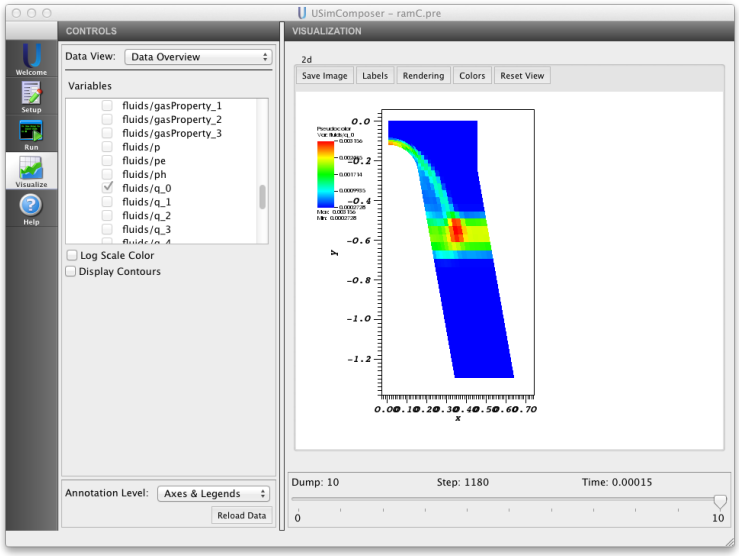


Fig. 5.2: Visualization tab in USimComposer after executing the input file for the tutorial.

5.11 Advanced Time-Stepping Methods in USim

USim implements methods that allows a simulation to be advanced on the timestep associated with the inviscid equations (e.g. the Euler equations), rather than that associated with (for example) the Navier-Stokes equations or complex reaction chemistry. These methods typically allow a speed up $\sim \sqrt{N}$ where N is the ratio of the inviscid to the (for example) viscous timesteps.

Contents

5.11.1 Subcycling for Complex Chemsitry

The *Flow over cylindrical rod* example of the USimHS package of USim implements methods for accelerating chemical reaction rates through sub-cycling. An example updater block that implements these methods is shown below:

```
<Updater chemistryUpdater>
  kind = stsUpdater2d
  onGrid = domain
  timeIntegrationScheme = zerothOrder
  updaters = [sourceUpdater]

  integrationVariablesIn = [qSpecies]
  integrationVariablesOut = [qSpeciesNew]
  timeStepRestrictions = [reactionrateDT]
  dummyVariables_qSpecies = [dummySpecies1, dummySpecies2, dummySpecies3, dummySpecies4, dummySpecies
</Updater>
```

The meanings for the input blocks in this updater are as follows:

`kind` (string)

> Specifies the *stsUpdater2d* updater, which tells USim to advance the solution vector *super time stepping* methods in 2d.

`timeIntegratioScheme` (string)

> The order of accuracy of the time-integration scheme that to be used. Here we have specified a sub-cycling scheme using *zerothOrder*.

`updaters` (string)

> The list of updaters that are used to perform the time integration. This is equivalent to the *loop* field of the update-sequence that was discussed in USimBase Tutorial on Euler Equations, input file, with the exception that updater are called directly, rather than through an update-step. In this case, we call one updater *sourceUpdater*, which calculates chemical reaction rates.

`integrationVariablesIn` (string)

> Specifies the data structure that contains the conserved state at time t, here this is *qSpecies*.

`integrationVariablesOut` (string)

> Specifies the data structure that contains the conserved state updated to time $t + \Delta t$, here this is *qSpecies-New*

`timeStepRestrictions` (string)

Specifies the *dynVector* data structure that holds the timestep associated with the chemical reaction. Here, this is *reactionrateDT*, which is defined through:

```
<DataStruct reactionrateDT>
  kind = dynVector
  numComponents = 1
  writeOut = false
</DataStruct>
```

The timestep restriction is calculated using the *dynVectorOperator* updater:

```
<Updater fixDt>
  # to fix dt to specified value
  kind = dynVectorOperator
  in = [ ]
  out = [reactionrateDT]
  onGrid = [domain]

  maxDt = MAX_DT

  exprs = ["maxDt"]
</Updater>
```

This updater sets the *dynVector* called *reactionrateDT* to hold the value specified by *MAX_DT*.

`dummyVariables_q` (string)

A list of dummy variables that are used by the *multiUpdater* to perform the integration. These are substituted for the *qSpecies* vector and should have the same number of components. The *stsUpdater* requires five dummy vectors, irrespective of the choice of *timeIntegrationScheme*.

5.11.2 Super Time Stepping for Viscous Operators

The *Flow over cylindrical rod* example of the USimHS package of USim implements methods for accelerating chemical reaction rates through sub-cycling. An example updater block that implements these methods is shown below:

```
<Updater stsUpdater>
  kind = stsUpdater2d
  onGrid = domain
  timeIntegrationScheme = secondOrder

  updaters = [bcAxis, bcInflow, bcWall, bcOutflow,bcFreeflow, \
                    computeViscousSource,setViscSource]

  integrationVariablesIn = [q]
  integrationVariablesOut = [qnew]
  timeStepRestrictions = [diffDT1,diffDT2]
  dummyVariables_q = [dummy1, dummy2, dummy3, dummy4, dummy5]
</Updater>
```

The meanings for the input blocks in this updater are as follows:

`timeIntegratioScheme` (string)

The order of accuracy of the time-integration scheme that to be used. Here we have specified a second order super time stepping scheme using *secondOrder*.

`timeStepRestrictions` (string)

Specifies the *dynVector* data structure that holds the timestep associated with the chemical reaction. Here, this is *[diffDT1,diffDT2]*, which are defined through:

```
<DataStruct diffDT1>
  kind = dynVector
  numComponents = 1
  writeOut = false
</DataStruct>

<DataStruct diffDT2>
  kind = dynVector
  numComponents = 1
  writeOut = false
</DataStruct>
```

These timestep restriction are calculated using the *getTimeStepRestriction* updater:

```
 <Updater getDiffDT1>
  kind = getTimeStepUpdater2d
  in = [kinematicViscosity]
  out = [diffDT1]
  onGrid = domain
  restrictions = [quadratic]

  <TimeStepRestriction quadratic>
    kind = quadratic
    cfl = CFL
  </TimeStepRestriction>
</Updater>

 <Updater getDiffDT2>
  kind = getTimeStepUpdater2d
  in = [kinematicConductivity]
  out = [diffDT2]
  onGrid = domain
  restrictions = [quadratic]

  <TimeStepRestriction quadratic>
    kind = quadratic
    cfl = CFL
  </TimeStepRestriction>
</Updater>
```

These updaters set the *dynVector* called *diffDT1* and *diffDT2* to hold the timesteps associated with the viscous and conductivity operators in the Navier-Stokes equations respectively.

5.11.3 An Example Simulation

The *Flow over cylindrical rod* example of the USimHS package demonstrates each of the concepts described above. Executing the *Flow over cylindrical rod* input file within USimComposer and switching to the *Visualize* tab yields the plot shown in Fig. 5.3.

5.12 Running USim from the Command Line

The following sections describe how to run USim from the command line.

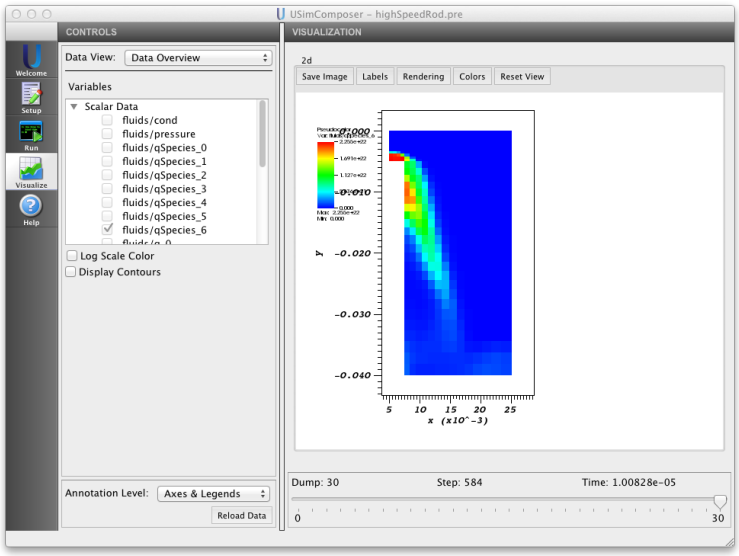


Fig. 5.3: Visualization tab in USimComposer after executing the input file for the tutorial.

5.12.1 PATH Definitions

The following definitions will be used for the remainder of this section.

On Mac:

```
<ULIXES_BIN_DIR>=/Applications/USimComposer.app/Contents/engine/bin
<ULIXES_LIB_DIR>=/Applications/USimComposer.app/Contents/engine/lib
<ULIXES_SHARE_DIR>=/Applications/USimComposer.app/Contents/engine/share
```

On Linux (assuming you have chosen /usr/local/USimComposer as your default installation directory):

```
<ULIXES_BIN_DIR>=/usr/local/USimComposer/Contents/engine/bin
<ULIXES_LIB_DIR>=/usr/local/USimComposer/Contents/engine/lib
<ULIXES_SHARE_DIR>=/usr/local/USimComposer/Contents/engine/share
```

On Windows (e.g. 64-bit)

```
<ULIXES_BIN_DIR>=C:\Program Files\Tech-X (Win64)\USim 6.0\Contents\engine\bin
<ULIXES_SHARE_DIR>=C:\Program Files\Tech-X (Win64)\USim 6.0\Contents\engine\share
```

5.12.2 Running a USim Pre File

Your USim distribution package contains two executable programs for running USim, one for serial computations (`ulixesser`) and another for parallel computations (`ulixes`). Both versions of the USim executables are located in the `ULIXES_BIN_DIR` directory.

Command Line Features

If USim is run from the command line, input file and runtime options are specified as command line options. USimComposer sets up your environment prior to running USim, and therefore you must set up your environment when running USim from the command line. You will need to modify the environment variables LD_LIBRARY_PATH and PATH.

If you are using the bash shell, you will need to modify LD_LIBRARY_PATH by adding the following line to your .bashrc:

```
export LD_LIBRARY_PATH=<ULIXES_LIB_PATH>:$LD_LIBRARY_PATH
```

Instead, if you are using the csh shell (or one of its variants such as tcsh), you will need to add the following line to your .cshrc:

```
setenv LD_LIBRARY_PATH <ULIXES_LIB_DIR>:$LD_LIBRARY_PATH
```

Similarily, to set your PATH in the bash shell, you will need to add the following line to your .bashrc:

```
export PATH=<ULIXES_BIN_DIR>:$PATH
```

whereas in csh/tcsh, you will need to add the following line to your .cshrc:

```
setenv PATH <ULIXES_BIN_DIR>:$PATH
```

Note that any changes you make to your .bashrc/.cshrc do not take effect until the next time you log in, so after modifying your startup file, you must execute the following command in your current shell, but will not need to do it in the future:

```
source ~/.bashrc    (for bash)
```

or

```
source ~/.cshrc     (for csh/tcsh)
```

Order of Parameter Precedence

If a parameter is both set within the input file and specified on the command line, the command line parameter value takes precedence. The command line override enables you to configure an input file with default values while exploring alternative parameter settings from the command line. From the command line, you can quickly change simulation run lengths, dimensionality, output timing, etc.

Examples of Running USim from the Command Line

In these examples, it is assumed that you are either in the directory in which the **`ulixesser`** is installed or you have added the appropriate directory to your shell path.

Command Line Options

The first step in running USim on the command line is to preprocess the input file. Run the preprocessor (txpp.py) on the .pre file as follows

```
./txpp.py filaname.pre
```

The output will be a file called filename.in. The .in file is then used with the ulixes executable as follows To use multiple options, the command line syntax is:

```
./ulixesser -i filename.in [-o prefix_name] [-r num]
```

in which `./ulixesser` is used to run a serial computation. See *Serial Computation* for details about serial computation. See the *Parallel Computation* for details of command line invocation with parallel computation scripts.

Commonly used options that you can specify on the command line include:

-i filename.in Read input from file named *filename*.

For example:

```
./ulixesser -i sodShock.in
```

-o prefix_name Base names of output files on the text string *prefix_name*.

For example, if you want output files named `newforwardFacingStep` rather than `forwardFacingStep`, use:

```
./ulixesser -i forwardFacingStep.in -o newforwardFacingStep
```

-r num Restart USim from dump *num*.

For example, if you want to restartforwardFacingStep using the output dumped at time step 50, use:

```
./ulixesser -i sodShock.in -r 50
```

More details on how to restart USim are given in *Restarting a USim Simulation*

5.12.3 Serial Computation

The USim executable for use in serial computation is named **ulixesser**. Except as noted, the explanations and tutorials within the *USim In Depth* and USim-Quick-Start demonstrate USim usage for serial computations. Here is an example of USim command line invocation using a pre file named `myfile.pre` with corresponding .in file `myfile.in`. By default, the output files for this example would be named using the format `myfile.out`.

```
ulixesser -i myfile.in
```

Note: The above invocation line assumes you added <ULIXES_BIN_DIR> to your PATH, as noted in the *Command Line Features* section.

Note: When running USim via USimComposer, command line options are not directly available, however `-i` and `-o` command line options described in this document are implicit; that is, these options are automatically invoked when running USimComposer.

5.12.4 Parallel Computation

The USim executable for use in parallel computation is named **ulixes**. This section explains use of the USim executable program for parallel computations.

USim for parallel computations requires the Message Passing Interface (MPI). For information about mpi for use with USim, see *Running USim with mpiexec*.

Parallel Computation Scripts

Running USim with **MPI** or Parallel Queuing Systems requires use of different shell scripts to enable invocation of the USim executable as discussed in the following sections.

Running USim with mpiexec

USim comes bundled with a distribution of **Open MPI**, which must be used to run USim in parallel (in other words, even if you have an installation of **Open MPI** on your machine, you should use the one included with USim).

In order to run USim in parallel via the command line, you must first add the <ULIXES_BIN_DIR> to your PATH, as noted in the *Command Line Features* section.

To run USim in parallel, execute the following command:

```
mpiexec -np <#> ulixes -i filename.in
```

in which <#> is the number of processors, **ulixes** is the executable program for parallel computations, and `filename.in` is the name of the USim .in file (which must be in the current directory, or must be specified by a full path).

Following **mpiexec**, but before <ULIXES_BIN_DIR>, you can specify a variety of **mpiexec** options. For more information about **mpiexec**, including the complete list of options, see a man page or other documentation for **mpiexec**.

Following **ulixes**, you can specify a variety of USim options. For a list of commonly used options, see *Command Line Features*.

USim automatically adjusts its decomposition to match the number of processors it is given.

Running USim with Parallel Queuing Systems

Parallel queuing systems, such as LoadLeveler and PBS, require the submission of a shell script with embedded comments that the systems interpret. Here is an example of a basic shell script for a PBS-based system:

```
#PBS -N NDS_ulixes
#PBS -l nodes=2:ppn=2
cd /directory/containing/your/input/file
mpiexec -np 4 ulixes -i your_input_file.in
```

Running USim in Parallel under Windows

To run USim in parallel on a Windows system, bring up a DOS window. From the command prompt,

```
<ULIXES_BIN_DIR>\mpiexec.exe -np 2 <ULIXES_BIN_DIR>\ulixes.exe -i your_input_file.in
```

Running SEACAS partitioner on Linux

To decompose a meshfile for a number of processors use the provided SEACAS partitioner script (decomp). For example, if you plan to use 8 processors, from the command line run:

```
<ULIXES_BIN_DIR>/decomp -p 8 meshfile.g
```

For more information about the decomp script, run the script with the help option:

```
<ULIXES_BIN_DIR>/decomp -h
```

5.13 Restarting a USim Simulation

Restart in USim is performed on the command line with the *-r* option. To restart from dump 5 in parallel enter:

```
mpiexec -np 4 ulixes -i sodshock.in -r 5
```

Restart in USim can be tricky so for large problems it's best to first test restart to make sure that you've dumped out all the required data and furthermore that you've included the correct updater in the *restoreOnly* loop and entered the proper synchronization during the restoreOnly loop. The restore loop should include all the entities that are generated along with anything in the initialization loop that does not overwrite the data you are reading in. An example of the *restoreOnly* loop and the associated *UpdateStep* is:

```
<UpdateStep generateStep>
  updaters = [generateOpen, generateWall, generateInflow, generateNoInflow]
  syncVars = [q]
</UpdateStep>

restoreOnly=[generateStep]
```

Geometry is regenerated at the beginning of every simulation regardless of whether it is restarted or no. Secondly, all DataStruct write their data out to file by default unless writeOut=false (or 0). All the DataStructs that have been written out will be written in. It is best not to modify the writeOut option before a simulation is restarted since USim only reads in values whose writeOut value is true (the default). The example below is a fairly typical input file using an unstructured grid. The input file uses the *entityGenerator* which is typically called only once during the initialization step. Restore will work properly in this case only if the entity generators are also called with the *restoreOnly* option. The startOnly loop is not called on restart so restoreOnly acts as it's substitute for restart. The reason for this is simple, during restart we do not want to initialize our DataStruct since they are being read in, however we do want to initialize geometric quantities. One final important point to make, in addition to reading in the proper data synchronization must be performed across the DataStructs that are read in. In the example below our step *generateStep* initialized entities and synchronizes the *DataStruct q*.

An example of a USim input file that illustrates the concepts described above is given below:

```
<Component fluids>
  kind = updaterComponent

  <Grid domain>
    kind = unstructured

    writeGeom = true
    writeConn = true
    ghostLayers = 2
    decomposeHalos = false

    <Creator ctor>
      kind = gmsh
      ndim = 2
      file = rampgeom4.msh
    </Creator>
  </Grid>

  <DataStruct q>
    kind = nodalArray
    numComponents = 5
    onGrid = domain
  </DataStruct>

  <DataStruct dummy1>
    kind = nodalArray
    numComponents = 5
    onGrid = domain
    writeOut = false
  </DataStruct>
```

```
<DataStruct qnew>
  kind = nodalArray
  numComponents = 5
  onGrid = domain
  writeOut = false
</DataStruct>

.
.
.

<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>

<Updater generateWall>
  kind = entityGenerator2d
  onGrid = domain
  newEntityName = wallBoundary
  onEntity = ghost
  <Function mask>
    kind = exprFunc
    exprs = ["if( (x>0.01) and (y<0.35),1.0,-1.0)"]
  </Function>
</Updater>

<Updater generateInflow>
  kind = entityGenerator2d
  onGrid = domain
  newEntityName = inflowBoundary
  onEntity = ghost
  <Function mask>
    kind = exprFunc
    exprs = ["if(x<0.01,1.0,-1.0)"]
  </Function>
</Updater>

<Updater generateNoInflow>
  kind = entityGenerator2d
  onGrid = domain
  newEntityName = noInFlowBoundary
  onEntity = ghost
  <Function mask>
    kind = exprFunc
    exprs = ["if(y<0.0,1.0,-1.0)"]
  </Function>
</Updater>

.
.
.
```

```
  <UpdateStep initStep>
    updaters = [initdn]
  </UpdateStep>

  <UpdateStep generateStep>
    updaters = [generateOpen, generateWall, generateInflow, generateNoInflow]
    syncVars = [q]
  </UpdateStep>

  .
  .
  .

  <UpdateSequence seq>
    startOnly = [initStep, generateStep]
    restoreOnly = [generateStep]
    loop = [boundaryStep, hyperStep, copyStep]
  </UpdateSequence>

</Component>
```

5.14 Running on a Remote Host

USim allows the user to run on a remote host if desired. This is potentially beneficial for several reasons, including runs with large data sets, running on a large cluster, or shared resources.

5.14.1 Setting up a Remote Host

To use the remote capability, one must install USim on both the local and remote hosts. Only 64-bit Linux platforms are supported for the remote host, so follow the Linux installation instructions. The local host can be any of the supported operating systems and with the proper license a user can switch back and forth between local and remote operations from the same local USim installation.

Once a Linux version of USim is installed on the remote host, note the location of the installation (e.g. /path/to/usim/installation). On the local machine:

- Open USimComposer.
- Open the Settings Dialog (choose *Tools* -> *Settings* on Windows/Linux or *Preferences* under the USimComposer menu in Mac OS).
- Click on the *Host Settings* tab on the left.
- Click on the *Add* button under the Profiles pane.
- Under *General* :
  - Choose the host profile name, enter your user name, and host address.
  - Optionally Enter your password if you want to test the connection. You will be prompted for your password later as well, when needed. Note that your password will not be saved. Dynamic (key-fob-based) passwords are also supported.
  - Click *Apply* and switch to *Paths*.

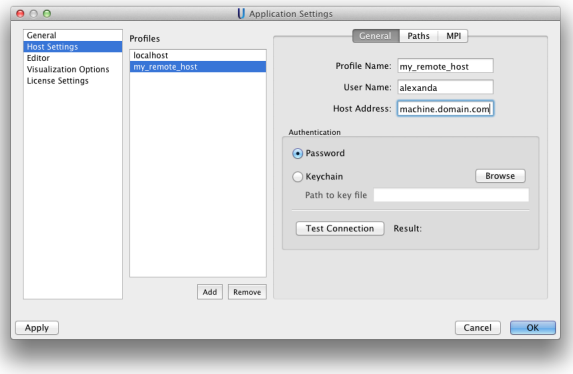


Fig. 5.4: The Host Settings Window showing the general settings

5.14.2 Setting the Remote Host Paths

Under *Paths*:

- Choose the path to your workspace directory on the remote host. This directory is where all template data will be copied, and where generated data will be saved. For example:

```
/path/to/user/workspace
```

- Enter the path to the USimComposer installation directory. For example:

```
/path/to/usim/installation
```

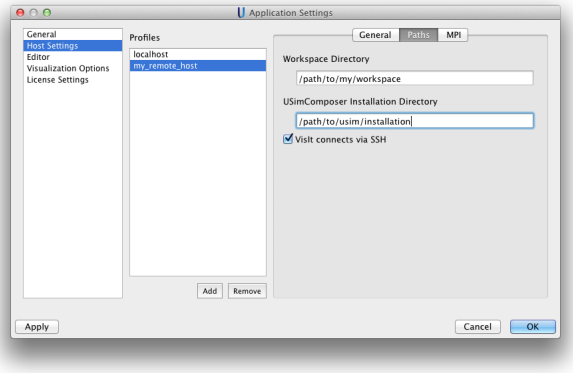


Fig. 5.5: The Host Settings Window showing the path settings

- Click *Apply* and then *OK*.

At this point, the remote host is set up, and you may use either this remote host or the local host.

5.14.3 Using the Remote Host

When asked by USimComposer to choose where to save your files (from new from template or from existing examples), you can choose either local or remote host under *Host*. From then on, you will be on the chosen host.

Similarly, if you want to open an existing file of your own, you can elect to *Open Runspace* under the *File* menu. From here, select either the local or remote host under *Host*.

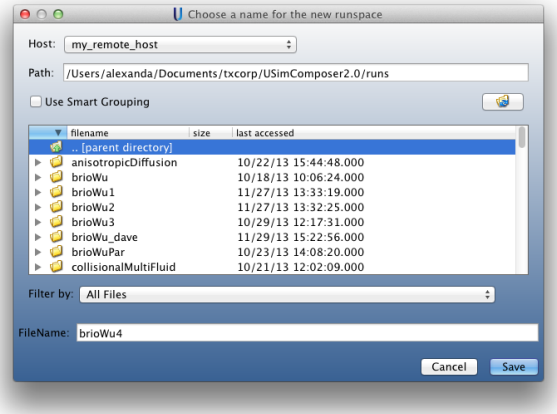


Fig. 5.6: Choosing the remote host

5.14.4 Troubleshooting

If you have problems with remote visualization, please make sure that your configuration is set up as required by remote VisIt: ports 5600-5609 are open on the client and the server machine.

- USim © 2011-2018 Tech-X Corporation. All rights reserved.

For USim licensing details please email sales@txcorp.com. All trademarks are the property of their respective owners. Redistribution of any USim™ simulation input file code examples from the USim Document Set, including the USim In Depth and USim Reference, is allowed provided that this copyright statement is also included with the redistribution.