# Uintah User Guide

Jim Guilkey, Todd Harman, Justin Luitjens,
John Schmidt, Jeremy Thornock, J. Davison de St. Germain,
Siddharth Shankar, Joseph Peterson, Carson Brownlee,
Charles Reid, Tony Saad, Jacqueline Beckvermit,
Alan Humphrey, Brad Peterson

Version 2.1

# Contents

# Chapter 1

# Overview of Uintah

## 1.1 The Center for the Simulation of Accidental Fires and Explosions (C-SAFE)

### 1.1.1 Center History

The Uintah software suite was created by the Center for the Simulation of Accidental Fires and Explosions (C-SAFE). C-SAFE was originally created at the University of Utah in 1997 by the Department of Energy's Accelerated Strategic Computing Initiative's (ASCI) Academic Strategic Alliance Program (ASAP). (ASCI has since been renamed to the Advanced Simulation and Computing (ASC) program.)

#### Center Objective

C-SAFE's primary objective has been to provide a software system in which fundamental chemistry and engineering physics are fully coupled with nonlinear solvers, visualization, and experimental data verification, thereby integrating expertise from a wide variety of disciplines. Simulations using the Uintah software can help to better evaluate the risks and safety issues associated with fires and explosions in accidents involving both hydrocarbon and energetic materials.

#### Target Simulation

The Uintah software system was designed to support the solution of a wide range of highly dynamic physical processes using a large number of processors. However, our specific target simulation has been the heating of an explosive device placed in a large hydrocarbon pool fire and the subsequent deflagration explosion and blast wave (Figure 1.1). The explosive device is a small cylindrical steel container (4" outside diameter) filled with plastic bonded explosive (PBX-9501). Convective and radiative heat fluxes from the fire heat the outside of the container and subsequently the PBX. After some amount of time the critical temperature in the PBX is reached and the explosive begins to rapidly decompose into a gas. The solid→gas reaction pressurizes the interior of the steel container causing the shell to rapidly expand and eventually rupture. The gaseous products of reaction form a blast wave that expands outward along with pieces of the container and any unreacted PBX. The physical processes in this simulation have a wide range in time and length scales from microseconds and microns to minutes and meters. Uintah was designed as a general-purpose fluid-structure interaction code that can simulate not only this scenario but a wide range of related problems.

Complex simulations such as this require both immense computational power and complex software. Typical simulations include solvers for structural mechanics, fluids, chemical reactions, and material models. All of these aspects must be integrated in an efficient manner to achieve the scalability required to perform these simulations. The heart of Uintah is a sophisticated computational framework that can integrate multiple simulation components, analyze the dependencies and communication patterns between them, and efficiently execute the resulting multi-physics simulation. Uintah also provides mechanisms for automating load-balancing, checkpoint/restart, and parallel I/O. The Uintah core was designed to be general, and is appropriate for use in a wide range of PDE algorithms based on structured (adaptive) grids and particle-in-cell algorithms.

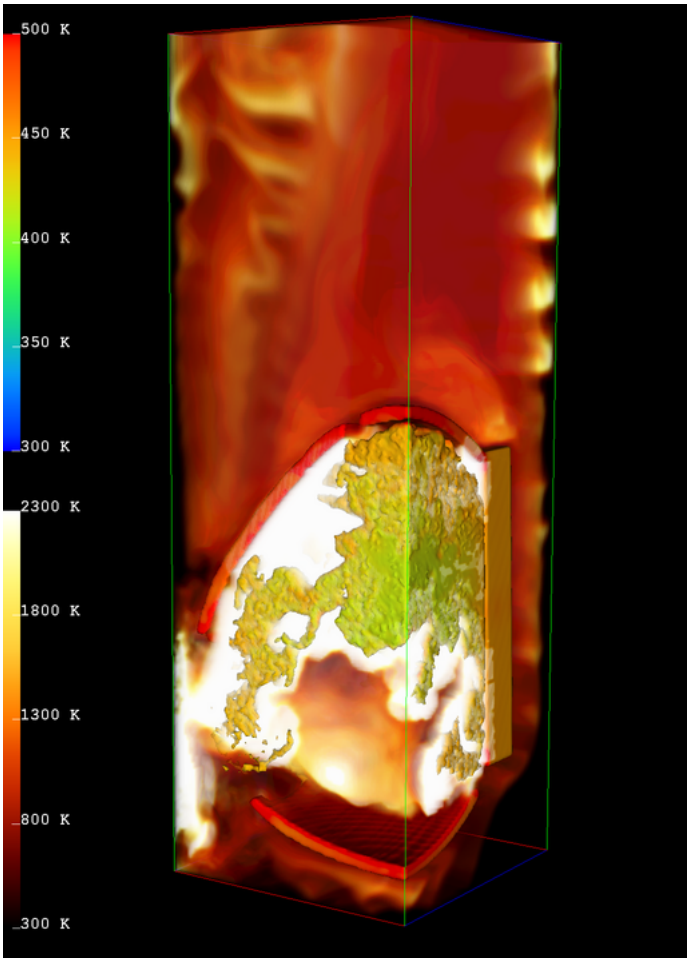



Figure 1.1: Target Simulation - Fire-Container-Explosion.

## 1.2 Uintah Software

The Uintah Computational Framework (also referred to as Uintah or the UCF) consists of a set of software components and libraries that facilitate the solution of Partial Differential Equations (PDEs) on Structured AMR (SAMR) grids using up to hundreds to thousands of processors.

One of the challenges in designing a parallel, component-based and multi-physics application is determining how to efficiently decompose the problem domain. Components, by definition, make local decisions. Yet parallel efficiency is only obtained through a globally optimal domain decomposition and scheduling of computational tasks. Typical techniques include allocating disjoint sets of processing resources to each component, or defining a single domain decomposition that is a compromise between the ideal load balance of multiple components. However, neither of these techniques will achieve maximum efficiency for complex multi-physics problems.

Uintah uses a non-traditional approach to achieving parallelism by employing an abstract task graph representation to describe computation and communication. The task graph is an explicit representation of the computation and communication that occur in the coarse of a single iteration of the simulation (typically a timestep or nonlinear solver iteration). Uintah components delegate decisions about parallelism to a scheduler component by using variable dependencies to describe communication patterns and characterizing computational workloads to facilitate a global resource optimization. The task graph representation has a number of advantages, including efficient fine-grained coupling of multi-physics components, flexible load balancing mechanisms and a separation

of application concerns from parallelism concerns. However, it creates a challenge for scalability which we overcome by creating an implicit definition of this graph and representing it in a distributed fashion.

The primary advantage of a component-based approach is that it facilitates the separate development of simulation algorithms, models, and infrastructure. Components of the simulation can evolve independently. The component-based architecture allows pieces of the system to be implemented in a rudimentary form at first and then evolve as the technologies mature. Most importantly, Uintah allows the aspects of parallelism (schedulers, load-balancers, parallel input/output, and so forth) to evolve independently of the simulation components. Furthermore, components enable replacement of computation pieces without complex decision logic in the code itself.

Please see the Developers Guide (`http://www.uintah.utah.edu/trac/chrome/site/UintahAPI.pdf`) for more information about the internal architecture of Uintah.

### 1.2.1 Software Ports

Uintah has been ported and runs well on a number of operating systems. These include Linux, Mac OSX, Windows, AIX, and HPuX. Simulating small problems is perfectly feasible on 2-4 processor desktops, while larger problems will need 100s to 1000s of processors on large computer clusters.

### 1.2.2 Uintah Software History

The UCF was orginally built on top of the SCIRun Problem Solving Environment. SCIRun provided a core set of software building blocks, as well as a powerful visualization package. While Uintah continues to use the SCIRun core libraries, Uintah's use of the SCIRun PSE has been retired in favor of using the VisIt visualization package from LLNL.

# Chapter 2

# Using Uintah

Several executable programs have been developed using the Uintah Computational Framework (UCF). The primary code that drives the components implemented in Uintah is called `sus`, which stands for Standalone Uintah Simulation. The existing components were originally developed to solve a complex fluid structure problem involving a container filled with an explosive enveloped in a fire.

The code models the fire and the subsequent heat transfer to the container followed by the resultant container deformation and ultimate rupture due to the ignition and burning of the explosive material all running on thousands of processors requiring thousands of hours of computer time and hundreds of gigabytes of data storage. Although Uintah was developed originally to solve this complicated multi-physics problem, the general nature of the algorithms and the framework have allowed researchers to use the code to investigate a wide range of problems. The framework is general purpose enough to allow for the implementation of a variety of implicit and explicit algorithms on structured grids. In addition, particle based algorithms can be implemented using the native particle support found in the framework.

This code leverages the task based parallelism inherent in the UCF to implement several time stepping algorithms for structural mechanics, fluid dynamics, and fluid structure interactions. What follows is a description of using `sus` within the realm of structural mechanics, fluid mechanics, and structure-fluid interactions.

## 2.1 Installing the Uintah Software

For information on downloading the Uintah software package (via tarball or SVN), and how to setup (configure) and build (make) the system, please refer to the Uintah Installation Guide.

## 2.2 Mechanics of Running sus

For single processor simulations, the `sus` executable (Standalone Uintah Simulation) is run from the command line prompt like this:

```
sus input.ups
```

where `input.ups` is an XML formatted input file. The Uintah software release contains numerous example input files located in the `src/StandAlone/inputs/UintahRelease` directory.

For multiprocessor runs, the user generally uses `mpirun` to launch the code. Depending on the environment, batch scheduler, launch scripts, etc, `mpirun` may or may not be used. However, in general, something like the following is used:

```
mpirun -np num_processors sus -mpi input.ups
```

`num_processors` is the number of processors that will be used. The input file must contain a patch layout that has at least the same number (or greater) of patches as processors specified by a number following the -np option shown above.

In addition, the `-mpi` is optional but often times necessary if the mpi environment is not automatically detected from within the sus executable.

Uintah provides for restarting from checkpoint as well. For information on this, see Section 2.6, which describes how to create checkpoint data, and how to restart from it.

## 2.3 Uintah Problem Specification (UPS)

The Uintah framework uses XML like input files to specify the various parameters required by simulation components. These Uintah Problem Specification (.ups) files are validated based on the specification found in `src/StandAlone/inputs/UPS_SPEC/ups_spec.xml` (and its sibling files).

The application developer is free to use any of the specified tags to specify the data needed by the simulation. The essential tags that are required by Uintah include the following:

```
<Uintah_specification>

<SimulationComponent>

<Time>

<DataArchiver>

<Grid>
```

Individual components have additional tags that specify properties, algorithms, materials, etc. that are unique to that individual components. Within the individual sections on MPM, ICE, MPMICE, Arches, and MPMArches, the individual tags will be explained more fully.

The sus executable verifies that the input file adheres to a consistent specification and that all necessary tags are specified. However, it is up to the individual creating or modifying the input file to put in physically reasonable set of consistent parameters.

## 2.4 Simulation Components

The input file tag for SimulationComponent has the `type` attribute that must be specified with either `mpm, mpmice, ice, arches,` or `mpmarches,` as in:

```
<SimulationComponent type = "mpm" />
```

## 2.5 Time Related Variables

Uintah components are time dependent codes. As such, one of the first entries in each input file describes the time-stepping parameters. An input file segment is given below that encompasses all of the possible parameters. The function of each of these parameters is described below.

```
<Time>
    <maxTime>            1.0         </maxTime>
    <initTime>           0.0         </initTime>
    <delt\_min>           0.0         </delt\_min>
    <delt\_max>           1.0         </delt\_max>
    <delt\_init>          1.0e-9      </delt\_init>
    <max\_delt\_increase>  2.0         </max\_delt\_increase>
    <timestep\_multiplier>1.0         </timestep\_multiplier>
    <max\_Timestep>       100         </max\_Timestep>
    <end\_on\_max\_time\_exactly>true    </end\_on\_max\_time\_exactly>
</Time>
```

The following fields are required:

- maxTime - how long in physical time to run the simulation for
- initTime - what time to begin the simulation at
- delt_min - the smallest timestep the simulation will take
- delt_max - the largest timestep the simulation will take
- timestep_multiplier - multiplies the timestep by this number (before adjusting to min or max timestep)

The following fields are optional:

- delt_init - The timestep to take initially (assuming it's less than the one computed by the simulation)
- initial_delt_range - The period of time to use the delt_init (default = 0)
- max_delt_increase - Maximum amount to multiply the previous delt by (if the newly computed delt is greater than the previous one)
- max_iterations - The number of timesteps to run the simulation for (even on a restart)
- max_Timesteps - The timestep number to end the simulation on (not usually used with max_iterations)
- override_restart_delt - On a restart, use this delt instead of the most-recently-used delt.
- clamp_timesteps_to_output - Sync the delt with the DataArchiver - when an output timestep occurs, reduce the delt to have the time land on the timestep interval (default = false)
- end_on_max_time_exactly - clamp the delt such that the last timesteps end on what was specified in maxTime (default = false)

A word about timesteps: In general, the timestep (delt) is computed at various stages within a timestep, and the smallest one is used, unless it needs to raise the delt to the delt_min.

## 2.6 Data Archiver

The Data Archiver section specifies the directory name where data will be stored and what variables will be saved and how often data is saved and how frequently the simulation is checkpointed.

The `<filebase>` tag is used to specify the directory name and by convention, the `.uda` suffix is attached denoting the "Uintah Data Archive".

Data can be saved based on a frequency setting that is either based on time intervals;
`<outputTimestepInterval> integer_number_of_steps </outputTimestepInterval>`
or timestep intervals;
`<outputInterval> floating_point_time_increment </outputInterval>`

Each simulation component specifies variables with label names that can be specified for data output. By convention, particle data are denoted by `p.` followed by a particular variable name such as mass, velocity, stress, etc. Whereas for node based data, the convention is to use the `g.` followed by the variable name, such as mass, stress, velocity, etc. Similarly, cell-centered and face-centered data typically end with the a trailing `CC` or `FC,` respectively. Within the DataArchiver section, variables are specified with the following format:

```
<save label = "p.mass" />
<save label = "g.mass" />
```

To see a list of variables available for saving for a given component, execute the following command from the `StandAlone` directory:

```
inputs/labelNames component
```

where `component` is, e.g., `mpm`, `ice,` etc.

Check-pointing information can be created that provides a mechanism for restarting a simulation at a later point in time. The `<checkpoint>` tag with the `cycle` and `interval` attributes describe how many copies of checkpoint data is stored (cycle) and how often it is generated (interval). You may also use the `walltimeStart` and `walltimeInterval` options for specifying when and how offen a checkpoint will be output based on wall-clock time.

As an example of checkpoint data that has two timesteps worth of che dckpoint data that is created every .01 seconds of simulation time are shown below:

```
<checkpoint cycle = "2" interval = "0.01"/>
```

To restart from a checkpointed archive, simply put "`-restart`" in the sus command-line arguments and specify the .uda directory instead of a ups file (sus reads the copied `input.xml` from the archive). One can optionally specify a certain timestep to restart from with `-t timestep` with multiple checkpoints, but the last checkpointed timestep is the default. When restarting, sus copies all of the appropriate information from the old uda directory to its new uda directory.

Here are some examples:

```
./sus -mpm -restart disks.uda.000 -nocopy
./sus -mpm -restart disks.uda.000 -t 29
```

## 2.7 Simulation Options

There are many options available when running MPM simulations. These are generally specified in the `<MPM>` section of the input file. A list of these options taken from `inputs/UPS_SPEC/mpm_spec.xml` is given in section 9.4.

## 2.8 Geometry objects

Within several of the components, the material is described by a combination of physical parameters and the geometry. Geometry objects use the notion of constructive solid geometry operations to compose the layout of the material from simple shapes such as boxes, spheres, cylinders and cones, as well as operators which include the union, intersections, differences of the simple shapes. In addition to the simple shapes, triangulated surfaces can be used in conjunction with the simple shapes and the operations on these shapes.

Each geometry object has the following properties, label (string name), type (box, cylinder, sphere, etc), resolution (vector quantity), and any unique geometry parameters such as origin, corners, triangulated data file, etc. The operators which include, the union, the difference, and intersection tags contain either lists of additional operators or the primitives pieces.

As an example of a non-trivial geometry object is shown below:

```
<geom_object>
     <intersection>
       <box label = "Domain">
          <min>[0.0,0.0,0.0]</min>
          <max>[0.1,0.1,0.1]</max>
       </box>
       <union>
         <sphere label = "First node">
            <origin>[0.022,0.028,0.1  ]</origin>
            <radius>0.01</radius>
         </sphere>
         <sphere label = "2nd node">
            <origin>[0.030,0.075,0.1  ]</origin>
            <radius>0.01</radius>
         </sphere>
       </union>
     </intersection>
     <res>[2,2,2]</res>
     <velocity>[0.,0.,0.]</velocity>
     <temperature>0 </temperature>
</geom_object>
```

The following geometry objects are given with their required tags:

`box` has the following tags: min and max which are vector quantities specified in the `[a, b, c]` format.

`sphere` has an origin tag specified as a vector and the radius tag specified as a float.

`cone` has a tag for the top and bottom origins (vector) as well as tags for the top and bottom radius (float) to create a right circular cone/frustum.

`cylinder` has a tag for the top and bottom origins (vector) plus a tag for the radius (float).

`smoothcyl` is a geomtry object designed for use with the cpdi algorithm, which uses a body fit particle spatial distribution. This eliminates "stair-stepped" boundaries typical of the standard, grid-based, discretization scheme. Thus *it is important to note that this geometry is only designed to work with* `<interpolator>cpdi</interpolator>`. *Other algorithms may give erroneous answers.*

This geometry has the following tags:

```
<smoothcyl label = "label name">
  <discretization_scheme> string </discretization_scheme>
  <bottom> vector </bottom>
```

```
  <top> vector </top>
  <outer_radius> float </outer_radius>
  <inner_radius> float </inner_radius>
  <num_radial> integer </num_radial>
  <num_axial> integer </num_axial>
  <num_angular> integer </num_angular>
  <arc_start_angle> double (in degrees) </arc_start_angle>
  <arc_angle> double (in degrees) </arc_angle>
</smoothcyl>
```

The complete or partial annulus or cylinder specified by inner (optional, defaulting to zero) and outer radii, cylinder axis bottom and top (all required) and start and final angle (both optional, defaulting to 0 and 360) will be discretized using planes of concentric rings of particles. Particle density is specified by `num_axial` and `num_radial`, the number of particles in the axial and radial dimensions, respectively . Note that these particle density specifications supercede those specified in the `<geom_object> <res>` tag, which is ignored.

The required discretization scheme may be either `pie_slices` or `constant_particle_volumes`. For the `pie_slices` discretization, `num_angular` is required to specify the number of particles between `arc_start` and `arc_angle` . For the `constant_particle_volumes` discretization, the number of particles between `arc_start` and `arc_angle` is determined individually for each ring of particles by attempting to keep particle spacings approximately equal in the radial and angular directions, and thus particle volumes approximately constant.

End caps may be added to the smoothcyl using the optional `<endcap_thickness>` tag, which specifies the axial dimension of cylinders which are appended to each end of the specified `smoothcyl`(the radii are the same as the `smoothcyl` ). Presently, the end cap body fit discretization uses a legacy scheme.

Note: At the time of writing, multiple `smoothcyl` geometries within a `<geom_object>` tag were not discretized using a body fit particle distribution as described here (rather the default discretization scheme is used). This will be fixed eventually, at which point it may be possible to create more general endcaps using unions of `smoothcyl` .

`ellipsoid` has an origin tag specified as a vector. There are two ways to assign axis lengths depending on the orientation of the ellipsoid. If the axes are aligned with the Cartesian grid, they may be specified as floating point values with tagnames: rx, ry, rz. For all other orientation, three vector quantities must be specified in the `[a,b,c]` format. Vector quantity tag names are: v1, v2, v3.These vectors must be orthogonal to within 1e-12 after dot product or the simulation will throw an exception. Note, if both vector quantities and floating point tags are used, the vector quantity inputs will take precedence.

`parallelpiped` requires that four points be specified as illustrated by the ASCII art snippet taken from the source code:

```
//*****************************************
//                                        //
//                                        //
//       *------------------*             //
//      / \                / \            //
//    P3...\..............*   \           //
//      \   \              .    \          //
//      (z)  P2-----------------*          //
//        \  /              .   /          //
//         \/                . /           //
//         P1-------(x)-----P4            //
```

```
//                                          //
//
//  Returns true if the point is inside (or on) the parallelepiped.
```

`tri` is a tag for describing a triangulated surface. The name tag specifies the file name to use for reading in the triangulated surface description and the points file. The triangulated surface (file_name.tri) contains a list of integers describing the connectivity of points specified in file_name.pts. Here is an excerpt from a tri file and a points file:

```
Triangulated file

1 39 41
1 41 38
38 41 42
. . .

Points file

0 0.03863 -0.005
0.35227 0.13023 -0.005
0.00403479 0.0296797 -0.005
. . .
```

The Mach 2 Wedge example in Section 10.5 depicts usage of this option.

The boolean operators on the geometry pieces include `difference, intersection,` and `union.`

The `difference` takes two geometry pieces and subtracts the second geometry piece from the first geometry piece. The `intersection` operator requires at least two geometry pieces in forming an intersection geometry piece. Whereas the `union` operator aggregates a collection of geometry pieces. Multiple operators can be used to form very complex geometry pieces.

An additional input in the `<geom_object>` field is the `<res>` tag. In MPM, this simply refers to how many particles are placed in each cell in each coordinate direction. For multi-material ICE simulations, the `<res>` serves a similar purpose in that one can specify the subgrid resolution of the initial material distribution of mixed cells at the interface of geometry objects.

In addition to the above, it is also possible in MPM simulations to describe geometry by providing a file containing a series of particle locations. These can be in either ASCII or binary format. In addition, it is also possible to provide initial data for certain variables on the particles, including volume, temperature, external force, fiber direction (used in material models with transverse isotropy) and velocity. The following is an example in which external force and fiber direction are specified:

```
<file>
    <name>LVcoarse.pts</name>
    <var>p.externalforce</var>
    <var>p.fiberdir</var>
</file>
```

where the text file LVcoarse.pts looks like:

```
0.0385 0.0335 0.0015 0 0 0 0.248865 -0.0593421 -0.966718
0.0395 0.0335 0.0015 0 0 0 0.254892 -0.0220365 -0.966718
0.0405 0.0335 0.0015 0 0 0 0.267002 0.0197728 -0.963493
0.0415 0.0335 0.0015 0 0 0 0.261177 0.0588869 -0.963493
```

.
.
.

Because these files can be arbitrarily large, an additional preprocessing step must be taken before issuing the `sus` command. `pfs` for "Particle File Splitter" is a utility that splits the data in the `.pts` file into a series of files (`file.pts.0, file.pts.1,` , etc), one for each patch. By doing this, each processor needs only read in the data for the patches that it contains, rather than each processor reading in the entire file, which can be hard on the file system. Note, that this step is required, even if only using a single patch, and must be reissued any time the patch configuration is changed. Usage of this utility, which is compiled into the `StandAlone/tools/pfs` directory, is:

```
pfs input.ups
```

One final option is available for initializing particle positions in MPM simulations, and that is through the use of three dimensional image data, such as might be collected via CT scans or confocal microscopy. The image data are provided as 8-bit raw files, and usage in the input file is given as:

```
<image>
  <name>spheres.raw</name>
  <res>[1600, 1600, 1600]</res>
  <threshold>[1, 25]</threshold>
</image>
<file>
  <name>spheres.pts</name>
  <format>bin</format>
</file>
```

The `<image>` section gives the name of the file, the resolution, in pixels, in the various coordinate directions, and threshold range. Particles will be generated at voxels within the specified range. The `<file>` section is the same as that described above. A different preprocessing utility is provided when using image data (for the same reasons described previously). Usage is as follows:

```
pfs2 -b input.ups
```

The -b indicates that binary `spheres.pts.#` files will be created, which saves considerable disk space when performing large simulations.

## 2.9 Boundary conditions

Boundary conditions are specified within the `<Grid>` but are described separately for clarity. The essential idea is that boundary conditions are specified on the domain of the grid. Values can be assigned either on the entire face, or parts of the face. Combinations of various geometric descriptions are used to aid in the assignment of values over specific regions of the grid. Each of the six faces of the grid is denoted by either the minus or plus side of the domain.

The XML description of a particular boundary condition includes which side of the domain, the material id, what type of boundary condition (Dirichlet or Neumann) and which variable and the value assigned. The following is a an MPM specification of a Dirichlet boundary condition assigned to the velocity component on the x minus face (the entire side) with a vector value of [0.0,0.0,0.0] applied to all of the materials.

```
<Grid>
      <BoundaryConditions>
        <Face side = "x-">
            <BCType id = "all" var = "Dirichlet" label = "Velocity">
                  <value> [0.0,0.0,0.0] </value>
            </BCType>
        </Face>
        <Face side = "x+">
           <BCType id = "all" var = "Dirichlet" label = "Velocity">
                <value> [0.0,0.0,0.0] </value>
           </BCType>
        </Face>
       . . . .
       <BoundaryCondition>
  . . . .
 <Grid>
```

The notation `<Face side = "x-">` indicates that the entire x minus face of the boundary will have the boundary condition applied. The `id = "all"` means that all the materials will have this value. To specify the boundary condition for a particular material, specify an integer number instead of the "all". The `var = "Dirichlet"` is used to specify whether it is a Dirichlet or Neumann or symmetry boundary conditions. Different components may use the `var` to include a variety of different boundary conditions and are explained more fully in the following component sections. The `label = "Velocity"` specifies which variable is being assigned and again is component dependent. The `<value> [0.0,0.0,0.0] </value>` specifies the value.

An example of a more complicated boundary condition demonstrating a hot jet of fluid issued into the domain is described. The jet is described by a circle on one side of the domain with boundary conditions that are different in the circular jet compared to the rest of the side.

```
<Face circle = "y-" origin = "0.0 0.0 0.0" radius = ".5">
       <BCType id = "0"   label = "Pressure" var = "Neumann">
                          <value> 0.0   </value>
       </BCType>
       <BCType id = "0" label = "Velocity" var = "Dirichlet">
                          <value> [0.,1.,0.] </value>
       </BCType>
       <BCType id = "0" label = "Temperature" var = "Dirichlet">
                          <value> 1000.0  </value>
       </BCType>
       <BCType id = "0" label = "Density" var = "Dirichlet">
                          <value> .35379  </value>
       </BCType>
       <BCType id = "0" label = "SpecificVol"  var = "computeFromDensity">
                          <value> 0.0  </value>
       </BCType>
     </Face>
     <Face side = "y-">
       <BCType id = "0"   label = "Pressure"     var = "Neumann">
                          <value> 0.0   </value>
       </BCType>
       <BCType id = "0" label = "Velocity"     var = "Dirichlet">
                          <value> [0.,0.,0.] </value>
```

```
      </BCType>
      <BCType id = "0" label = "Temperature"  var = "Neumann">
                           <value> 0.0  </value>
      </BCType>
      <BCType id = "0" label = "Density"      var = "Neumann">
                           <value> 0.0  </value>
      </BCType>
      <BCType id = "0" label = "SpecificVol"  var = "computeFromDensity">
                           <value> 0.0  </value>
      </BCType>
    </Face>
```

The jet is described by the circle on the y minus face with the origin at 0,0,0 and a radius of .5. For the region outside of the circle, the boundary conditions are different. Each side must have at least the `"side"` specified, but additional circles and rectangles can be specified on a given face.

An example of the `rectangle` is specified as with the lower corner at 0,0.181,0 and upper corner at 0,0.5,0.

```
<Face rectangle = "x-" lower = "0.0 0.181 0.0" upper = "0.0 0.5 0.0">
```

An example of a rectangular annulus "`rectangulus`" is created by specifying an outer and inner `rectangle`. For example an outer rectangle with the lower corner at 0.0,0.5,0.5 and upper corner at 0.0,1.0,1.0, and an inner rectangle with the lower corner at 0.0,0.6,0.6 and upper corner at 0.0,0.9,0.9 can be specified as follows:

```
<Face rectangulus="x-" inner_lower="0.0 0.6 0.6" inner_upper="0.0 0.9 0.9"
                       outer_lower="0.0 0.5 0.5" outer_upper="0.0 1.0 1.0">
```

## 2.10 Grid specification

The `<Grid>` section specifies the domain of the structured grid and includes tags which indicate the lower and upper corners, the number of extra cells which can be used by various components for the application of boundary conditions or interpolation schemes.

The grid is decomposed into a number of patches. For single processor problems, usually one patch is used for the entire domain. For multiple processor simulations, there must be at least one patch per processor. Patches are specified along the x,y,z directions of the grid using the `<patches> [2,5,3] </patches>` which specifies two patches along the x direction, five patches along the y direction and 3 patches along the z direction. The maximum number of processors that `sus` could use is $2 * 5 * 3 = 30$. Attempting to use more processors than patches will cause a run time error during initialization.

Finally, the grid spacing can specified using either a fixed number of cells along each x,y,z direction or by the size of the grid cell in each direction. To specify a fixed number of grid cells, use the `<resolution> [20,20,3] </resolution>` . This specifies 20 grid cells in the x direction, 20 in the y direction and 3 in the z direction. To specify the grid cell size use the `<spacing> [0.5,0.5,0.3] </spacing>` . This specifies the a grid cell size of .5 in the x and y directions and .3 in the z direction. The `<resolution>` and `<spacing>` cannot be specified together. The following two examples would generate identical grids:

```
<Level>
    <Box label="1">
```

```
      <lower>        [0,0,0]          </lower>
      <upper>        [5,5,5]          </upper>
      <extraCells>   [1,1,1]          </extraCells>
      <patches>      [1,1,1]          </patches>
    </Box>
    <spacing>        [0.5,0.5,0.5]    </spacing>
</Level>

<Level>
    <Box label="1">
      <lower>        [0,0,0]          </lower>
      <upper>        [5,5,5]          </upper>
      <resolution>   [10,10,10]       </resolution>
      <extraCells>   [1,1,1]          </extraCells>
      <patches>      [1,1,1]          </patches>
    </Box>
</Level>
```

The above examples indicate that the grid domain has a lower corner at 0,0,0 and an upper corner at 5,5,5 with one extra cell in each direction. The domain is broken down into one patch covering the entire domain with a grid spacing of .5,.5,.5. Along each dimension there are ten cells in the interior of the grid and one layer of "extraCells" outside of the domain. extraCells are the Uintah nomenclature for what are frequently referred to as "ghost-cells".

## 2.11 Schedulers

In Uintah, the task scheduler component is responsible for computing task dependencies, determining the order of task execution and ensuring that the correct inter-process communication is performed. The Uintah task scheduler compiles all of the tasks and variable dependencies into a task-graph. Dependency edges are added between tasks based on the supplied variable dependencies. The computed dependency edges can be either internal or external. Internal dependencies are between patches on the same processor and external dependencies are between patches on different processors. Thus internal dependencies imply a *necessary order* where external dependencies specify *required communication*. The compilation process also combines external dependencies from the same source or to the same destination, thus coalescing messages.

The following is a brief summary of Uintah's available schedulers, followed by more in-depth discussion of each, its usage and input (XML) specification.

- **MPI Scheduler:** Static task ordering and deterministic execution with MPI. One MPI rank per CPU core.
- **Dynamic MPI Scheduler:** Dynamic scheduling with non-deterministic, out-of-order execution of tasks at runtime. *One MPI rank per CPU core.*
- **Unified Scheduler:** An advanced, multi-threaded scheduler that uses a combination of MPI + Pthreads and offers support for GPU tasks (MPI + Pthreads + NVIDIA CUDA). Dynamic scheduling with non-deterministic, out-of-order execution of tasks at runtime. *One MPI rank per multi-core node.* Pthreads are pinned to individual CPU cores where these tasks are executed. Uses a decentralized model wherein all threads can access task queues, processes there own MPI send and recvs, with shared access to the DataWarehouse. (requires **MPI_THREAD_MULTIPLE** support)

A comprehensive example of Scheduler input file options with explanation.

```
<Scheduler type="DynamicMPI">
  <small_messages> true </small_messages>
  <taskReadyQueueAlg> MostMessages </taskReadyQueueAlg>
  <VarTracker>
      <start_time> 0.0 </start_time>
      <end_time> 0.2 </end_time>
      <start_index> [0,0,0] </start_index>
      <end_index> [12,12,12] </end_index>
      <var label="divQ" dw="OldDW"/>
      <locations before_comm="false" before_exec="true" after_exec="true"/>
      <task name="Ray::rayTrace"/>
  </VarTracker>
</Scheduler>
```

- *Scheduler* - specifies the specific Uintah task scheduler to use. Valid options are: `MPI` `DynamicMPI` `Unified`
- *small_messages* - whether or not to turn on MPI message combination, e.g. send small, individual messages (takes more work to organize) or large, combined messages (more communication time). There is no "best" value for it. Sometimes MPI message combination works better, sometimes not.
- *taskReadyQueueAlg* - (only applicable for Dynamic and Unified Schedulers) Priority for sorting of tasks in task queues. Valid options are:
  `PatchOrder` `PatchOrderRandom` `MostMessages` `LeastMessages` `Random` `FCFS` `Stack`.
  Evidence suggests using `MostMessages` algorithm works best in general. This means highest execution priority is given to tasks that will generate *the most outgoing MPI messages*.
- *VarTracker* - This allows the user to track values for variables throughout a simulation or at specific points/ranges in time. The elements below control this.
- *start_time* - Variable tracking start time.
- *end_time* - Variable tracking end time.
- *start_index* - Variable tracking starting cell index.
- *end_index* - Variable tracking starting end index.
- *var* - Specify the name of the variable to track and the DataWarehouse to track from. Valid DataWarehouse options are: `NewDW` `OldDW` `CoarseNewDW` `CoarseOldDW` `ParentOldDW` `ParentNewDW`
- *locations* - The points in the simulation at which variable information will be reported, e.g. before communication and before/after task execution.
- *task* - The specific task to track the specified variable.

Using the `<Scheduler>` section above in a particular input file, the Uintah infrastructure will report something like the following at the beginning of the simulation:

```
Parallel: 1 MPI process (using MPI)
Parallel: MPI Level Required: 0, provided: 0
Date: Thu Jan 8 16:05:22 2015
Machine: albion
SVN: Revision: 52944
SVN: Last Changed Date: 2015-01-08 09:41:48 -0700 (Thu, 08 Jan 2015)
Assertion level: 3
CFLAGS: -fPIC -fopenmp -Wall -g -O0 -fno-inline-functions
Implicit Solver:      CGSolver
Simulation Component: 'RMCRT_Test'
Load Balancer:        SimpleLoadBalancer
Scheduler:            DynamicMPI

------------------------------------------------------------
-- Initializing VarTracker...
-- Running from time 0 to 0.2
-- for indices: [int 0, 0, 0] to [int 12, 12, 12]
-- Printing variable information before task execution.
-- Printing variable information after task execution.
-- Tracking variable 'phi' in DataWarehouse 'OldDW'
-- Tracking variables for specific task: Poisson1::timeAdvance
------------------------------------------------------------
```

Do note that you may use the MPI or Unified scheduler without ANY of the above input file options. By launching `sus` via `mpirun (mpiexec)` will invoke the MPI Scheduler automatically; and by launching `sus` via `mpirun (mpiexec)` with the added "`nthreads <n>`" will invoke the Unified Scheduler automatically. The input file options simply give explicit control over what scheduler is used and also makes the Variable Tracking capabilities available to a Uintah simulation.

### 2.11.1 MPI Scheduler

Uintah's MPI Scheduler, shown in Figure 2.1 is the basic task scheduler to be used in a distributed, cluster setting. The MPI scheduler uses a pure process model, e.g. one MPI process per core. Though distributed, the task-graph on each compute node will be identical and is executed in the same order, that is static task ordering and deterministic execution of ALL tasks. This normally works well for most cases under roughly 100K cores and where results need to be reproduced to machine precision.

It has been shown that there is a substantial increase in MPI communication time at larger numbers of cores due to dependencies between computing tasks distributed to different nodes and Uintah's memory use associated with ghost cells and global meta-data [**?**]. This increase becomes a barrier to scalability beyond $\mathcal{O}(100K)$ cores. Beyond these core counts you will have to employ Uintah's Unified Scheduler, described below in Subsection 2.11.3, which moves to a shared memory model on-node, and drastically reduces the memory footprint seen at high core counts with an MPI-only approach [**?**, **?**].

For most cases, this is the default scheduler, but you may also specify `MPI` in the `<Scheduler>` section of your input file. An example command line to use the MPI Scheduler would look like:

```
mpirun -np <#procs> sus -mpi input.ups
```

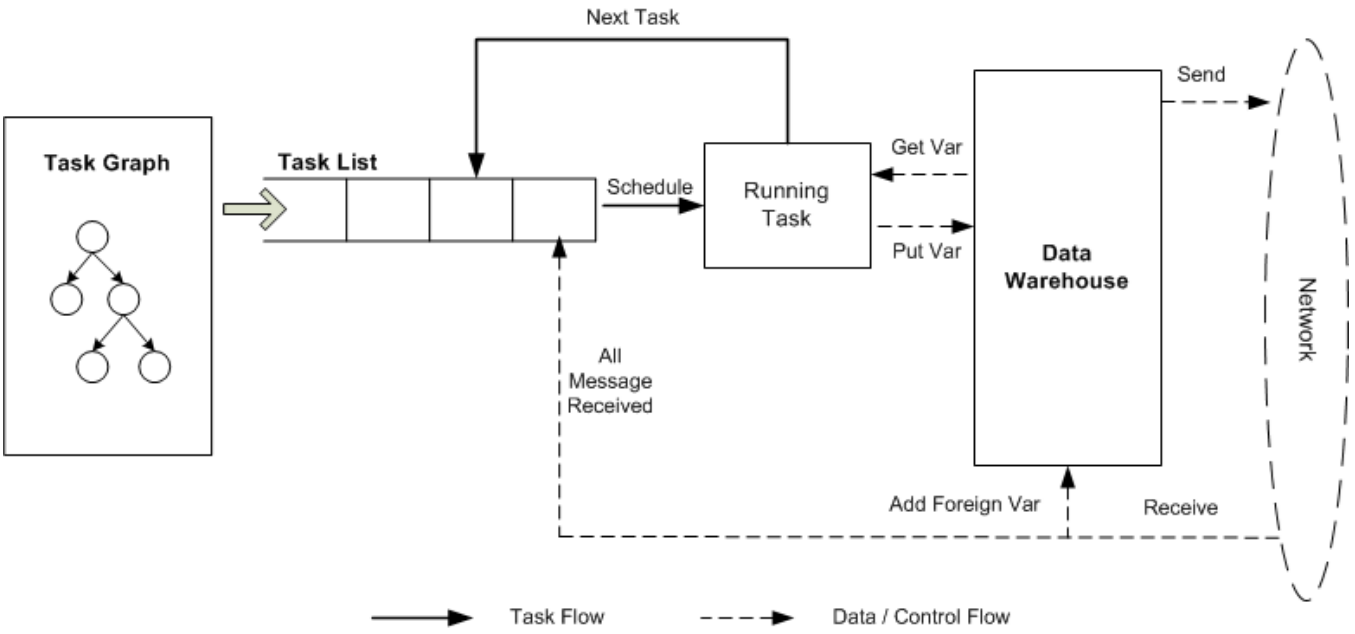



Figure 2.1: MPI Scheduler

### 2.11.2 Dynamic MPI Scheduler

In the MPI scheduler described above in Subsection 2.11.1, tasks are executed in a pre-determined order, which may cause the simulation to sit idle when a single task is waiting for a message. Measurements have shown that this type of delay can be nearly 80 percent of the total MPI wait time in Uintah [**?**]. The Dynamic MPI scheduler, shown in Figure 2.2 adaptively changes the task order during the execution to overlap communication and computation. This scheduler achieves a significant performance benefit in lowering both the MPI wait time and the overall runtime. The dynamic scheduler utilizes two task queues: an internal ready queue and an external ready queue. If a tasks internal dependencies are satisfied, then that task will be put in the internal ready queue where it will wait until all required MPI communication has finished. A counter of outstanding MPI messages is tracked for each task. When this counter reaches zero the requisite, external communication is complete and the task is ready to be executed. At that point it is placed in the external ready queue (ranked based on the task priority algorithm used). To invoke the Dynamic MPI Scheduler, you must specify `DynamicMPI` in the `<Scheduler>` section of your input file. An example command line to use the Dynamic MPI Scheduler would look like:

```
mpirun -np <#procs> sus -mpi input.ups
```

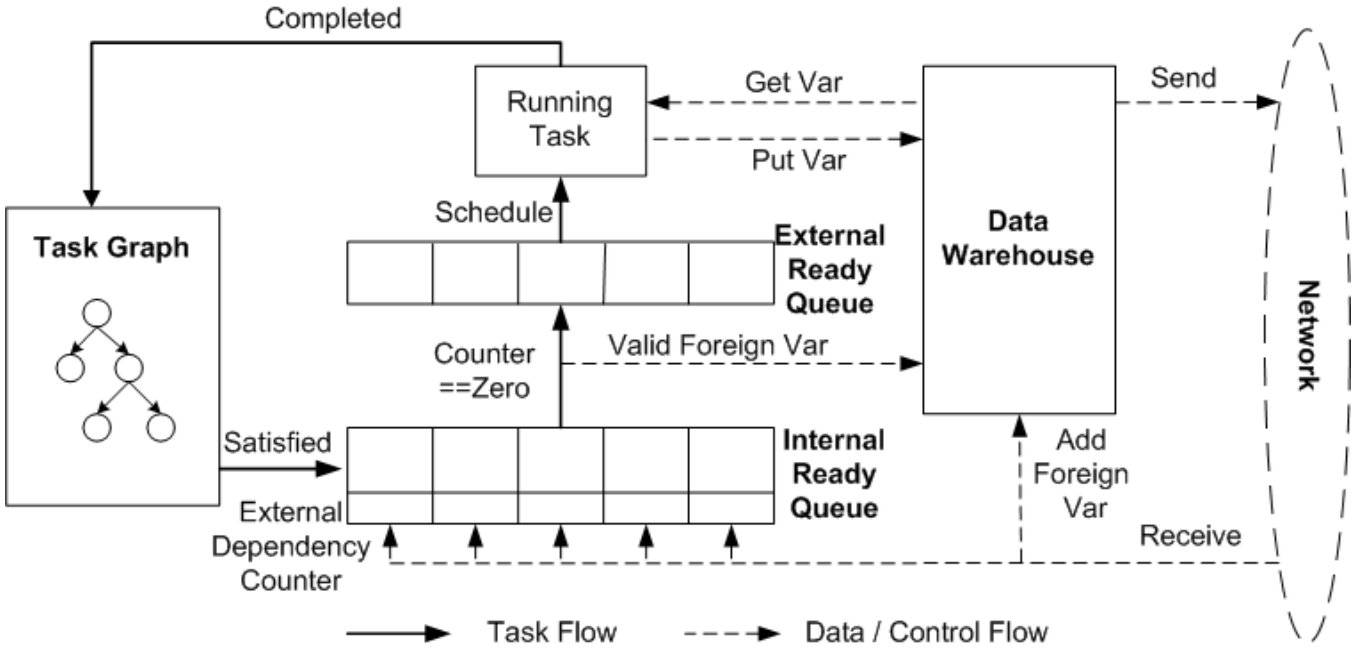



Figure 2.2: Dynamic MPI Scheduler

### 2.11.3 Unified Scheduler

At 100K cores, we have shown that with an MPI-only approach, memory use associated with ghost cells and global meta-data becomes a barrier to scalability beyond $\mathcal{O}(100K)$ cores [**?**]. This limitation has been overcome within Uintah through the use of hybrid parallelism, significantly reducing the memory footprint of the code per core by the development of a multi-threaded task scheduler that takes advantage of current multi-core and emerging many-core architectures. This hybrid memory approach (a combination of MPI + Pthreads (std::thread)) has enabled Uintah to demonstrate scalability up to 256K cores on DOE Titan 768k cores on the DOE Mira system [**?**].

With Uintah's dynamic and static MPI schedulers, based on a pure process model (Subsections 2.11.1 and 2.11.2), data structures are created on each MPI process. Although most Uintah infrastructure components are carefully designed to be stored in a distributed manner, it is necessary for some data to be stored multiple times, e.g. neighboring patch sets, neighboring tasks and ghost variables. A limitation of pure MPI scheduling is that tasks which are created and executed on the same node cannot share data. The multi-threaded Unified scheduler (2.3) extends the Uintah runtime system to support multi-threaded execution, allowing all threads to access task queues (assigning work to themselves) during execution, process their own MPI sends and recvs, and share the same infrastructure components between all threads. The architecture of the runtime system has been extended to support multi-threaded and GPU execution [**?**]. The scalability of UIntah's GPU support has been shown to scale to 16K GPUs on the DOE Titan system [**?**].

Compared to Uintahs dynamic MPI and Threaded MPI scheduler, the Unified scheduler has several independent worker threads per MPI process. These all share infrastructure components such as the regridder, the load balancer, the task graph and the data warehouse and all have read and write access to them (via efficient, lock-free data structures). Again, all threads on a node additionally process their own MPI.

To invoke the Unified scheduler, you may specify `Unified` in the `<Scheduler>` section of your

input file, or simply add the "`nthreads <n>`" (shown below) to your command to launch the `sus` executable.

```
mpirun -np <#procs> sus -mpi -nthreads 16 input.ups
```

Typically the `<n>` will be equal to the number of cores available on a compute node. The above command line will launch 1 MPI process per node and create 15 (`nthreads -1`) additional task execution threads. The main thread of execution will also execute tasks itself in the same way spawned threads do.

**NOTE: This scheduler requires MPI_THREAD_MULTIPLE support.**

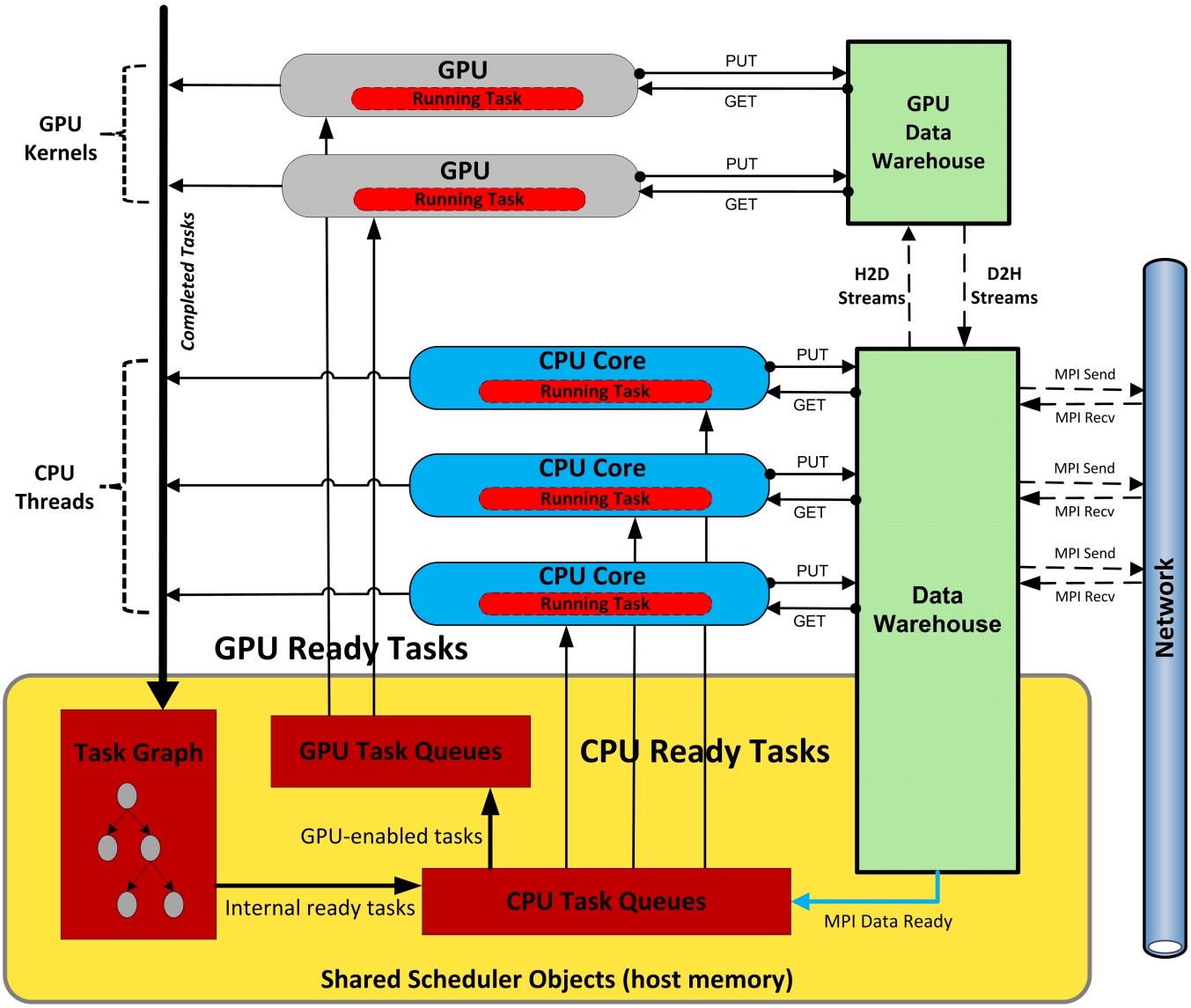



Figure 2.3: Multi-threaded Unified Scheduler with GPU support

## 2.12 GPU Support

Uintah provides support for NVIDIA GPUs through simulation variable preparation and task execution. The high-level goal is to maintain a strong layer of separation between the runtime and the application task developer using the same philosophy that exists for CPU tasks. Uintah's runtime manages copying simulation variables in and out of GPU memory, ghost cell scattering and gathering, task execution, overlapping GPU kernels, and overlapping computation with communication. This GPU engine has been demonstrated at scale using 16K GPUs [**?**] and in a heterogeneous production 1000 MWe coal boiler simulation on Titan using 7.5K GPUs in a mixture of ∼1400 GPU and CPU tasks per MPI rank. GPU support is still under active development, and some features are likely to change, most notably the host-side GPU Data Warehouse, and support for GPUs with Kokkos.

This user guide will describe the features of Uintah's GPU support through a simple Poisson example. To begin, Uintah must be compiled with CUDA support which requires Uintah's configure step list the path to CUDA using `--with-cuda=/path/to/cuda`. Once configured and then compiled, Uintah can verify if a GPU is active by running `sus -gpucheck`.

A task is designated as a GPU enabled task within its declaration, generally alongside when its `computes` and `requires` are also declared. This declaration is done like so:

```
task->usesDevice(true);
```

An example of this is found in Uintah's UnifiedSchedulerTest.cc in the `src/CCA/Components/Examples directory` in the `scheduleTimeAdvance` method. The task's callback function remains the same, a call to a CPU task. Even though it is a GPU enabled task, a CPU task function must manually invoke a GPU kernel containing CUDA code, as seen in that UnifiedSchedulerTest.cc's `timeAdvanceUnified` method. All GPU task code should still follow normal strategies for CUDA programming.

Uintah's runtime maintains a Task Data Warehouse for GPU tasks to enable a task developer to access simulation variables within CUDA code. A task's Task Data Warehouses contain only entries for all the simulation variables defined in that task's declaration. It is strongly recommended (though not required) that these data warehouses are passed into the kernel. An example of this is seen in `timeAdvanceUnified`'s method. A task developer can retrieve a simulation variable through a simple `get()` or `getModifiable()`API call, as seen in UnifiedSchedulerTest.cu's `unifiedSchedulerTestKernel()`. An example is given below:

```
const GPUGridVariable<double> phi;
GPUGridVariable<double> newphi;
old_gpudw->get(phi, "phi", patchID, 0, 0);
new_gpudw->getModifiable(newphi, "phi", patchID, 0);
```

From here individual cells in these grid variables variables can be accessed using an (i,j,k) syntax.

```
newphi(i,j,k) = (1. / 6)
* (phi(i-1, j, k) + phi(i+1, j, k)
+  phi(i, j-1, k) + phi(i, j+1, k)
+  phi(i, j, k-1) + phi(i, j, k+1));
```

The underlying runtime logic ensures that if two different tasks share the same simulation variable on a patch, only one instance of that simulation variable will be created. This sharing reduces total GPU memory usage. A consequence of this logic that the GPU Data Warehouse

has no allocateAndPut() API call. Instead, the task scheduler allocates and frees all simulation variables before and after task execution.

This paragraph describes limitations of Uintah's GPU support. Only the Unified Scheduler can be used, the other schedulers do not have any GPU runtime code. Uintah's CPU tasks and associated data warehouses support five types of simulation variables, namely 1) grid, 2) per patch variables, 3) reduction, 4) sole, and 5) particle. GPU tasks and associated data warehouses do not support sole and particle variables at this time. Uintah supports one GPU per MPI rank. There are no future plans to support multiple GPUs per MPI rank due to a combination of three reasons: 1) the lack of any target problem requiring this need, 2) the challenges and scope issues of multiple GPUs in an MPI rank require significant development time, and 3) Kokkos has no plans to support multiple GPUs per MPI rank. If a target machine has multiple GPUs, consider using the environment variable `CUDA_VISIBLE_DEVICES` to enable one GPU. Support for NVLink has not been enabled. Any simulation declared as `modifies` is not currently supported, but will likely be in the near future.

## 2.13 AMR

In general, the AMR input looks like:

```
<AMR>
    <ICE>
      <do_Refluxing>        false    </do_Refluxing>
      <orderOfInterpolation>1         </orderOfInterpolation>
      <Refinement_Criteria_Thresholds>
        <Variable name = "press_CC" value = "1e6" matl = "0" />
      </Refinement_Criteria_Thresholds>
    </ICE>
    <MPM>
      <min_grid_level>-1</min_grid_level>
      <max_grid_level>-1</max_grid_level>
    </MPM>
    <useLockStep>true</useLockStep>
    <Regridder type="Tiled">

      <!--General Regridder Settings-->
      <max_levels>2</max_levels>
      <cell_refinement_ratio>    [[2,2,1]]</cell_refinement_ratio>
      <cell_stability_dilation>   [2,2,1]   </cell_stability_dilation>
      <cell_regrid_dilation>   [1,1,0]   </cell_regrid_dilation>
      <min_boundary_cells>       [1,1,0]   </min_boundary_cells>

      <!--Tiled Specific Settings-->
      <min_patch_size>  [[8,8,1]] </min_patch_size>
      <patches_per_level_per_proc>8</patches_per_level_per_proc>
    </Regridder>
  </AMR>
```

When running an ICE simulation, you must specify the following tags in the ICE section of your input deck.

- do_refluxing - specifies whether or not to perform refluxing (true or false), which equalizes the face values of coarse/fine boundaries between levels.

- orderOfInterpolation - specifies how many coarse cells to use when refining the coarse-fine interface (see below).
- Refinement_Criteria_Thresholds section specifies the variables whose value will determine where to mark refinement flags, see below. Variables need only be specified on adaptive problems.
- min_grid_level (optional) - coarsest level to run ICE on (default = 0).
- max_grid_level (optional) - finest level to run ICE on (default = max-level -1).

If you run an MPM simulation, you must specify the MPM section, and set min_grid_level and max_grid_level to the finest level of the simulation, 0-based (i.e., if there are 2 levels, the level needs to be set to 1). A shortcut to this is to set min- and max_grid_level to -1.

- useLockStep - Some simulations require a lock step cycle (mpmice and implicit ice), as there has to be inter-level communication in the middle of a timestep. See "W-cycle" diagram below. Otherwise the time refinement ratio will be computed from the cell refinement ratio.

The presence of the Regridder section specifies you want to run an adaptive problem.

- type (optional) - sets the Regridder type. The options are "Tiled" (default), "SingleLevel".
- max_levels - maximum number of levels to create in the grid.
- cell_refinement_ratio - How much to refine a cell in each dimension. This can be specified in a comma-separated list, with the dimensions in the default order [[x,y,z]].
- cell_stability_dilation - How much to pad the refinement flags in each dimension for stability reasons. Reset on every timestep.
- cell_regrid_dilation - How much to pad the refinement flags in each dimension in order to reduce regridding frequency. If the refinement flags are still in the finest level due to large enough padding then regridding will not occur. Reset only when regridding occurs.
- min_boundary_cells - The minimum number of cells that needs to exist between one level's coarser level and its finer level (i.e., between level 0 and 2).

When running a non adaptive problem. Adaptive regridding can be turned off by commenting out the regridding section.

- min_timestep_interval - The minimum number of timesteps between each regrid. This will not force a regrid but after the set number of timesteps a regrid will be considered. Only used when adaptive regridding is turned off. min_timestep_interval $\leq$ cell_stability_dilation + 1
- max_timestep_interval - The maximum number of timesteps between each regrid. This will not force a regrid. It tells the code it has been a set number of timesteps since the last regrid and it should look at if a regrid is needed. Used primarily when adaptive regridding is turned off.

Tiled Specific Settings

- min_patch_size - sets the minimum patch size created by the regridder per level. This size must divide evenly into the resolution and must be divisible by the cell refinement ratio.
- patches_per_level_per_proc - sets the number of patches per level per processor that the load balancer attempts to achieve. If the number of patches is significantly more than the number specified the tiled regridder will increase the tile size by a factor of two in order to reduce the number of patches.

An example of a simple, 2-dimensional, tiled AMR problem can be found at `StandAlone/inputs/MPMICE/advect_2L_MI.ups`. When the AMR input for this simulation matches the input at the beginning of this section (2.13), we see the following:

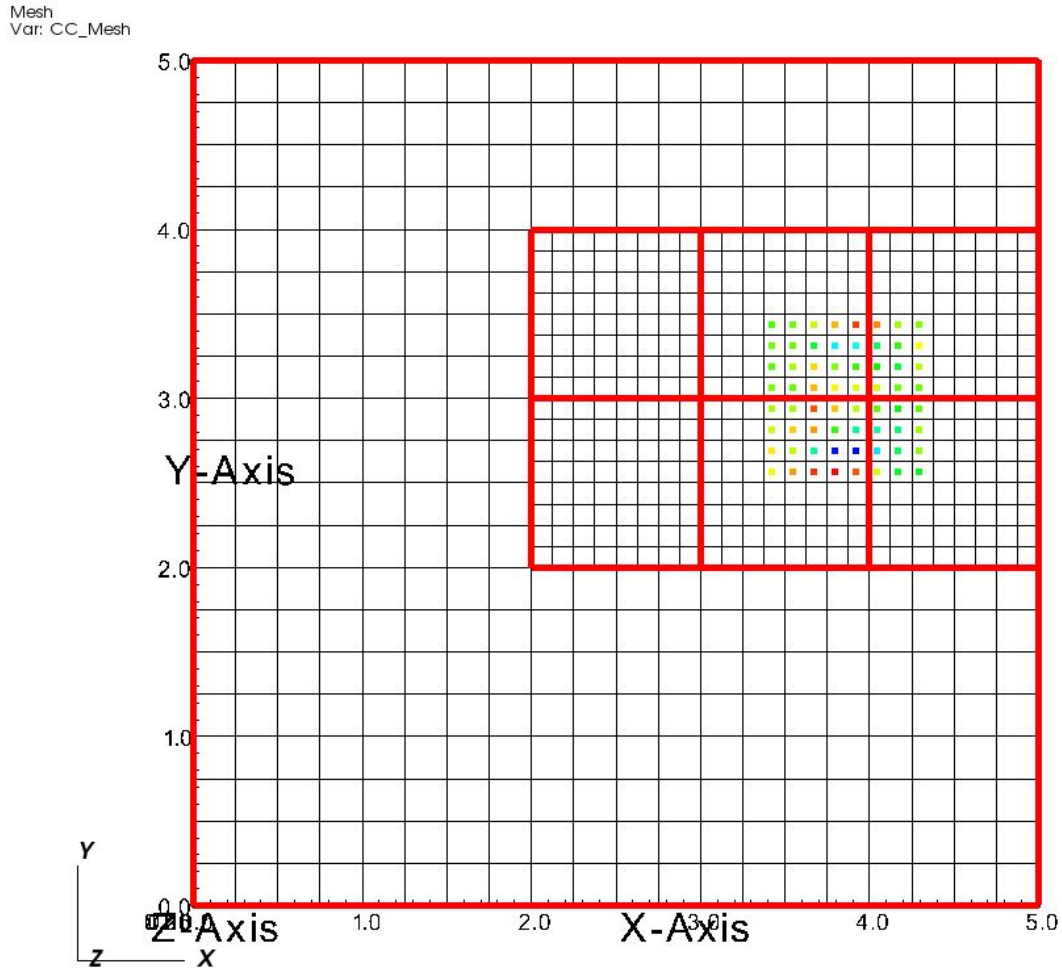



Figure 2.4: Advect_2L_MI with a minimum patch size of [8,8,1]

Here the black lines represent cells and the red lines represent patches with the box of multi-colored points being particles. As can be seen, there are 8 cells per patch in the x and y-direction and only one cell per patch in the z-direction (2D simulation). When we increase the minimum patch size to [20,20,1] we see:

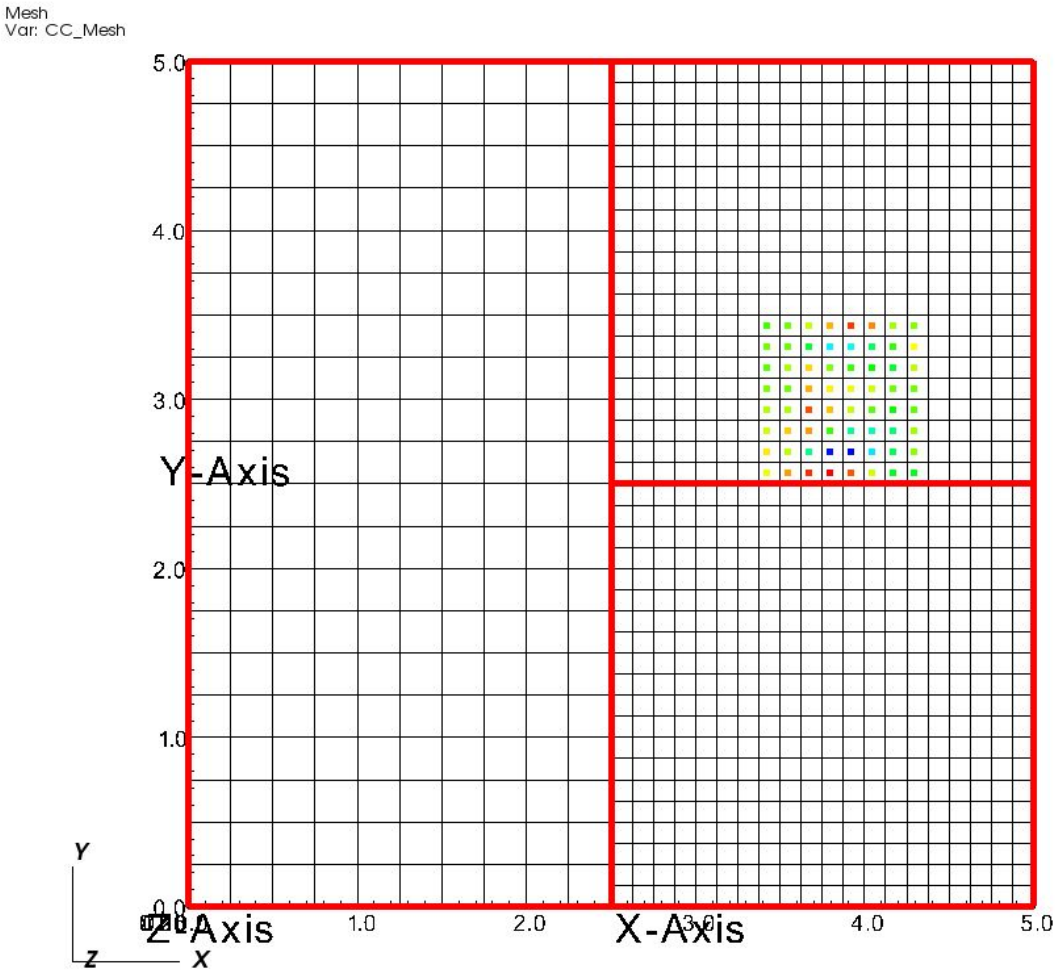



Figure 2.5: Advect_2L_MI with a minimum patch size of [20,20,1]

Note that now there are only 2 patches covering our finest level compared to the 6 patches covering our finest levels in figure 2.4. This is because there is enough "padding" in between our refinement flags and differing levels to avoid setting up whole new patches at the most refined level. This option can be adjusted to increase or decrease the overall amount of regridding necessary with the cell_regrid_dilation flag.

When we decrease our minimum patch size to [4,4,1] we see:

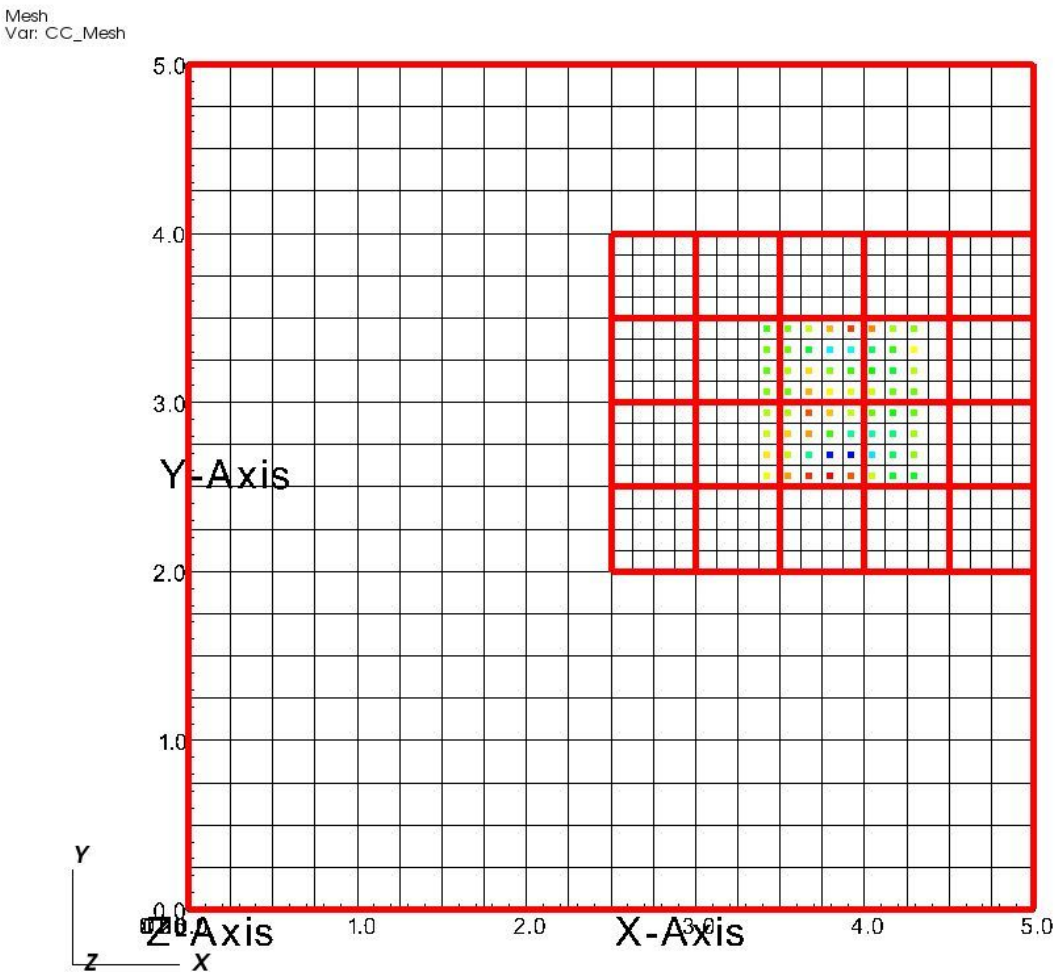



Figure 2.6: Advect_2L_MI with a minimum patch size of [4,4,1]

The best way to increase the "padding" of your simulation and therefore decrease the number of regrids required is to increase the cell_stability_dilation. Below we see the 2 dimensional example with a minimum patch size of [8,8,1], like figure 2.4, but with a cell stability dilation of [10,10,1] rather than [2,2,1].

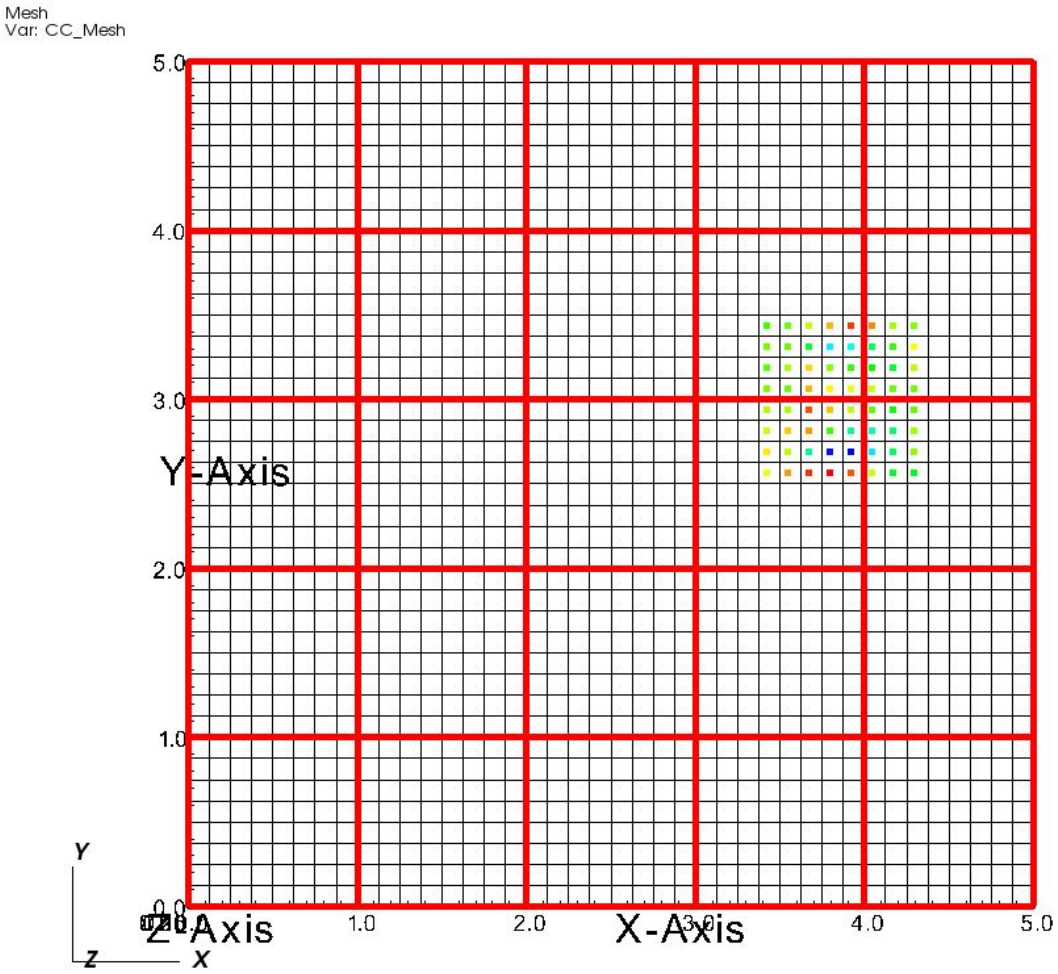



Figure 2.7: Advect_2L_MI with a minimum patch size of [8,8,1] and a cell stability dilation of [10,10,1]

There is a balance between cell stability dilation and computational time. If your domain is set up like figure 2.7, less computational time will be wasted regridding. However, it could be more economical to utilize some coarser levels in the areas that see little or no particles.

### 2.13.1 AMR Grids

There are two ways to run with mesh-refinement, either adaptive or non-adaptive (static). Adaptive grids are created based on the existence of refinement flags that are created during the simulation. A regridder will analyze the whole domain, and, wherever there are refinement flags, construct patches around them on a finer level. See more on Regridding below.

#### Regridding

For an adaptive problem, specify the Regridder section in the input file. The Tiled regridder works as follows:

Tiles sized according to the minimum patch size are laid across domain. Refinement flags are then used to determine which of those tiles are in the patch set. If the number of tiles is more than

twice the target number of patches then the tile size is doubled in the shortest dimension. If the number of tiles is less than the target number of patches then the tile size is halved in the longest dimension. The tile size will never get smaller than the minimum specified tile size. This regridder produces regular patch sets that are easy to load balance. After patches are added, data is stored on them. Then data will be initialized for those new patches, and in the next timestep, those patches will be included in the regridding process.

A constraint of the Regridder is that any two patches that share a boundary must be within one level of each other. See (E) and (F) below.

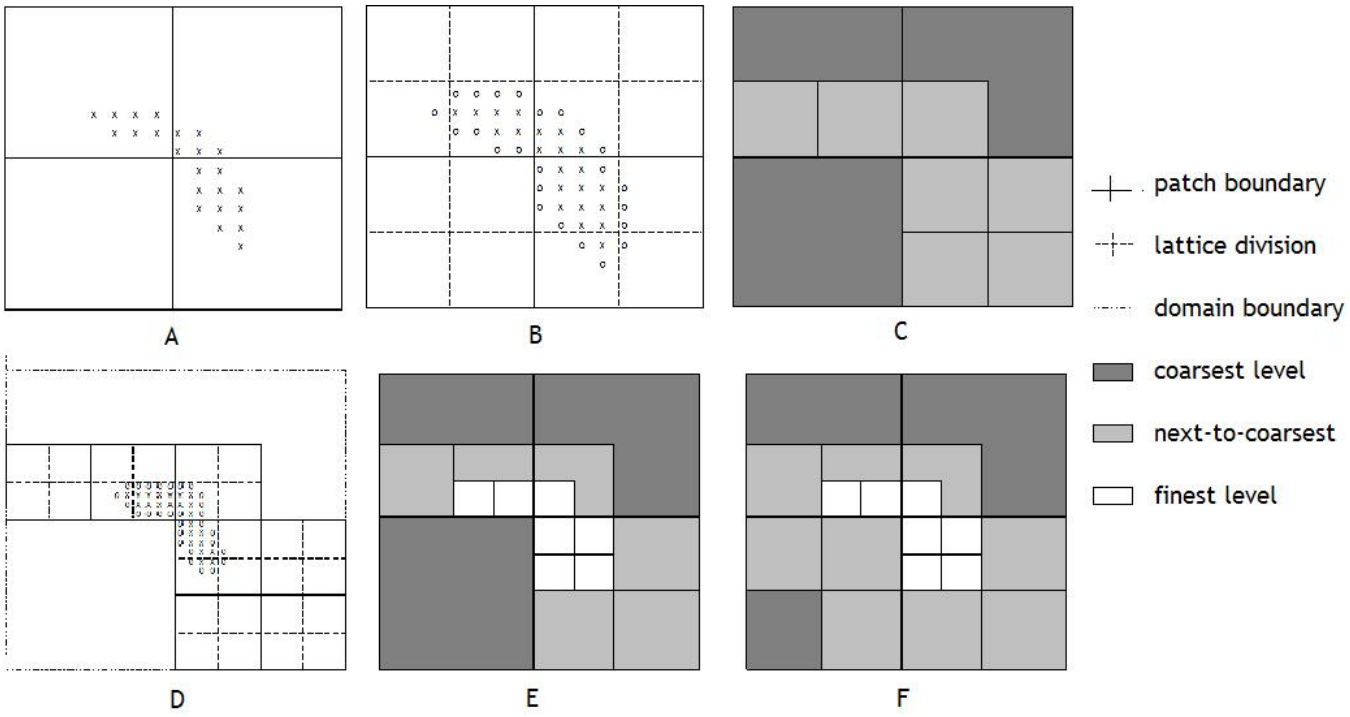



Figure 2.8:

In the diagram above, image (A) show 4 coarse patches with some marked error flags. (B) Shows the subpatches for the next level and has the error flags dilated. (C) Shows the coarse level together with the fine level you end up with.

(D) During the next regrid, the next level can create error flags as well. These are some example error flags that are dilated, with the subpatches for the next level. (E) shows the resulting level with the other levels. However there are some patch boundaries that span more than one level. So in (F) we must expand out the middle level to compensate.

Note that if you define multiple levels in the input file, all but the coarsest level will be recycled, and levels will be added where the Regridder wants to put them.

### Static Grids

Static grids can be defined by simply not including a Regridder section in the input file. See the multiple level example in Grid 2.10.

**Ideal Patch Configuration for Good Scaling**

The patch configuration on multiple levels has been shown to greatly effect the scalability and mean time per time step. Here are some guidelines to follow when trying to get good scaling with multilevel AMR. This example is for a 3 Level simulation.

- On Level 0 and 1 $\frac{\#cells}{patch} = 8$
  i.e. min_patch_size [8,8,8]
- On finer levels (level 2)
  - min_patch_size MUST be greater than or equal to 8,8,8!! This is very important!
  - $\frac{\#cores}{2} \leq$ # patches on the finest level $\leq$ # cores
  - The min_patch_size on L-2 must be divisible by the min_patch_size on L-1

### 2.13.2 AMR Cycle

Whether working with an adaptive or a static grid, AMR problems follow the same cycle.

In short, there are 3 main AMR operations

- Coarsen - This occurs after each execution of a finer level, if the time of the finer level lines up with the time of the coarser level (see the "W-cycle" diagram). Its data are coarsened to the coarser level so that the coarse level has a representation of the data at the finest resolution. Also as part of this operation is the "reflux" operations, which to makes the fluxes across the face of the coarse-fine boundary consistent across levels.
- Refine the coarse-fine interface - This occurs after the execution of each level and after an associated coarsen (if applicable). The cells of the boundary of the finer level are interpolated with the nearest cells on the coarser level (so the finer level stays in sync with the coarser levels).
- Refine - This occurs for new patches created by the regrid operation. Variables that are necessary will be created on those patches by interpolation from the coarser level.

After an entire cycle, then we check to see if we need to regrid. If the flags haven't changed such that patches would form, the grid will remain the same.

In short, these diagrams may be useful:
"W-cycle" (time refinement ratio of 2)
"Lockstep cycle"

## 2.14 Regridder

The regridder creates a multilevel grid from the refinement flags. Each level will completely cover the refinement flags from the coarser level. The primary regridder used in Uintah is the `Tiled` regridder. The tiled regridder creates a set of evenly sized tiles across the domain that will become patches if refinement is required in the tiles region.

The following is an example of this regridder.

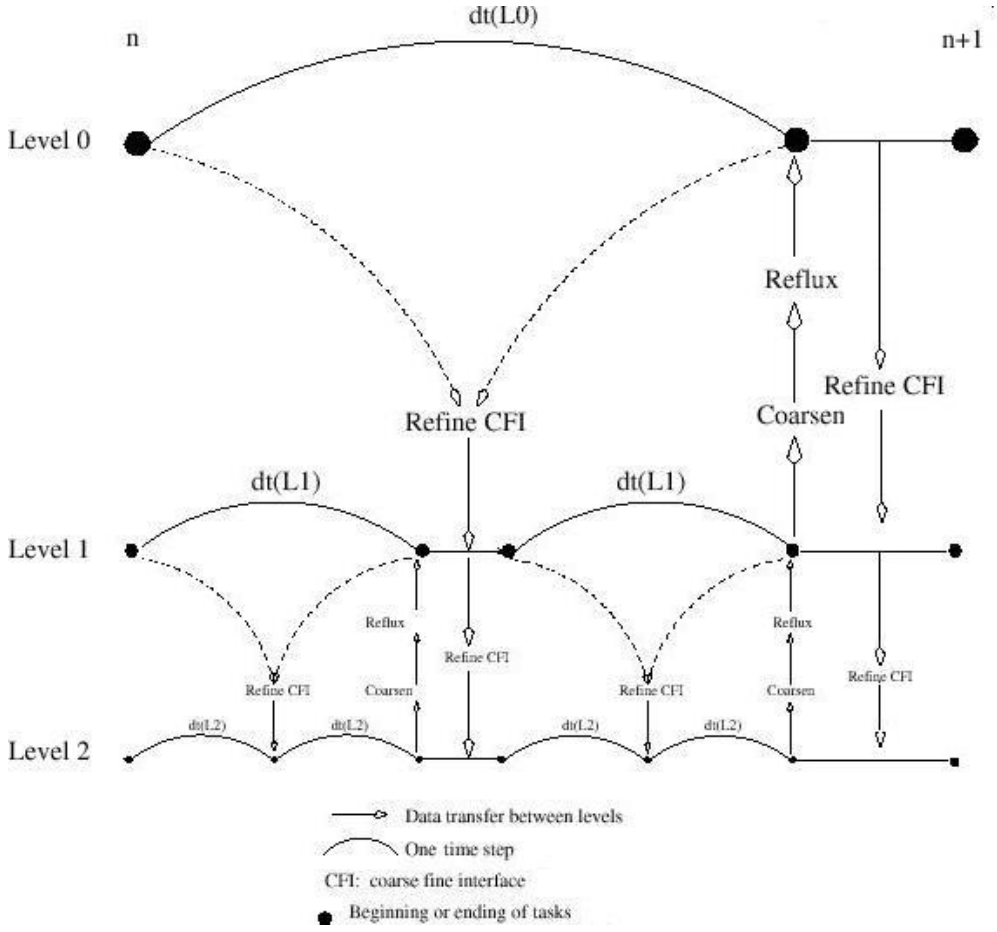



Figure 2.9:

```
<Regridder type="Tiled">
    <max_levels>2</max_levels>
    <cell_refinement_ratio> [[2,2,1]] </cell_refinement_ratio>
    <cell_stability_dilation> [2,2,0] </cell_stability_dilation>
    <min_boundary_cells> [1,1,0] </min_boundary_cells>
    <min_patch_size>  [[8,8,1]] </min_patch_size>
</Regridder>
```

The `max_levels` tag specifies the maximum number of levels to be created. The `cell_refinement_ratio` tag specifies the refinement ratio between the levels. This can be specified on a per level basis as follows:

```
    <cell_refinement_ratio> [[2,2,1],[4,4,1]] </cell_refinement_ratio>
```

The `cell_stability_dilation` tag specifies how many cells around the refinement flags are also guaranteed to be refined. The `min_boundary_cells` tag specifies the size of the boundary layers. The size of the tiles is specified using the `min_patch_size` tag and can also be specified on a per level basis.

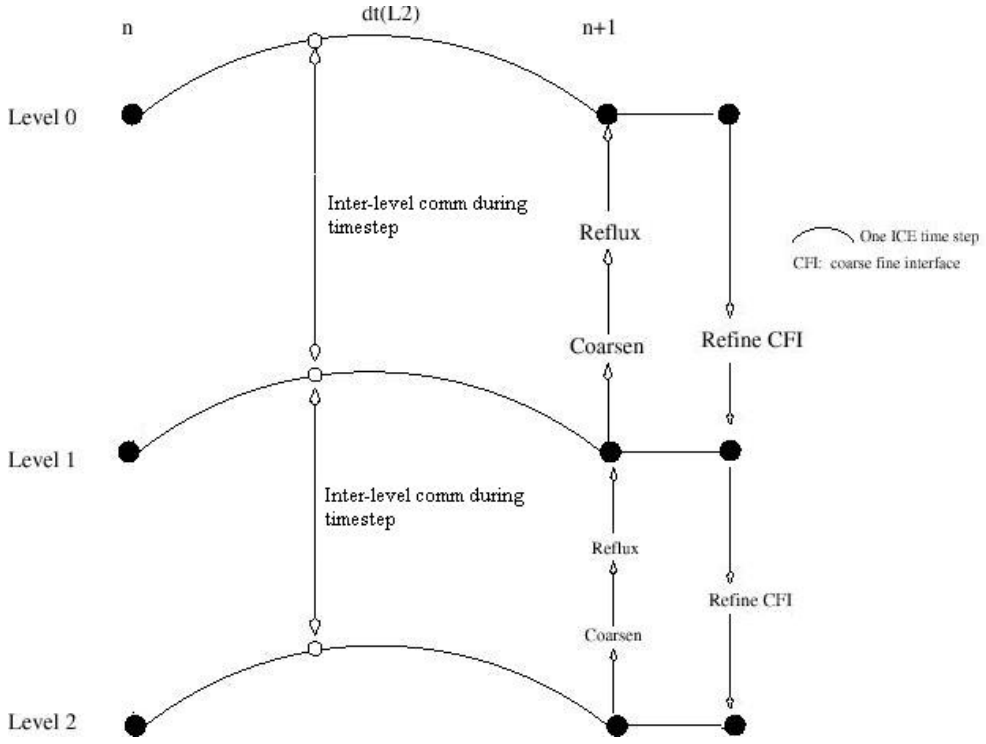



Figure 2.10:

## 2.15 Dynamic Load Balancing

Uintah has a couple of load balancing options which may be useful for increasing performance by decreasing the load imbalance. The following describes the loadbalancer section of an input file and what effects it has on the load balancer.

If no load balancer is specified then a simple load balancing method which assigns an equal number of patches to processors. This is not ideal in most cases and should be avoided.

### 2.15.1 Input File Specs

```
<LoadBalancer type="DLB">
     <!-- DLB specific flags -->
     <costAlgorithm>ModelLS</costAlgorithm>
     <hasParticles>true</hasParticles>

     <!-- DLB/PLB flags -->
     <timestepInterval>25</timestepInterval>
     <gainThreshold>0.15</gainThreshold>
     <outputNthProc>1</outputNthProc>
     <doSpaceCurve>true</doSpaceCurve>

</LoadBalancer>
```

There are two main load balancers used in Uintah. The first is the DLB load balancer. This is a robust load balancer that is good for many problems. In addition, this load balancer can utilize profiling in order to tune itself during the runtime in order to achieve better results.

To use this load balancer the user must specify the type as "DLB". It is also suggested that the user specify a costAlgorithm which can be "Model", "ModelLS", "Memory", or "Kalman" with the

default being "ModelLS". If "hasParticles" is set to true then these cost algorithms will take the number of particles into account when determining the cost.

This algorithm first orders the patches linearly. If doSpaceCurve is set to true then this ordering is done according to a Hilbert Space-Filling curve, which will likely provide better clusterings. Once the patches are ordered linearly, costs are assigned to each patch and the patches are distributed onto processors so that the costs on each processor are even.

The PLB load balancer is an alterantive to the DLB load balancer which is likely more efficent for particle based calculations. This load balancer divides the patches into two sets (cell dominate and particle domintate), which is determined using the particleCost and cellCost parameters. The particle dominate patches are then assigned to processors while trying to equalize the number of particles on each processor. Finally the cell dominate patches are assigned to patches in order to equalize the number of cells while accounting for the number of cells already assigned during the particle assignment phase. This method can also utilize a space-filling curve.

The following list describes other flags utilized by these load balancers:

- timestepInterval - how many timesteps must pass before reevaluating the load balance.
- gainThreshold - the predicted percent improvement that is required to reload balance.
- outputNthProc - output data on only every Nth processor (experimental).

## 2.16 UDA

The UDA is a file/directory structure used to save Uintah simulation data. For the most part, the user need not concern himself with the UDA layout, but it is a good idea to have a general feeling for how the data is stored on disk.

Every time a simulation (sus) is run, a new UDA is created. Sus uses the <filebase> tag in the simulation input file to name the UDA directory (appending a version number). If an UDA of that name already exists, the next version number is used. Additionally, a symbolic link named "disks.uda" (is updated to and) will point to the newest version of this simulations UDA. Eg:

```
disks.uda.000
disks.uda.001
disks.uda.001 <- disks.uda
```

Each UDA consists of a number of top level files, a checkpoints subdirectory, and subdirectories for each saved timestep. These files include:

- `.dat` files contain global information about the simulation (each line in the .dat files contains: simulation_time value).
- `checkpoints` directory contains a limited number of time step data subdirectories that contain a complete snapshot of the simulation (allowing for the simulation to be restarted from that time).
- `input.xml` contains the original problem specification (the .ups file).
- `index.xml` contains information on the actual simulation run.
- `t0000#` contains data saved for that specific time step. The data saved is specified in .ups file and may be a very limited subset of the full simulation data.

The 'validateUda' script (src/Packages/Uintah/scripts/) can be used to test the integrity of a UDA directory. It does not interogate the data for 'correctness', but it performs 5 basic tests on each uda:

```
Usage validateUda  <udas>
Test 0:  Does index.xml exist?                                        true or false
Test 1:  Does each timestep in index.xml exist?                       true or false
Test 2:  Do all timesteps.xml files exist?                            true or false
Test 3:  Do all the level directories exist:                          true or false
Test 4:  Do all of the pxxxx.xml files exist and have size >0:        true or false
Test 5:  Do all of the pxxxx.data files exist and have size > 0:      true or false
```

If any of the tests fail then the corrupt output timestep should be removed from the index.xml file.

See Section 2.6 for a description of how to specify what data are saved and how frequently.

# Chapter 3

# Visualization tools – VisIt

Visualization of Uintah data is currently possible using any of two software packages. These are SCIRun and VisIt. Of these, SCIRun is no longer supported, although legacy versions will continue to work. The VisIt package from LLNL is general purpose visualization software that offers all of the usual capabilities for rendering scientific data. It is still developed and maintained by LLNL staff, and its interface to Uintah data is supported by the Uintah team.

## 3.1 Reading Uintah Data Archives

Once you have installed VisIt and the UDA reader plugin, you can launch VisIt and start visualizing UDA's. To open a UDA, select `Open File` from the `File` menu. Browse into the UDA you want to load and select the `index.xml` file. Then hit on `OK` and a list of timesteps should now appear on the gui. Figure 3.1 illustrates this process.

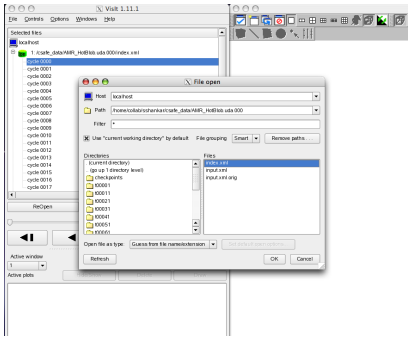


Figure 3.1: Opening an UDA with VisIt

## 3.2 Plots

VisIt displays data as plots. A plot might render a specific variable or it might render the structure of the mesh. Figure 3.2 illustrates this.

Note that VisIt attempts to analyze the variables and associate them with the appropriate plots. As shown in Figure 3.2, only vector variables are available for the vector plot. The most commonly used plots for visualizing UDA's are Pseudocolor, Volume and the Vector plot. The Subset plot can be used to visualize the structure of patches in an AMR dataset.

Once you have a plot, you change plot attributes by clicking on the PlotAtts menu and selecting the plot of you choice. Alternatively,

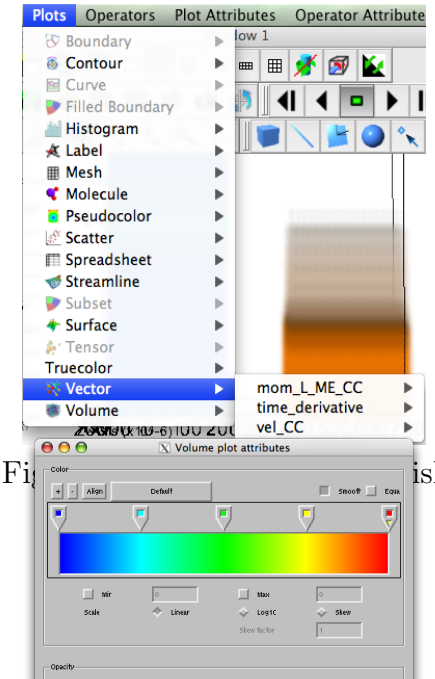



Figure 3.2: Plots in VisIt

you may double click on the plot itself in Active plots window. For example, if you have a Volume plot and you want to change its attributes, the window shown in Figure 3.3 pops up.

As seen in Figure 3.3, you can change the color map, opacity curve, rendering method, no. of samples, lighting options, etc. in this window.

## 3.3 Operators

A wide variety of operators can be applied to the plots, as mentioned earlier. These modify the incoming datasets in some way (eg., a slice formats a 3D dataset into a 2D slice), which can then be plotted. However, you will first need to select a plot and then only you can apply an operator to it (though the order of operation is opposite). An important thing to keep in mind is that when you select an operator, by default it gets applied to all the plots in the Active plots window. You will need to uncheck the Apply operators checkbox, in case you just want to apply the operator to a single plot as shown in Figure 3.4.


Figure 3.4: Unchecking "selection to all plots"

The entire list of operators that VisIt supports can be seen by clicking on the Operators menu. Also, once you have applied an operator, you can change its attributes by clicking on the OpAtts menu and then clicking on the desired operator. Figures 3.5a and 3.5b illustrate how you can apply a Slice operator to a Pseudocolor plot and then change the operator attributes. First, apply the Pseudocolor plot to a desired variable, and then select the Slice operator from the Operators menu.

At this point in time, you should have an ordering similar to that in Figure 3.6a. Once you have this order, select Slice from the OpAtts menu. This will pop up the Slice operator attributes window, as shown in Figure 3.6b.

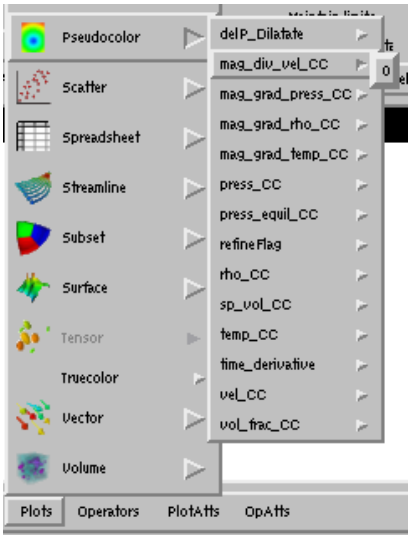
You can now play up with the various attributes (eg., selecting normal plane) to obtain the desired visualization. The checkbox "Project to 2D" should be unchecked is you want to have the slice in 3D space.

## 3.4 Vectors

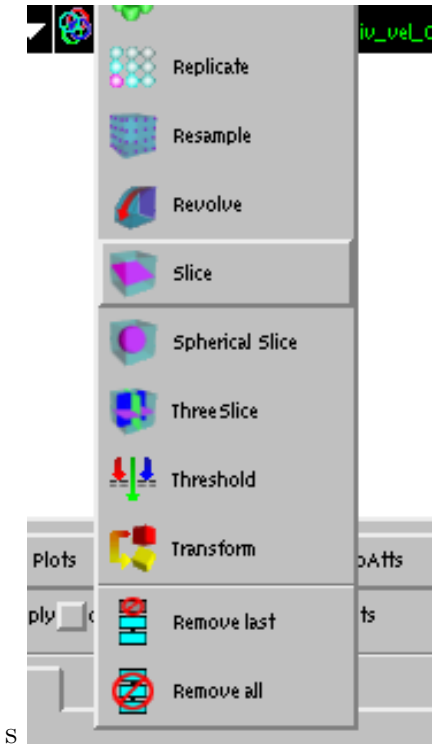
S


By default, VisIt reduces the number of vectors plotted (to 400) and this needs to be manually changed to the original number or something greater, only if required. This can be accomplished by changing the attributes of the Vector plot. In Figure 3.7, the number of vectors has been increased to 2000.

Also if you would like all the vectors to

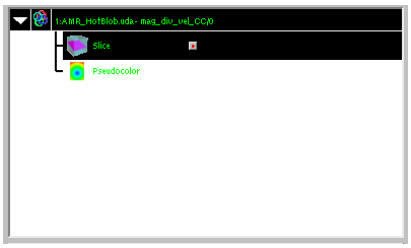



(a) Applying the Pseudocolor plot to a variable

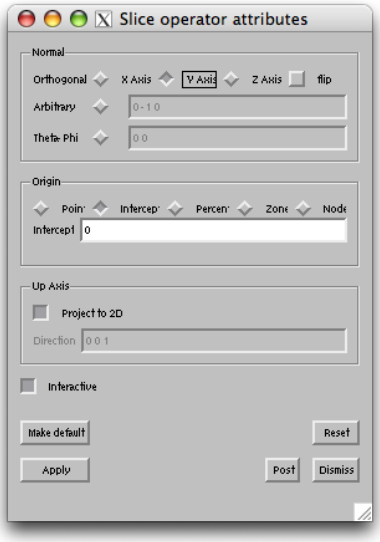



(b) Applying an operator to a plot

Figure 3.5:

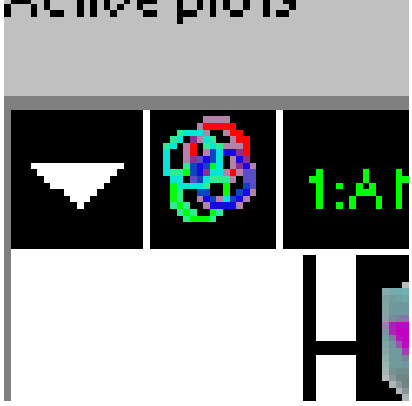



(a) Ordering of an operator and a plot

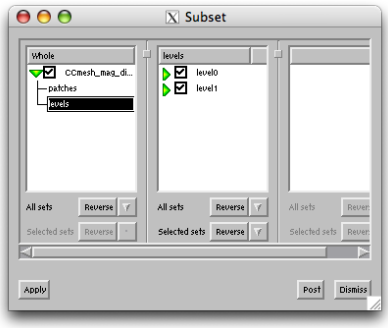



(b) Slice plot attributes in VisIt

Figure 3.6:



(a) Clicking on this icon pops up the Subset window




(b) The Subset window in VisIt

Figure 3.9:

be visible, you would need to switch off both the options, `Scale by magnitude` and `Auto scale` under the Scale tab in the same window as shown in figure 3.8 describes this.

## 3.5 AMR datasets

AMR datasets are read the same way as single level datasets. Once you have it read, you can apply an plot/ operator on it. Since the dataset is organized as levels and patches, you now have the flexibility of visualizing each of them independently or as in a group. To achieve this (assuming that you have already selected a plot), click on the Subset button either on the Active Plots window in the gui or on the same option in the Controls menu. This is illustrated in Figures 3.9a and 3.9b.

## 3.6 Examples

### 3.6.1 Volume visualization

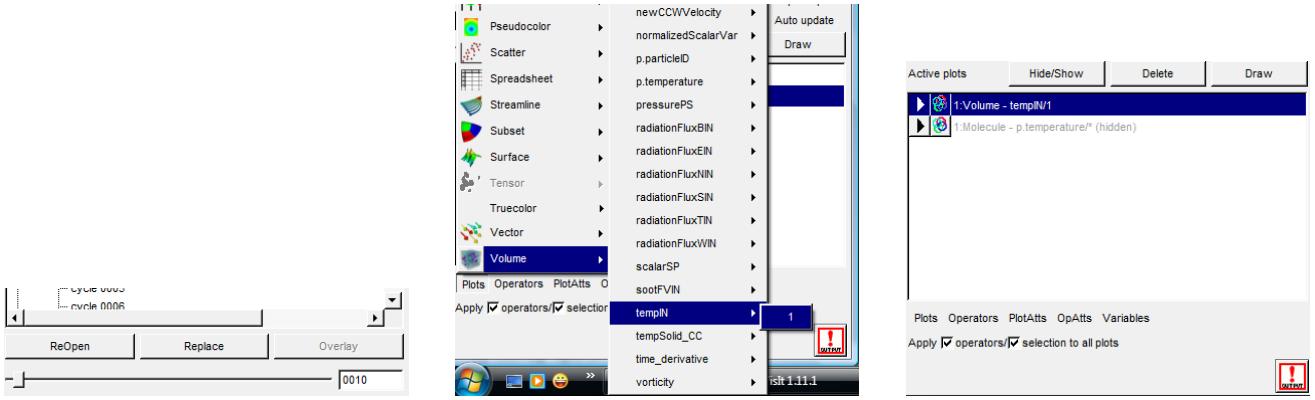
1. Read in the uda by selecting the index.xml file. A list of timesteps should now appear on the gui.

2. The first timestep (cycle 0000) should be preselected. In case you are interested in plotting a different timestep, just double click on it. Alternatively you can type it in the small rectangular box (Figure 3.10a), just below the list of timesteps. This can also be done at a later period in time, when you are done plotting the variable associated with a specific timestep and want to traverse through the others.

3. Next we select a variable to plotted. We click on the Plots menu, select the Volume plot and then select the variable tempIN as shown in the Figure 3.10b. The number '1' refers to the material associated with the variable.

4. The variable tempIN/1 now appears on the Active plots window (Figure 3.10c). Select the variable and click Draw.

5. A visualization now appears on the Viewer window, as shown in Figure 3.11a. You can interact with the visualization in terms of rotating it (holding the left mouse button and dragging it),




(a) The window on the gui lists all the timesteps

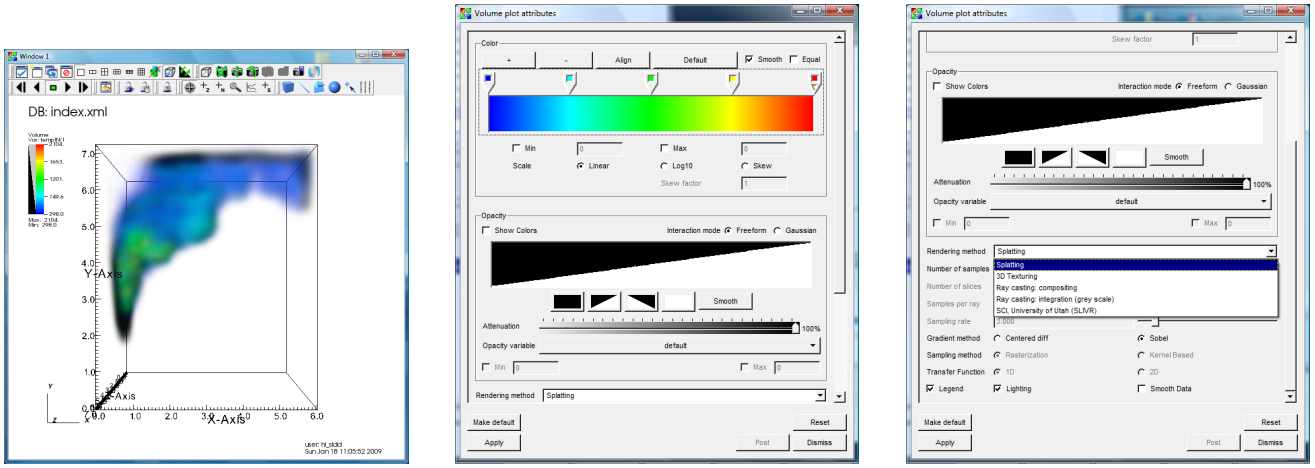
(b) Selecting a volume plot and an associated variable/ material

(c) The list of plots in the Active plots window

Figure 3.10:

zooming in/ out (scrolling the roller on the mouse and/ or selecting the magnifier at the top of the Viewer window) etc.

6. Once you have this basic volume visualization, you can change its attributes by double clicking on the Volume - tempIN/1 plot in the Active plots window. This pops up the Volume plot attributes window (Figure 3.11b and figure 3.11c).




(a) Visualization of a volume on the viewer window

(b) Volume visualization attributes window

(c) Volume visualization attributes window

Figure 3.11:

The tab Color specifies the color table and the various options associated with it. The user can add/ remove control points by clicking on the + and - buttons. These can then be equally spaced by pressing the Align button.

A different color table can be selected by clicking on the Default button and then selecting an appropriate color table. The color(s) associated with the control points can be changed

s

by right-clicking on the them and then selecting an appropriate color.

The user also has the option of specifying a Min and Max on the scalar value range by checking on the associated box(s) and entering in the values.

The second tab Opacity lets you specify a transfer function for the color table. Clicking on the check box Show Colors copies the colors from the color table onto this graph. Selecting the Interaction Mode as Gaussian lets you draw curves and specify a more accurate color table (Figure 3.12).

You can add in as many curves on the graph by clicking on the left mouse button and then placing them accordingly. To delete an unwanted curve, just right click on it.

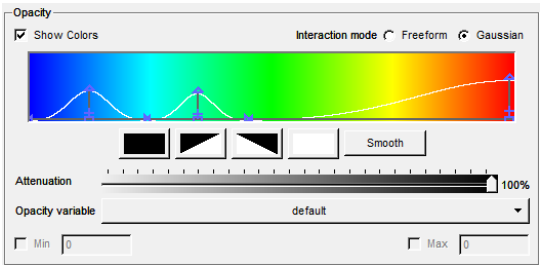



Figure 3.12: The opacity transfer function in the attributes window

After specifying an opacity transfer function, one can select an appropriate rendering method, Splatting being the default. The related fields thereafter become active/ inactive as and when different rendering methods are selected.

### 3.6.2 Particle visualization

1. To add particles, we select the Molecule plot and then click on the variable p.temperature as shown in the Figure 3.13a. The asterisk '*' refers to all the materials associated with the variable.

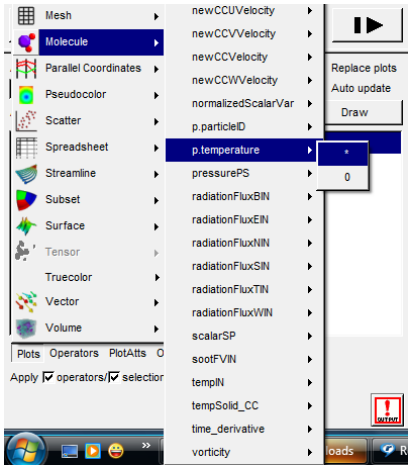
2. The variable p.temperature/* now appears on the Active plots list. Select the variable and hit Draw. A container in the form of particles now appears on the Viewer window.

3. Now double click on the variable name in Active plots list. This brings up the Molecule plot attributes window as shown in Figure 3.13b.

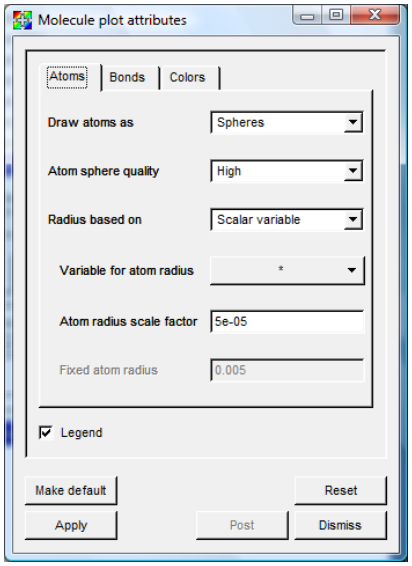
We choose to visualize the particles as Sphere Impostors (doesn't runs the GPU out of memory, drawing as Spheres does). We also choose to scale the sphere radius by a Scalar Variable and specify that variable to be p.temperature/* itself (therefore the * appears). Since the temperature values are too high, we scale them all by a factor of 5.e-05 (on the basis of trial and error). Finally in Colors tab, we set the Color map for scalars as orangehot. Combined with volume visualization, we get a visualization as shown in Figure 3.13c.

### 3.6.3 Visualizing patch boundaries

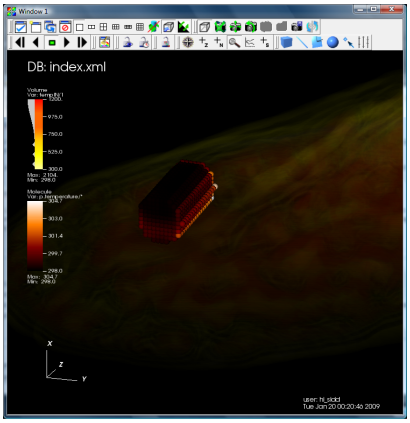
In order to visualize patch boundaries, we use the Subset plot. As with other variables, we select the Subset plot and an associated variable. The variables have a prefix 'level/ patch'. There is a level/ patch variable associated with every kind of variable (Cell Centered, Node Centered, Face Centered) present in the dataset. In the Figure 3.14a, we select one such variable. Next, we hit Draw. This produces a visualization as shown in Figure 3.14b.




(a) Selecting a molecule plot and an associated variable/ material




(b) Selecting a molecule plot and an associated variable/ material




(c) Selecting a molecule plot and an associated variable/ material

Figure 3.13:

To generate a wire-frame model, we double click on the Subset plot in the Active plots window. This pops up the Subset plot attributes window, where we check the Wireframe mode as shown in Figure‘3.15a. This would produce a visualization, similar to one shown in Figure 3.15b.

### 3.6.4 Iso-surfaces

The easiest way to draw iso-surfaces is to use the 'Contour' Plot. As with other plots demonstrated above, the contour plot is selected on a regular 3D scalar variable. Figure 3.16 illustrates this.

Once the plot is selected, we hit 'Draw'. This would produce a visualization, similar to one shown in Figure 3.17a. You can then modify the plot attributes by double clicking on the plot in the 'Active plots' window. This would pop up the 'Contour plot attributes window', as shown in Figure 3.17b.

The 'Select by' option can be changed to 'Value(s)' and 'Percent(s)'. When specifying multiple values, they should be separated by a space.

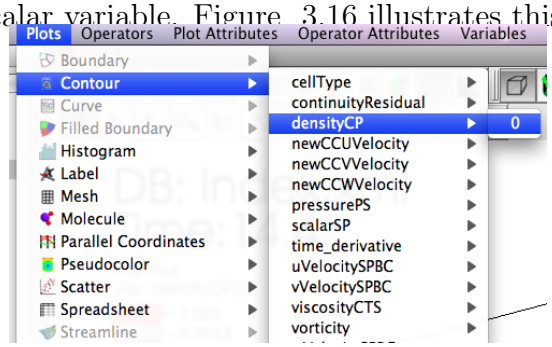
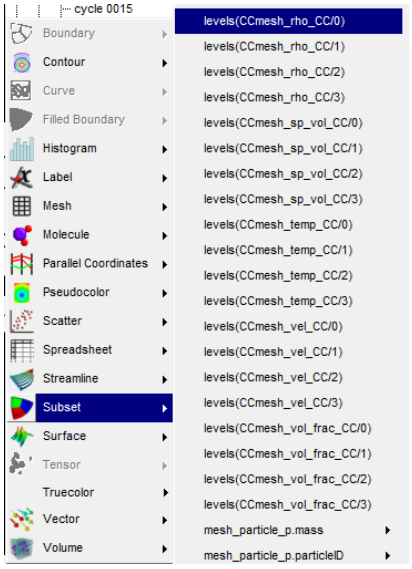



Figure 3.16: Selecting the 'Contour' plot on a regular 3D scalar variable

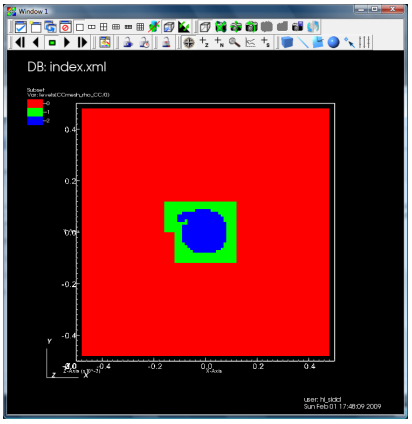
### 3.6.5 Streamlines

The 'Streamlines' plot has issues with the current version of VisIt (1.11.2 and older). However, these have been corrected in the trunk and should be out in version 2.0. The controls remain the same in all these version and the example below was implemented on the trunk version.

As shown in Figure 3.18 we select the 'Streamlines' plot on a vector variable. We then double click on the plot itself, which pops up the 'Streamlines attributes window'.

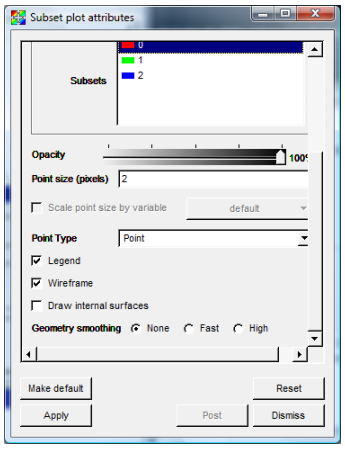



(a) A patch/ level variable, associated with every kind of variable

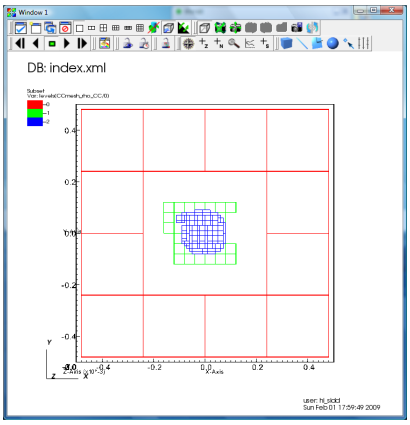



(b) The default visualization of patches

Figure 3.14:




(a) Enabling the 'Wireframe' mode for visualizing patch boundaries




(b) The patch boundaries after enabling the wireframe mode

Figure 3.15:

We set the 'Distance' parameter such that it covers the entire computational domain. We set the 'Streamline direction' as forward. In the 'Source' tab, Figure 3.19a, we define the 'Source type' as 'Line'. We can select other options too, notably 'Single Point', 'Sphere' etc. We now define the line 'Start' and 'End' coordinates. In this specific case, we define them as [-0.1 -0.05 0] and [-0.1 0.05 0] respectively. This choice ensures that we cover the entire y axis and start at the leftmost corner of the computational domain.

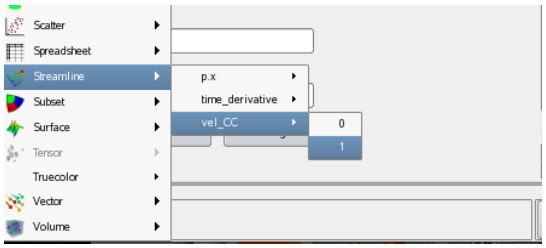
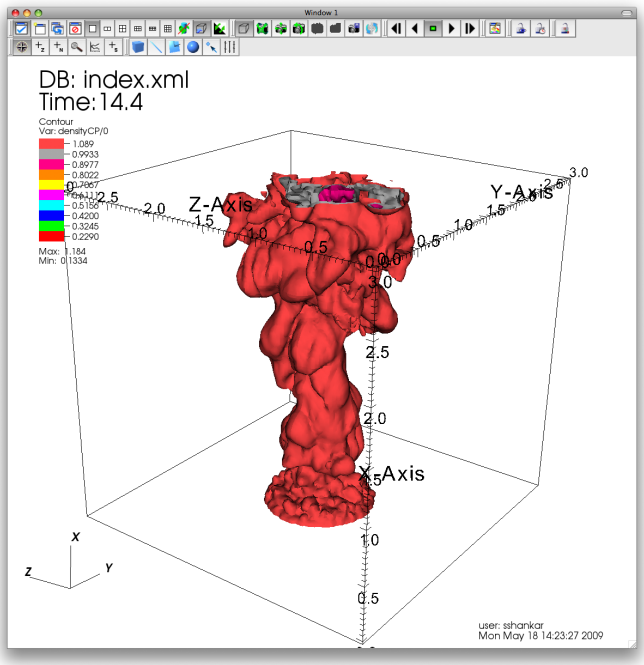
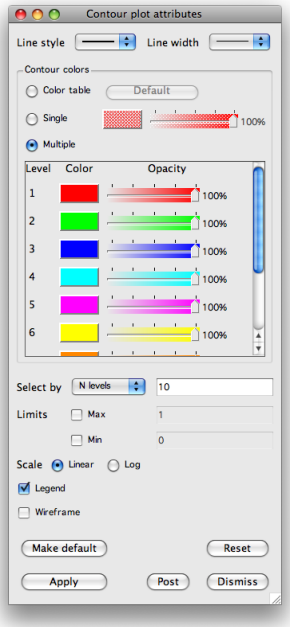



Figure 3.18: Selecting the 'Streamlines' plot on a vector variable

To ensure that our stream lines are smooth, we change the 'Maximum step length' in the in the 'Advanced' tab. In this case, we change it to 1.e-05. The thing to keep in mind is that this length




(a) Iso-surface visualization




(b) The attributes window for the 'Contour' plot

Figure 3.17:

should be order of magnitude smaller than the length of the computational domain. This is shown in Figure 3.19b.

Once these parameters are set, we hit 'Apply' and then click on the 'Draw' button on the gui. This produces a visualization similar to one shown in Figure 3.19c.

### 3.6.6 Visualizing extra cells

For visualizing extra cells we use the 'Inverse Ghost Zone' operator 3.20a in conjunction with the 'Pseudocolor' plot. Since the plugin reads in extra cells as ghost cells, the usage of this operator make sense in this scenario.

After the operator is applied to the 'Pseudocolor' plot, we double click on the operator to change its attributes. We switch to 'Both ghost zones and real zones' in this window 3.20b and hit 'Apply'.

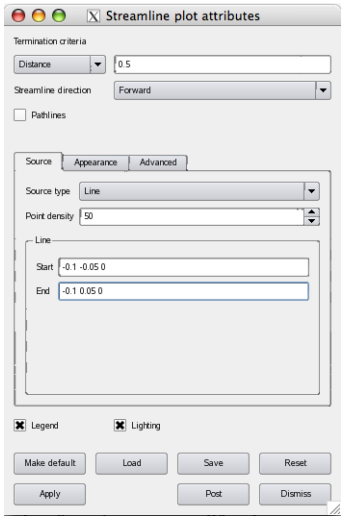
We then hit 'Draw'. When combined with the 'Mesh' plot we get a visualization similar to the one shown in Figure 3.20c. The pick operations on the viewer can then be used to investigate the value(s) in these extra cells.

### 3.6.7 Picking on particles

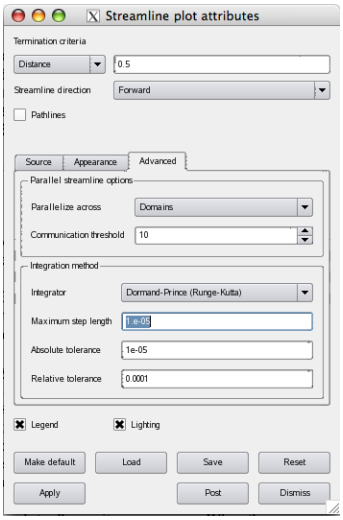
The 'Node pick mode' on the visualization window can be used to pick particles and investigate particles attributes. After plotting particles using the 'Molecule' plot, the user can then select the 'Node pick mode' 3.21 and select particles (by clicking on them) of interest.

Once a particle is picked, the 'Pick' window pops up with the particle attributes. By default only the variable plotted is queried, if the user wants to query more variables per pick - they

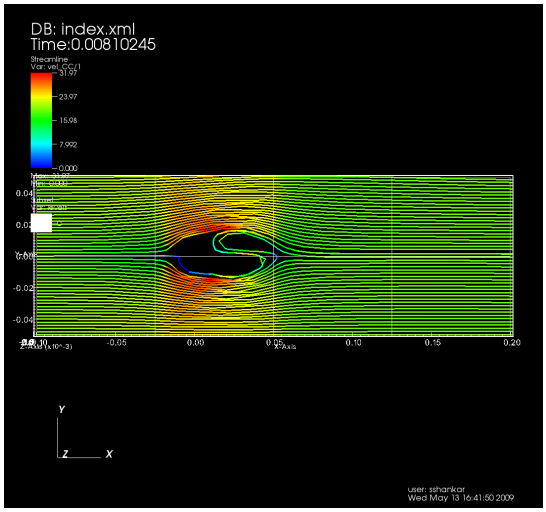

Figure 3.21: The 'Node pick mode' on the visualization window




(a) Setting the 'Source' tab parameters




(b) Setting the 'Advanced' tab parameters




(c) Streamlines visualization

Figure 3.19:

can be added by selecting additional variables from the 'Variables' menu and as shown in the Figure 3.22.

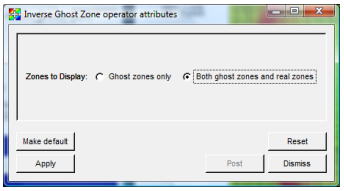
### 3.6.8 Selectively visualizing vectors

The expression editor can be used to define a vector variable with magnitude greater or lesser than a certain extent. An example of this is shown below,

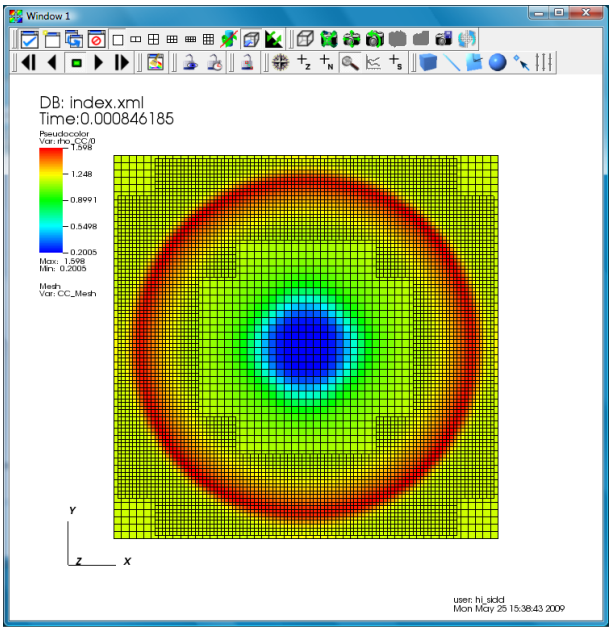
```
if(gt(magnitude(<vel_CC/1>), 0.0), <vel_CC/1>, {0, 0, 0})
```

Put into words, if magnitude of vel_CC/1 is greater than 0.0, display it, else display a zero magnitude vector. To use the 'lesser than' parameter, replace 'gt' with 'lt'.


(a) Selecting the Inverse Ghost Zone operator




(b) The attributes window for the 'Inverse Ghost Zone' operator




(c) Extra cells together with the 'Mesh' plot

Figure 3.20:

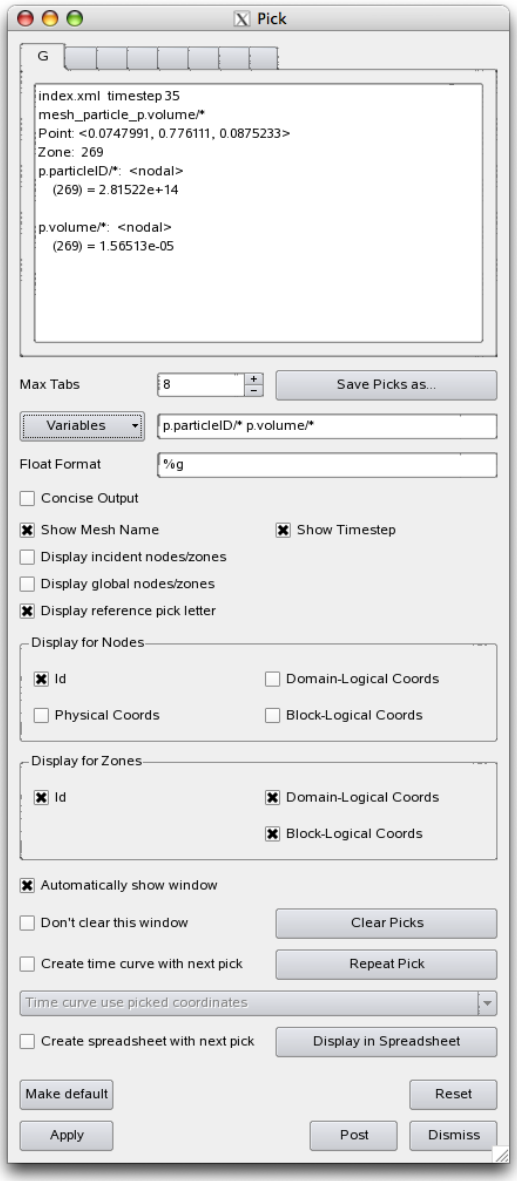



Figure 3.22: The 'Pick' window

# Chapter 4

# Data Extraction Tools

Uintah offers a number of tools for accessing data stored in Uintah Data Archives ("UDAs"). Because the format of Uintah data is specific to the framework, these tools allow a user to quickly extract data, which can then either be postprocessed within that tool (simple modification of the source code may be necessary), postprocessed with external software such as Matlab or Octave, or simply plotted with, e.g. gnuplot. These tools are not compiled automatically when "make sus" is issued. To compile them cd to "opt/StandAlone/tools" and issue "make". These tools are described below.

## 4.1 puda

The command line extraction utility `puda` (for "parse Uintah data archive") has a number of uses. For example, it may be used to extract a subset of particle data from a UDA. Once the extraction tools have been compiled, the puda executable will be located in `opt/StandAlone/tools/puda/.` If the executable is run with no additional command line arguments, the following usage information will be displayed:

```
Usage: puda [options] <archive file>

Valid options are:
  -h[elp]
  -timesteps
  -gridstats
  -listvariables
  -varsummary
  -jim1
  -jim2
  -partvar <variable name>
  -asci
  -tecplot <variable name>
  -no_extra_cells     (Excludes extra cells when iterating over cells.
                       Default is to include extra cells.)
  -cell_stresses
  -rtdata <output directory>
  -PTvar
  -ptonly             (prints out only the point location
  -patch              (outputs patch id with data)
  -material           (outputs material number with data)
  -NCvar <double | float | point | vector>
```

```
  -CCvar <double | float | point | vector>
  -verbose            (prints status of output)
  -timesteplow <int>  (only outputs timestep from int)
  -timestephigh <int> (only outputs timesteps up to int)
  -matl,mat <int>         (only outputs data for matl)
*NOTE* to use -PTvar or -NVvar -rtdata must be used
*NOTE* ptonly, patch, material, timesteplow, timestephigh are used in conjunction with -PTvar.
```

As an example of how to use `puda`, suppose that one wanted to know the locations of all particles at the last archived timestep for the `const_test_hypo.uda` First one may wish to know how many timesteps have been archived. This could be accomplished by:

```
puda -timesteps const_test_hypo.uda
```

The resulting terminal output would be:

```
Parsing const_test_hypo.uda/index.xml
There are 11 timesteps:
1: 1.8257001926347728e-05
548: 1.0012914931998474e-02
1094: 2.0005930425875382e-02
1640: 3.0015616802173569e-02
2184: 4.0005272397960444e-02
2728: 5.0011587657447343e-02
3271: 6.0016178181543284e-02
3812: 7.0000536667661845e-02
4353: 8.0001537138146825e-02
4893: 9.0000702723306208e-02
5433: 1.0001655973087024e-01
```

These represent all of the timesteps for which data has been archived. Suppose now that we wish to know what the stress state is for all particles (in this case two) at the final archived timestep. For this one could issue:

```
puda -partvar p.stress -timesteplow 10 -timestephigh 10 const_test_hypo.uda
```

The resulting output is:

```
Parsing const_test_hypo.uda/index.xml
1.00016560e-01 1 0 281474976710656 -2.72031498e-10 -1.05064208e-26 -2.53781271e-08 -1.05064208e-26 -2.72031
1.00016560e-01 1 1 0 1.93256890e-13 6.56787331e-18 1.85514400e-14 6.56787331e-18 2.24310469e-13 1.85519650e
```

The first column is the simulation time, the third column is the material number, the fourth column is the particle ID, and the remaining nine columns represent the components of the Cauchy stress tensor ($\sigma_{11}$,$\sigma_{12}$,$\sigma_{13}$, ..., $\sigma_{32}$,$\sigma_{33}$). If desired, the terminal output can be redirected to a text file for further use.

## 4.2 partextract

The command-line utility `partextract` may be used to extract data from an individual particle. To do this you first need to know the ID number of the particle you are interested in. This may be done by using the puda utility, or the visualization tools. Once the extraction tools have been compiled, the partextract utility executable will be located in `/opt/StandAlone/tools/extractors/` . If the executable is run without any arguments the following usage guide will be displayed in the terminal:

```
No archive file specified
Usage: partextract [options] <archive file>

Valid options are:
  -mat <material id>
  -partvar <variable name>
  -partid <particleid>
  -part_stress [avg or equiv or all]
  -part_strain [avg/true/equiv/all/lagrangian/eulerian]
  -timesteplow [int] (only outputs timestep from int)
  -timestephigh [int] (only outputs timesteps upto int)
```

As an example of how to use the partextract utility, suppose we wanted to find the velocity at every archived timestep for the particle with ID 281474976710656 (found above using puda) in the "const_test_hypo.uda" file (src/StandAlone/inputs/MPM). The appropriate command to issue is:

```
partextract -partvar p.velocity -partid 281474976710656 const\_test\_hypo.uda
```

The output to the terminal is:

```
Parsing const_test_hypo.uda/index.xml
1.82570019e-05 1 0 281474976710656 0.00000000e+00 0.00000000e+00 -1.00000000e-02
1.00129149e-02 1 0 281474976710656 -1.03554318e-19 -1.03554318e-19 -1.00000000e-02
2.00059304e-02 1 0 281474976710656 -1.99388121e-19 -1.99388121e-19 -1.00000000e-02
        .
        .
        .
```

It is noted that if the stress tensor is output using the partextract utility, the output format is different than for the puda utility. The partextract utility only outputs the six independent components instead of all nine. For example, if we use partextract to get the stress tensor for the same particle as above at the last archived timestep only, the output is:

```
partextract -partvar p.stress -partid 281474976710656 -timesteplow 10 -timestephigh 10 const_test_hypo.uda
Parsing const_test_hypo.uda/index.xml
1.00016560e-01 1 0 281474976710656 -2.72031498e-10 -1.05064208e-26 -2.53781271e-08 -1.05064208e-26 -2.72031
```

Compare this output with the output from puda above. Notice that the ordering of the six independent components of the stress tensor for partextract are $\sigma_{11}$,$\sigma_{22}$, $\sigma_{33}$, $\sigma_{23}$, $\sigma_{13}$ , $\sigma_{12}$.

## 4.3 lineextract

Lineextract is used to extract an array of data from a region of a computational domain. Data can be extracted from a point, along a line, or from a three dimensional region and then stored as a variable for ease of post processing.

Usage:

```
./lineextract [options] -uda <archive file>

Valid options are:
 -h,        --help
 -v,        --variable:      <variable name>
 -m,        --material:      <material number> [defaults to 0]
```

```
-tlow,     --timesteplow:   [int] (sets start output timestep to int) [defaults to 0]
-thigh,    --timestephigh:  [int] (sets end output timestep to int) [defaults to last timestep]
-timestep, --timestep:      [int] (only outputs from timestep int) [defaults to 0]
-istart,   --indexs:        <x> <y> <z> (cell index) [defaults to 0,0,0]
-iend,     --indexe:        <x> <y> <z> (cell index) [defaults to 0,0,0]
-l,        --level:         [int] (level index to query range from) [defaults to 0]
-o,        --out:           <outputfilename> [defaults to stdout]
-vv,       --verbose:       (prints status of output)
-q,        --quiet:         (only print data values)
-cellCoords:                (prints the cell centered coordinates on that level)
--cellIndexFile:             <filename> (file that contains a list of cell indices)
                                 [int 100, 43, 0]
                                 [int 101, 43, 0]
                                 [int 102, 44, 0]
```

The following example shows the usage of lineextract for extracting density data at the 60th computational cell in the x-direction, spanning the width of the domain in the y-direction (0 to 1000), at timestep, 7, (note "timestep" actually refers to the seventh data dump, not necessarily the seventh timestep in the simulation. The variable containing the density data within the uda is "rho_CC," and the output variable that will store the data for post processing is "rho."

```
./lineextract -v rho_CC  -timestep 7 -istart 60 0 0 -iend 60 1000 0 -m 1 -o rho -uda test01.uda.000
```

## 4.4 compute_Lnorm_udas

`Compute_Lnorm_udas` computes the $L_1$, $L_2$ and $L_\infty$ norms for each variable in two udas. This utility is useful in monitoring how the solution differs from small changes in either the solution tolerances, input parameters or algorithmic changes. You can also use it to test the domain size influence. The norms are computed using:

$$d[i] = |uda_1[i] - uda_2[i]| \tag{4.1}$$

$$L_1 = \frac{\sum_i^{\text{All Cells}} d[i]}{\text{number of cells}} \tag{4.2}$$

$$L_2 = \sqrt{\frac{\sum_i^{\text{All Cells}} d[i]^2}{\text{number of cells}}} \tag{4.3}$$

$$L_\infty = \max(d[i]) \tag{4.4}$$

These norms are computed for each `CC, NC, SFCX, SFCY, SFCZ` variable, on each level for each timestep. The output is displayed on the screen and is placed in a directory named 'Lnorm.' The directory structure is:

```
Lnorm/
  -- L-0
   |-- delP_Dilatate_0
   |-- mom_L_ME_CC_0
   |-- press_CC_0
   |-- press_equil_CC_0
   |-- variable
   |-- variable
   |--etc
```

and in each variable file is the physical time, $L_1$, $L_2$ and $L_\infty$. These data can be plotted using `gnuplot` or another plotting program.

The command usage is

```
compute_Lnorm_udas <uda1> <uda2>
```

The utility allows for `udas` that have different computational domains and different patch distributions to be compared. The uda with the smallest computational domain should always be specified first. In order for the norms to be computed the physical times must satisfy

$$|\text{physical Time}_{uda_1} - \text{physical Time}_{uda_2}| < 1e^{-5}.$$

## 4.5 timeextract

Timeextract is used to extract a user specified variable from a point in a computational domain.

Usage:

```
./timeextract [options] -uda <archive file>

Valid options are:
  -h,--help
  -v,--variable <variable name>
  -m,--material <material number> [defaults to 0]
  -tlow,--timesteplow [int] (only outputs timestep from int) [defaults to 0]
  -thigh,--timestephigh [int] (only outputs timesteps up to int) [defaults to last timestep]
  -i,--index <x> <y> <z> (cell coordinates) [defaults to 0,0,0]
  -p,--point <x> <y> <z> [doubles] (physical coordinates)
  -l,--level [int] (level index to query range from) [defaults to 0]
  -o,--out <outputfilename> [defaults to stdout]
  -vv,--verbose (prints status of output)
  -q,--quite (only print data values)
  -noxml,--xml-cache-off (turn off XML caching in DataArchive)
```

The following example shows the usage of timeextract for extracting density data at the computationat cell coordinates 5,0,0, from timestep 0 to the last timestep. The variable containing the density data within the uda is "rho_CC" and the output variable that will store the data for post processing is "rho."

```
./timeextract -v rho_CC -i 5 0 0 -o rho -uda test01.uda.000
```

## 4.6 particle2tiff

particle2tiff is used to extract a user specified particle variable, compute a cell centered average and write that data to a series of tiff slices that can be used in further image processing. Each slice in the tiff file corresponds to a plane in $z$ direction in the computational domain. Each pixel in the tiff image represents a cell in the computational domain. This utility depends on `libtiff4, libtiff4-dev, & libtiffxx0c2`, please verify that they are installed on your system before configuring and compiling.

The data types supported are `double, Vector, Matrix3`, and the equations for computing the cell-centered average are:

$$CC_{ave} = \frac{\sum_{p=1}^{nParticles} Double[p]}{nParticles}$$
$$CC_{ave} = \frac{\sum_{p=1}^{nParticles} Vector[p].length()}{nParticles}$$
$$CC_{ave} = \frac{\sum_{p=1}^{nParticles} Matrix3[p].Norm()}{nParticles}$$

The usage is

```
Usage: tools/extractors/particle2tiff [options] -uda <archive file>

Valid options are:
  -h,        --help
  -v,        --variable:      [string] variable name
  -m,        --material:      [int or string 'a, all'] material index [defaults to 0]

  -max                        [double] (maximum clamp value)
  -min                        [double] (minimum clamp value)
  -orientation                [string] (The orientation of the image with respect to the rows and columns.)
                                      Options:
                                        topleft  0th row represents the ........
                                        topright 0th row represents the ........
                                        botright 0th row represents the ........
                                    default->  botleft  0th row represents the .........
                                        lefttop  0th row represents the ........
                                        righttop 0th row represents the ........
                                        rightbot 0th row represents the ........
                                        leftbot  0th row represents the ........
                                                    Many readers ignore this tag

  -tlow,     --timesteplow:   [int] (start output timestep) [defaults to 0]
  -thigh,    --timestephigh:  [int] (end output timestep) [defaults to last timestep]
  -timestep, --timestep:      [int] (only outputs timestep)  [defaults to 0]

  -istart,   --indexs:        <i> <j> <k> [ints] (starting point, cell index) [defaults to 0 0 0]
  -iend,     --indexe:        <i> <j> <k> [ints] (end-point, cell index) [defaults to 0 0 0]
  -startPt                    <x> <y> <z> [doubles] (starting point in physical coordinates)
  -endPt                      <x> <y> <z> [doubles] (end-point in physical coordinates)

  -l,        --level:         [int] (level index to query range from) [defaults to 0]
  -d,        --dir:           output directory name [none]
  --cellIndexFile:            <filename> (file that contains a list of cell indices)
                                   [int 100, 43, 0]
                                   [int 101, 43, 0]
                                   [int 102, 44, 0]
-------------------------------------------------------------------------------
 For particle variables the average over all particles in a cell is returned.
```

The following example shows the usage of particle2tiff for averaging the particle stress (p.stress) for all materials over the interior cells of the computational domain. A series of 3 tiff slices are saved for every timestep in the directory"output."

```
tools/extractors/particle2tiff -m all -d output -v p.stress -uda disks2mat4patch.uda.000/
There are 14 timesteps
 Initializing time_step_upper to 13
```

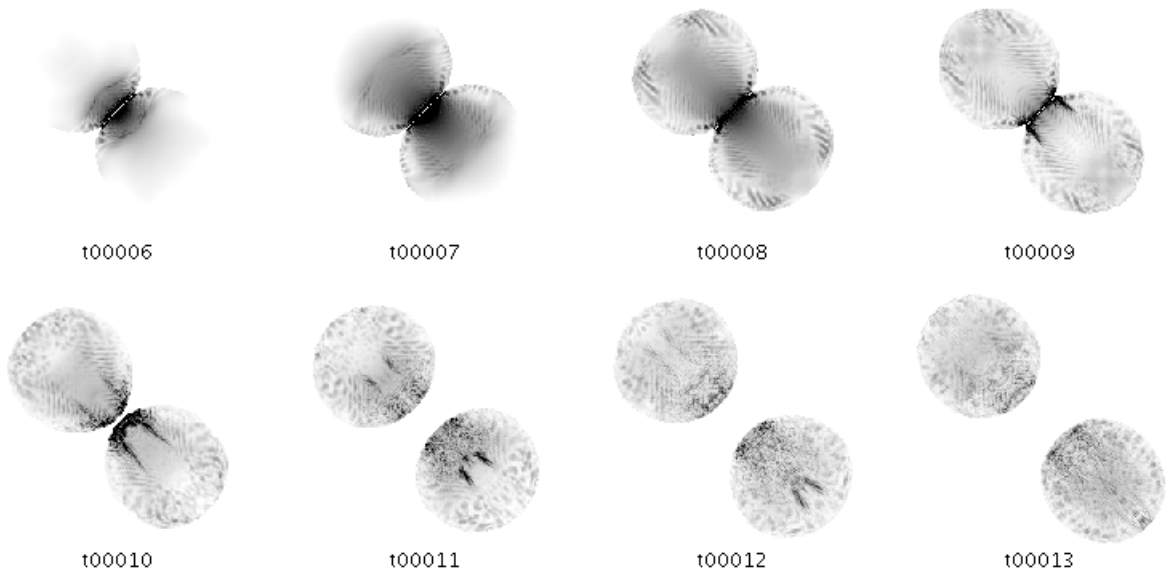



Figure 4.1: Montage showing the averaged particle stress for two cylindrical disks colliding.

```
Removed directory: output
Created directory: output
Timestep[0] = 2.08084e-05
  p.stress: ParticleVariable<Matrix3> being extracted and averaged for material(s): 0, 1,  ........
  writing slice: [0/3] width: 256 height 256
  writing slice: [1/3] width: 256 height 256
  writing slice: [2/3] width: 256 height 256
Timestep[1] = 0.0100184
  p.stress: ParticleVariable<Matrix3> being extracted and averaged for material(s): 0, 1,  ........
  writing slice: [0/3] width: 256 height 256
  writing slice: [1/3] width: 256 height 256
  writing slice: [2/3] width: 256 height 256
Timestep[2] = 0.0200161
  p.stress: ParticleVariable<Matrix3> being extracted and averaged for material(s): 0, 1,  ........
  writing slice: [0/3] width: 256 height 256
  writing slice: [1/3] width: 256 height 256
  writing slice: [2/3] width: 256 height 256
Timestep[3] = 0.0300137
  p.stress: ParticleVariable<Matrix3> being extracted and averaged for material(s): 0, 1,  ........
  writing slice: [0/3] width: 256 height 256
  writing slice: [1/3] width: 256 height 256
  writing slice: [2/3] width: 256 height 256
```

A montage showing the average particle stress computed from particle2tiff is shown in Fig. 4.1. In this simulation two similar disks collided at the center of the domain.

## 4.7 On the fly analysis

On the fly analysis is used to determine the minimum and maximum of specified variables while the simulation is running. Parameters are included in the input file to specify at what frequency the data is analyzed and which variables to look at. A new directory will be made in the uda directory for each level (e.g. L-0). Within the new directory the max and min of each variable for the specified material is determined as a function of time at the given sampling frequency.

| Dynamic Output Intervals Input Parameters | | |
|---|---|---|
| **Tag** | **Type** | **Description** |
| <samplingFrequency> | double | Sampling frequency in times per simulated second |
| <timeStart> | double | Simulation time when sampling begins (sec) |
| <timeEnd> | double | Simulation time when sampling ends (sec) |
| <Variables> | String | Variables to be analyzed including a specification for which material |

The input file specification is as follows:

```
<DataAnalysis>
  <Module name = "minMax">
    <samplingFrequency> 1e8 </samplingFrequency>
    <timeStart> 0 </timeStart>
    <timeEnd> 1000 </timeEnd>
    <Variables>
      <analyze label="press_CC" matl="0"/>
      <analyze label="vel_CC" matl="1"/>
      <analyze label="rho_CC" matl="0"/>
    </Variables>
  </Module>
</DataAnalysis>
```

# Chapter 5

# Order of Accuracy

Order of accuracy is an extremely powerful framework that has been built to run a series of tests where multiple parameters are varied. The framework was originally built to easily do an order of accuracy analysis when cell resolution was varied for a specific simulation. Since then it has been used for a variety of other tasks. The framework allows multiple tests to be easily repeated autonomously, proving to be extremely usefull in parametric studies.

## 5.1 Using The Framework

To run a study using the order of accuracy tools, edit the `components.xml` inside of `test_config_files`. All of the components (ICE MPM MPMICE Arches Wasatch) should be commented out, un-comment any component you wish to use. Next go into the directory for whichever component you un-commented. In this directory are all the available tests for that component and symbolic links to all the associated input files. To run these studies, edit `whatToRun.xml` and un-comment any simulations you would like to run.

Once all of the above steps have been done, you can follow these steps to run your studies.

1. cd to /opt/StandAlone or /dbg/StandAlone
2. Create a symbolic link to orderAccuracy
   `make link_orderAccuracy`
3. Run the following command to run your study

   ```
   orderAccuracy/runTests
   ```

This will create a directory called `order_of_accuracy` which will contain all the studies you have run.

## 5.2 Tests

The order of accuracy framework uses test or .tst files to determine which parameters will be changed. Once a new test has been created you must add it to `whatToRun.xml`. An example .tst file will look something like this.

```
<start>
<upsFile>cavityFlow_dx.ups</upsFile>

<gnuplot>
  <script>plotScript.gp</script>s
  <title>ICE:Cavity Flow Res</title>
  <ylabel>Error</ylabel>
  <xlabel>Resolution</xlabel>
</gnuplot>

<AllTests>
    <replace_lines>
      <maxTime> 20 </maxTime>
    </replace_lines>
    <replace_values>
      /Uintah_specification/Grid/BoundaryConditions/Face[@side='y+']
      /BCType[@id = '0' and @label ='Velocity' and @var='Dirichlet']/value : [1,0,0]
    </replace_values>
</AllTests>
<Test>
    <Title>16</Title>
    <sus_cmd>mpirun -np 6 sus -mpi </sus_cmd>
    <postProcess_cmd>compare_cavityFlow.m -pDir 1 -mat 0 -plot true
    -Re 5000</postProcess_cmd>
    <x>16</x>
    <replace_lines>
      <resolution>  [16,16,1]     </resolution>
    </replace_lines>
</Test>
<Test>
    <Title>32</Title>
    <sus_cmd>mpirun -np 6 sus -mpi</sus_cmd>
    <postProcess_cmd>compare_cavityFlow.m -pDir 1 -mat 0 -plot true
    -Re 5000</postProcess_cmd>
        <x>32</x>
    <replace_lines>
      <resolution>  [32,32,1]     </resolution>
    </replace_lines>
</Test>
<Test>
    <Title>64</Title>
    <sus_cmd>mpirun -np 6 sus -mpi</sus_cmd>
    <postProcess_cmd>compare_cavityFlow.m -pDir 1 -mat 0 -plot true
    -Re 5000</postProcess_cmd>
    <x>64</x>
    <replace_lines>
```

```
      <resolution>  [64,64,1]     </resolution>
    </replace_lines>
</Test>

</start>
```

The following is a list of required parameters for the .tst files. These fields must be present, but many of them may remain empty.

- <start>
- <upsFile>
  This points the .tst to the input file, only one input file can be specified per test.
- <gnuplot>
  This is a generic gnuplot script which can be used to plot any data output through the post processing script. It was created and originally configured to compute the L2 norm, but can be modified to plot anything by modifying `plotScript.gp` in the directory with the .tst file.
- <AllTests>
  Items put in this section will modify all simulations.
- <Test>
  Items put in this section will modify only this simulation.
- <Title>
  Title of the current test. Many names used for files and directories will be based off this title.
- <sus_cmd>
  This is the sus command line you would normally give to run an input file, the input and output files are put in automatically and can be omitted.
- <postProcess_cmd> :
  Optional: This line is used to feed the current tests uda and output file into a post processing script. The post processing scripts have their own inputs, and can be modified when the scripts are created. The one thing all compare scripts share as an input is -uda <uda name> and -o <output file> as these are automatically fed into the compare script by the framework, and must be included with all new post processing scripts.
- <x>
  this sets the x label for the gnuplot script

This section is for the two optional parameters, <replace_lines> and <replace_values>. These two parameters are what specify the changes between the tests. These can both be placed inside <Test> and <AllTests>.

- <replace_lines>
  This parameter is used to replace specific lines in the input files, and can be used when there is a unique tag such as <resolution>. To use this, just replace the line with the one you would like. This method is the simplest way to replace section of an input file.

- <replace_values>
  This section is for when a tag is not unique and a path to the correct place must be given. In the above example, the velocity value is not unique, it is located inside each face, thus the face must be specified. XML uses unique paths to find tags inside of other tags, this path must be given in order to replace the values. In order to find these tags the command `xmlstarlet` can be run on the input file, and the paths will be given.

## 5.3 Additional information

Order of accuracy has a large amount of small intricacies to learn to fully utilize it, and this section hopes to familiarize you with some things you might not have thought possible.

### 5.3.1 Post Processing Tools

Inside `orderAccuracy` is a directory called `postProcessTools`. This contains all the post processing scripts that are used in the current tests. All of these can be run on their own, but were created to be fed through the system. These scripts have generally been used to pull data out of the uda and compare it to either experimental or exact data. The scripts then plot this data with a unique filename, and output the L2 norm to a file. If you going to use a postprocessing script in a .tst file, be familiar with the script, and which arguments are needed for it to run correctly. Many postprocessing scripts have already been implimented, so be sure to become familiar with those that are already available.

### 5.3.2 Gnuplot scripts

Gnuplot is used in many sections of the order of accuracy framework, and learning where and how it uses it can be extremely helpful. plotScript.gp is the main gnuplot script that you can use to plot any data from your post processing script. This script may require a lot of manipulation to get what you want, but due to its integration with the system its much easier to work with. A few other gnuplot scripts are lying around to plot assorted data output through the post processing scripts. These scripts must be used after the fact.

### 5.3.3 Debugging

Many times an error you have is not brought up to the highest level and you will not see it. The easiest way to find an error is to familiarize yourself with the order of operations the system performs, and locate the point where the process ended. Sometimes the simulation will crash, and it may not be apparent what has happened. Be sure to check the output files of the latest test if something goes wrong.

# Chapter 6

# Uda Management

Uintah offers a number of tools for managing Uintah Data Archives ("UDAs"). These tools are especially useful for large simulations with large output. These tools are used for quickly moving data, reducing the size and number of variables within your UDA and combining multiple UDAs.

## 6.1 pscp2

pscp2 is a tool to quickly transfer data in parallel between two machines. In order for this to work there must be an password-less connection between the two machines. pscp2 is located in /src/scripts/udaTransferScripts/
The usage is

```
./pscp2 <# processors>
        <transfer entire uda (y/n)>
        <remove remote directory (y/n)>
        < name of local directory>
        <login@remote machine>:<remote path>
```

The following example shows the usage of pscp2 for transferring 1.uda.000 from the local machine to the home directory on ember using 8 processors. It will remove any files in the home directory on ember with the same title of 1.uda.000 but the original copy on the local machine will not be removed.

```
./pscp2 8 y n 1.uda.000 username@ember.chpc.utah.edu:/home/
```

## 6.2 Make Master Uda

Make master UDA is a tool used to combine multiple UDAs. This will be useful when a simulation must be restarted multiple times and post processing is required. Instead of having to do post processing on each individual UDA, make master UDA will combine all of the UDAs to allow for continuous analysis of the simulation start to finish. In order for this to work the UDAs must be in sequential order, it does not mater what the UDA number is as long as they go in order. This tool is found in /src/scripts/makeMasterUda.csh. When using this script it must be able to see /src/scripts/makeCombinedIndex.sh. This tool uses links to combine all of the UDAs, once the script is complete the original UDAs must stay were they are so the new master UDA can find the data. No changes will be made to the original UDAs.
The usage is

1. mkdir <masterUda> This is where all of the UDAs will be linked together
2. cd <masterUda>
3. makeMasterUda.csh ../uda.000 ../uda.001 ../uda.00N

## 6.3 pTarUda

pTarUda is a tool used to create/extract compressed tar files of each timestep in a UDA, including the checkpoints directory. pTarUda was designed to run in parallel and works well on udas with a large number of files and/or uncompressed data. What it will buy you is faster moves, copies and transfers since the OS doesn't have to process as many files. For uncompressed UDAs you may see up to a 30% reduction in size. The original timestep directories are not deleted unless the -deleteOrgTimesteps option is used. This tool is found in /src/scripts/udaTransferScripts.
Usage:

```
pTarUda -<create/extract> -uda <UDA directory name>

  Options:
  -np <int>:                 Number of processors. Default is 10
  -allTimesteps <y/n>:       Operate on all directories in uda? Default is yes.
                             If "n" then a vi window will open allowing you
                             to edit a list of timesteps to archive/extract.
  -deleteOrgTimesteps        Delete original timestep directories after they have
                             been tarred/untarred.
  -continueTarring:          Continue tarring/untarring if previous attempts failed
  -help:                     Display options summary
```

# Chapter 7

# Arches

## 7.1 Introduction

The ARCHES component was initially designed for predicting the heat-flux from large buoyant pool fires with potential hazard hazards immersed in or near a pool fire of transportation fuel. Since then, this component has been extended to solve many industrially relevant problems such as industrial flares, oxy-coal combustion processes, and fuel gasification.

The ARCHES component solves the conservative, finite volume, compressible, low-mach formulation of the Navier-Stokes equation with a pressure projection that includes the effect of variable density, reaction, and heat transfer modes in the gas phase including radiation. Given the wide range of length and time scales that are present in many combustion problems of interest, ARCHES utilizes models for bridging the molecular (micro) scales to the full, large (macro) scales. This bridging occurs through the use of subgrid and resolved scale mixing, reaction, and turbulence models. For example, momentum turbulence closure is accomplished by using various large eddy simulation (LES) type closure models. Fast chemistry scales are modeled by preprocessing the full chemical mechanism, modeled in various idealized configurations (e.g., equilibrium or flamelets), then tracking a set of reduced parameters on the LES mesh that map the full thermochemical state-space. In general, the chemistry is tabulated and stored in a reaction table. Subgrid turbulence species mixing processes and included by using presumed PDF methods and using models for certain moments of the distribution (e.g., scalar variance and mixture fraction). The turbulence-species subgrid interaction model description are termed mixing models. The chemistry and mixing models are usually completely preprocessed together into one tabular format to give a mixing table.

Note that the sister component, MPMARCHES, couples an MPM description of a solid object to include stationary solids with and without conjugate heat transfer. While run as a separate component, MPMARCHES is simply a wrapped version of ARCHES to include the MPM interface.

The following gives a brief introduction to a few of the key concepts of the Arches formulation of the transport equations and physical models as well as the CFD algorithm. Following this explanation, an overview of the ARCHES input parameters are given.

## 7.2 Governing Equations

The essential governing equations for the Arches component, written in finite volume form, include the mass balance, momentum balance, mixture fraction balance, and energy balance equations. Using a bold-face symbol to represent a vector quantity, the equations are:

1. The mass balance,
$$\int_V \frac{\partial \rho}{\partial t} dV + \oint_S \rho \mathbf{u} \cdot d\mathbf{S} = 0 \ , \tag{7.1}$$
where $\rho$ is density and $\mathbf{u}$ is the velocity vector.

2. The momentum balance,
$$\int_V \frac{\partial \rho \mathbf{u}}{\partial t} dV + \oint_S \rho \mathbf{u}\mathbf{u} \cdot d\mathbf{S} = \oint_S \tau \cdot d\mathbf{S} - \int_V \nabla p dV + \int_V \rho \mathbf{g} dV \ , \tag{7.2}$$
where $\tau$ is the deviatoric stress tensor defined as $\tau_{ij} = 2\mu S_{ij} - \frac{2}{3}\mu \frac{\partial u_k}{\partial x_k}\delta_{ij}$, the second isotropic term in $\tau_{ij}$ is absorbed into the pressure projection for the current low-Mach scheme, and $S_{ij} = \frac{1}{2}\left(\frac{\partial u_i}{\partial x_j} + \frac{\partial u_j}{\partial x_i}\right)$. Also in Equation 7.2, $\mathbf{g}$ is the gravitational body force and $p$ is pressure.

3. The mixture fraction balance,
$$\int_V \frac{\partial \rho f}{\partial t} dV + \oint_S \rho \mathbf{u} f \cdot d\mathbf{S} = \oint_S D \nabla f \cdot d\mathbf{S} \ , \tag{7.3}$$
where $f$ is the mixture fraction and a Fick's law form of the diffusion term assuming equal diffusivities results in a single diffusion coefficient, $D$.

4. The thermal energy balance,
$$\int_V \frac{\partial \rho h}{\partial t} dV + \oint_S \rho \mathbf{u} h \cdot d\mathbf{S} = \oint_S k \nabla h \cdot d\mathbf{S} - \oint_S q \cdot d\mathbf{S} \ , \tag{7.4}$$
where $h$ is the sum of the chemical plus sensible enthalpy, $q$ is the radiative flux, a Fourier's law form of the conduction term is used with a diffusion coefficient, $k$, and the pressure term is neglected.

These equations are solved in an LES context, meaning filters are applied to the equations. Here, we use Favre filtering, defined as
$$\overline{\phi} = \frac{\overline{\rho\phi}}{\overline{\rho}},$$
to isolate the density in the filtered equations. The filtering operations result in the classic turbulence closure problem and thus models are required.

Consider a control volume, $V$, with surface area $S$. Because the equations will be solved on a computational grid, one can safely assume that the the control volume has $N$ faces, where unique faces are identified with their index, $k$. The discussion is further simplified by only considering cubic volumes with length $h$. Given the cubic control volume, a surface-filtered field for a variable $\phi$ is defined as $\overline{\phi}^{(j)}(\mathbf{x})$, where the variable is filtered on a plane in the $x_j$ orthogonal direction. Then, for any surface, $k$, the field is sampled at the face centered location. For example, if $j = 1$, the surface-filtered quantity is
$$\overline{\phi}^{2d,(1)}(\mathbf{x}) = \frac{1}{h^2} \int_{x_2-h/2}^{x_2+h/2} \int_{x_3-h/2}^{x_3+h/2} \phi(\mathbf{x}') dx'_2 dx'_3 \ . \tag{7.5}$$
The volume average follows as
$$\overline{\phi}^{3d}(\mathbf{x}) = \frac{1}{h^3} \int_{x_1-h/2}^{x_1+h/2} \int_{x_2-h/2}^{x_2+h/2} \int_{x_3-h/2}^{x_3+h/2} \phi(\mathbf{x}') dx'_1 dx'_2 dx'_3 \ . \tag{7.6}$$

The bars over the variable, $\phi$, are labeled with '2d' and '3d' superscripts to distinguish between the two filters. Pope [10] identifies the proceeding definitions as using the "anisotropic box" filter kernel where the resultant variables are simply averages over the intervals $x_j - \frac{1}{2}h < x'_j < x_j + \frac{1}{2}h$.

For convenience in isolating density in the filtered equations, a Favre-filtered quantity is defined for an arbitrary variable, $\varphi$, as

$$\widetilde{\varphi}^{2d} \equiv \frac{\overline{\rho\varphi}^{2d}}{\overline{\rho}^{2d}} \ , \tag{7.7}$$

and

$$\widetilde{\varphi}^{3d} \equiv \frac{\overline{\rho\varphi}^{3d}}{\overline{\rho}^{3d}} \ . \tag{7.8}$$

Because the 2d and 3d filters are explicitly defined, this convention is slightly different than what is normally observed in the literature. Most literature, however, derives the filtered equations from the finite difference equations rather than the finite volume equations. Thus, using $\overline{\rho}^{2d}$ and $\overline{\rho}^{3d}$ in Equations 7.7 and 7.8 to stress surface and volume filtered densities are appropriate for the present discussion.

The previous definitions are applied to the integral forms of the governing equations to obtain the Favre-filtered LES equations. Nevertheless, there are terms in the Favre-filtered equations that cannot be solved. These include the surface filtered convection of momentum, $\widetilde{u_i u_j}^{2d}$, the surface filtered convection of mixture fraction, $\widetilde{u_j f}^{2d}$, and the surface filtered convection of enthalpy, $\widetilde{u_j h}^{2d}$.

For the filtered velocity product, $\overline{\rho}^{2d}\ \widetilde{u_i u_j}^{2d}$, a subgrid stress tensor is defined as,

$$\tau_{ij}^{sgs} = \widetilde{u_i u_j}^{2d} - \widetilde{u}_i^{2d}\widetilde{u}_j^{2d} \ . \tag{7.9}$$

Similarly, subgrid diffusion terms are defined for mixture fraction and enthalpy,

$$\mathcal{J}^f = \widetilde{u_j f}^{2d} - \widetilde{u}_j^{2d}\widetilde{f}^{2d} \ , \tag{7.10}$$

$$\mathcal{J}^h = \widetilde{u_j h}^{2d} - \widetilde{u}_j^{2d}\widetilde{h}^{2d} \ . \tag{7.11}$$

$$\tag{7.12}$$

Using these definitions, the final form of the Favre-filtered equations is

1. The filtered mass balance,

$$\frac{d}{dt}\left(\widetilde{\rho}^{3d}\right) + \frac{S_k}{V} n_{kj}\left(\overline{\rho}^{2d}\ \widetilde{u}_j^{2d}\right) = 0 \ . \tag{7.13}$$

2. The filtered momentum balance,

$$\frac{d}{dt}\left(\overline{\rho}^{3d}\ \widetilde{u}_i^{3d}\right) = \frac{S_k}{V} n_{kj}\left(-\overline{\rho}^{2d}\ \widetilde{u}_i^{2d}\widetilde{u}_j^{2d} + \overline{\tau}_{ij}^{2d} + \tau_{ij}^{sgs} - \overline{p}^{2d}\delta_{ij}\right) + \overline{\rho}^{3d}\ g_i \ . \tag{7.14}$$

3. The filtered mixture fraction balance,

$$\frac{d}{dt}\left(\overline{\rho}^{3d}\ \widetilde{f}^{3d}\right) = \frac{S_k}{V} n_{kj}\left(-\overline{\rho}^{2d}\ \widetilde{u}_j^{2d}\widetilde{f}^{2d} + D\nabla\overline{f}^{2d} + \mathcal{J}^f\right) \ . \tag{7.15}$$

4. The filtered thermal energy balance,

$$\frac{d}{dt}\left(\overline{\rho}^{3d}\ \widetilde{h}^{3d}\right) = \frac{S_k}{V} n_{kj}\left(-\overline{\rho}^{2d}\ \widetilde{u_j}^{2d}\widetilde{h}^{2d} + k\nabla\overline{h}^{2d} - \overline{q}^{2d} + \mathcal{J}^h\right) \ . \tag{7.16}$$

The subgrid momentum stress, $\tau_{ij}^{sgs}$, the subgrid mixture fraction dissipation, $\mathcal{J}^f$, and the subgrid heat dissipation, $\mathcal{J}^h$, contain the unresolved or subgrid action of the turbulence on the transported quantities. Since these terms arise from definitions, models are introduced to include the subgrid effects that they represent. These models are discussed next.

### 7.2.1 Subgrid Turbulence Models

The construction of both $\mathcal{J}^f$ and $\mathcal{J}^h$ is relatively straight forward. Invoking an "eddy-viscosity" modeling concept, the subgrid transport due to turbulent advection is treated as an enhanced diffusion term for the unclosed terms listed above. That is, the subgrid mixture fraction dissipation and subgrid enthalpy dissipation are respectively written as,

$$\mathcal{J}^f = D_t \frac{\partial \overline{f}^{2d}}{\partial x_j} \,, \tag{7.17}$$

and

$$\mathcal{J}^h = k_t \frac{\partial \overline{h}^{2d}}{\partial x_j} \,. \tag{7.18}$$

To model $D_t$ and $k_t$, constant turbulent Schmidt ($Sc_t$),

$$Sc_t = \frac{1}{\rho} \frac{\mu_t}{D_t} \,, \tag{7.19}$$

and Prandlt ($Pr_t$),

$$Pr_t = \frac{1}{\rho} \frac{\mu_t}{k_t} \,, \tag{7.20}$$

numbers are assumed with where $\mu_t$ is a turbulent viscosity. Following Pitsch and Steiner [9], the values of the turbulent Schmidt and Prandlt number are taken as $Sc_t = Pr_t = 0.4$, which is consistent with a unity Lewis number assumption.

For the subgrid scale stress tensor, $\tau_{ij}^{sgs}$, two common LES turbulence closure models are the constant coefficient Smagorinsky model [12] and the dynamic coefficient Smagorinsky model [6]. As with the scalar subgrid modeling terms, the eddy viscosity model is again invoked for $\tau_{ij}^{sgs}$. Defining the deviatoric subgrid stress tensor as,

$$\tau_{ij}^{d,sgs} = \tau_{ij}^{sgs} - \frac{1}{3} \tau_{kk}^{sgs} \delta_{ij}, \tag{7.21}$$

the subgrid stress is taken as,

$$\tau_{ij}^{d,sgs} \approx -2\nu_t \overline{S}_{ij} = -2(C_s \Delta)^2 |\overline{S}| \overline{S}_{ij} \,, \tag{7.22}$$

where $\Delta$ is the filter width, $\nu_t$ is the eddy viscosity and $|\overline{S}| \equiv (2\overline{S}_{ij}\overline{S}_{ij})^{1/2}$. For the Smagorinsky model, $C_s \approx 2$ depending on the filter type, numerical method, and flow configuration [10].

For the dynamic Smagorinsky model, $C_s$ is computed by taking a least squares approach to determining the length scale [5],

$$(C_s \Delta)^2 = \frac{\langle \mathcal{L}_{ij} M_{ij} \rangle}{\langle M_{ij} M_{ij} \rangle} \,, \tag{7.23}$$

where

$$\mathcal{L}_{ij} = 2(C_s \Delta)^2 \widehat{|\overline{S}| \overline{S}}_{ij} - 2(C_s \widehat{\Delta})^2 |\widehat{\overline{S}}| \widehat{\overline{S}}_{ij} \,, \tag{7.24}$$

and

$$M_{ij} \equiv 2\left(\widehat{|\overline{S}|\overline{S}}_{ij} - \alpha^2|\widehat{\overline{S}}|\widehat{\overline{S}}_{ij}\right) \ . \tag{7.25}$$

The hat defines an explicit test filter and the angled brackets in Equation 7.33 conceptually represent an averaging over a homogeneous region of space that, experience has shown, is necessary for stability. Experience has also shown that averaging over the test filter width is adequate and the filter width ratio, $\alpha = \widehat{\Delta}/\Delta$, is usually taken to be 2.

### 7.2.2 Subgrid Momentum Dissipation

Addressing the momentum closure involves finding a suitable model for the subgrid scale stress tensor, $\tau_{ij}^{sgs}$. Two common LES turbulence closure models are examined: the constant coefficient Smagorinsky model and the dynamic coefficient Smagorinsky model. In LES modeling, field variables are decomposed into a spatially filtered field and a residual component, $u = \overline{u} + u'$. This decomposition is known as a Leonard decomposition. While seemingly similar to a Reynolds decomposition used in Reynolds Averaged Navier-Stokes (RANS) models, the Leonard decomposition has the property that the filtered residual component is generally not equal to zero, $\overline{u'} \neq 0$. As a result, the subgrid stress term contains several terms,

$$\begin{aligned} \tau_{ij}^{sgs} &= \overline{(\overline{u}_i + u_i')(\overline{u}_i + u_i')} - \overline{u}_i\overline{u}_j \ , \\ &= \underbrace{\overline{\overline{u}_i\overline{u}_j} - \overline{u}_i\overline{u}_j}_{L_{ij}} + \underbrace{\overline{\overline{u}_i u_j'} + \overline{u_i'\overline{u}_j}}_{C_{ij}} + \underbrace{\overline{u_i' + u_j'}}_{R_{ij}} \ , \end{aligned} \tag{7.26}$$

referred to as the Leonard stress, the cross stresses, and the Reynolds stress respectively.

It is useful to consider the physical interpretation of the various components of the stress. The Leonard term is responsible for filtering and projecting the nonlinear interactions of the resolved components back to the finite LES space. This is a correction to the resolved advective term in accordance with the stated explicit filter used to derive the LES equations. It does not account for aliasing errors. The first cross term represents advection of the resolved field by turbulent fluctuations. The second cross term represents the advection of subgrid scales by the resolved field. The Reynolds stress is familiar from RANS and represents the advection of subgrid scales by turbulent fluctuations.

As with the scalar subgrid modeling terms, the eddy viscosity model is again invoked for $\tau_{ij}^{sgs}$. The most common eddy viscosity model in LES is the Smagorinsky model [12]. Defining the deviatoric subgrid stress tensor as,

$$\tau_{ij}^{d,sgs} = \tau_{ij}^{sgs} - \frac{1}{3}\tau_{kk}^{sgs}\delta_{ij}, \tag{7.27}$$

the subgrid stress is approximated by,

$$\tau_{ij}^{d,sgs} \approx -2\nu_t\overline{S}_{ij} = -2(C_s\Delta)^2|\overline{S}|\overline{S}_{ij} \ , \tag{7.28}$$

where, $\Delta$ is the filter width, $\nu_t$ is the eddy viscosity, $|\overline{S}| \equiv (2\overline{S}_{ij}\overline{S}_{ij})^{1/2}$, and typically $C_s \approx 2$ depending on the filter type, numerical method, and flow configuration [10]. This model is basically identical to Prandtl's mixing length model with $l = C_s\Delta$.

The dynamic procedure [2, 6] eliminates the need to specify the model constant, $C_s$, a priori, with the basic assumption that the constant is the same for two different filter scales. The smaller scale is historically referred to as the "grid scale" (though the filter width need not equal the grid

spacing, $\Delta \geq h$)), and the larger scale is referred to as the "test scale". Implicit in this assumption is the requirement that both scales lie within the inertial subrange.

Defining the deviatoric residual stress tensor as,

$$T_{ij}^d = T_{ij} - \frac{1}{3} T_{kk} \delta_{ij} \ , \tag{7.29}$$

the residual stress at the test scale is given by,

$$T_{ij}^d \equiv \widehat{\overline{u_i u_j}} - \widehat{\overline{u}}_i \widehat{\overline{u}}_j \approx -2(C_s \widehat{\Delta})^2 |\widehat{\overline{S}}| \widehat{\overline{S}}_{ij} \ . \tag{7.30}$$

where $\widehat{\Delta}$ is the test filter width and the hat defines an explicit test filter. By test filtering Equation 7.9 and combining this with 7.30, one can construct the Leonard term, $\mathcal{L}_{ij}$. This is also known as the "Germano identity",

$$\mathcal{L}_{ij} = T_{ij} - \widehat{\tau^{sgs}}_{ij} = \widehat{\overline{u_i u_j}} - \widehat{\overline{u}}_i \widehat{\overline{u}}_j \ . \tag{7.31}$$

Notice that the Leonard term is directly computable from resolved LES quantities. By restating the Smagorinsky model in terms of the Germano identity, one ends up with an over-determined system of equations for the unknown, $C_s$,

$$\mathcal{L}_{ij} = 2(C_s \Delta)^2 \widehat{|\overline{S}| \overline{S}}_{ij} - 2(C_s \widehat{\Delta})^2 |\widehat{\overline{S}}| \widehat{\overline{S}}_{ij} \ . \tag{7.32}$$

Although we have pulled $C_s$ out of the test filtering operation of the subgrid stress, this approximation yields acceptable results. In practice, one takes a least squares approach to determining the length scale [5],

$$(C_s \Delta)^2 = \frac{\langle \mathcal{L}_{ij} M_{ij} \rangle}{\langle M_{ij} M_{ij} \rangle} \ , \tag{7.33}$$

where

$$M_{ij} \equiv 2 \left( \widehat{|\overline{S}| \overline{S}}_{ij} - \alpha^2 |\widehat{\overline{S}}| \widehat{\overline{S}}_{ij} \right) \ . \tag{7.34}$$

The only model parameter, then, is the filter width ratio, $\alpha = \widehat{\Delta}/\Delta$, usually taken to be 2.

The angled brackets in Equation 7.33 conceptually represent averaging over a homogeneous region of space which, experience has shown, is necessary for stability. We have found that averaging over the test filter width is adequate. With these implementation practices, the dynamic model is generally robust. The implementation can be made more efficient by computing the constant roughly every 10 time steps (based on the advective CFL), and only for the first Runge-Kutta step.

### 7.2.3 LES Algorithm

The set of filtered equations (Equations 7.13-7.16) are discretized in space and time and solved on a staggered, finite volume mesh. The staggering scheme consists of four offset grids. One grid stores the scalar quantities and the remaining three grids store each component of the velocity vector. The velocity components are situated so that the center of their control volume is located on the face centers of the scalar grid in their respective direction. Figure 7.1 shows an example of a two-dimensional grid and the staggering arrangement.

The staggering arrangement is advantageous for computing low-Mach LES reacting flows. First, since a pressure projection algorithm is used, the velocities are exactly projected without interpolation error because the location of the pressure gradient coincides directly with the location of the velocity storage location. Second, Morinishi et al. [7] showed that kinetic energy is exactly

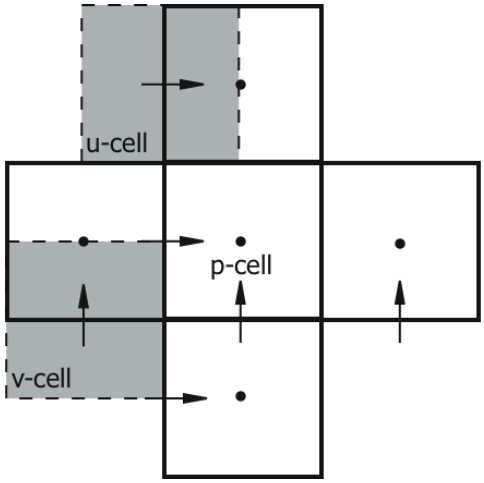



Figure 7.1: Staggered grid arrangement in two dimensions with u and v velocity cell centers located on the face centers of the scalar cells.

conserved when using a central differencing scheme on the convection and diffusion terms without a subgrid model and in combination with a staggered grid. Having a spatial scheme that conserves kinetic energy is advantageous because it limits artificial dissipation that arises from the differencing scheme. These conservation properties make the staggered grid a prime choice for LES reacting flow simulation.

For the spatial discretization of the LES scalar equations, flux limiting and upwind schemes for the convection operator are used. These schemes are advantageous for ensuring that scalar values remain bounded. For the momentum equation, a central differencing scheme for the convection operator is used. All diffusion terms are computed with a second order approximation of the gradient.

When computing the 2d surface filtered field on the faces of the control volume, one is forced to use an interpolation from the 3d volume filtered field. This approximation is tolerated because computing the 2d surface field is simply not possible with the given grid scheme.

An explicit time stepping scheme is chosen. A general, multistep explicit update for a variable, $\phi$, may be written as,

$$\begin{aligned} \phi^0 &= \phi^n \ , \\ \phi^{(i)} = V \sum_{k=0}^{m-1} \left( \alpha_{i,k} \phi^{(k)} + \Delta t \beta_{i,k} L(\phi^{(k)}) \right) \ , &\quad i = 1, ..., m \\ \phi^{(m)} &= \phi^{n+1} \ , \end{aligned} \tag{7.35}$$

where $n$ is the time level, $m$ is the substep between $n$ and $n+1$, $\alpha$ and $\beta$ are integration coefficients, and $L$ is a linearization operator on the the convective flux and source terms. Letting $m = 1$ and $\alpha = \beta = 1$ the forward-Euler time integration scheme is determined,

$$(\phi)^{n+1} = (\phi)^n + \Delta t (L(\phi)^n) \ . \tag{7.36}$$

A higher order, multistep method is derived by letting $m > 1$ and choosing appropriate constants for $\alpha$ and $\beta$. For this study, two step and three step, strong stability preserving (SSP) coefficients were chosen from Gottlieb et al. [3].

Using the coefficients given by Gottlieb et al., the SSP-RK 2 stepping scheme is

$$(\phi)^{(1)} = (\phi)^n + \Delta t(L(\phi)^n) \tag{7.37}$$
$$(\phi)^{n+1} = \frac{1}{2}(\phi)^n + \frac{1}{2}(\phi)^1 + \frac{1}{2}\Delta t(L(\phi)^{(1)}) \ .$$

SSP-RK 3 time stepping scheme is,

$$(\phi)^{(1)} = (\phi)^n + \Delta t(L(\phi)^n) \tag{7.38}$$
$$(\phi)^{(2)} = \frac{3}{4}(\phi)^n + \frac{1}{4}(\phi)^{(1)} + \frac{1}{4}\Delta t(L(\phi)^{(1)})$$
$$(\phi)^{(n+1)} = \frac{1}{3}(\phi)^n + \frac{2}{3}(\phi)^{(2)} + \frac{1}{4}\Delta t(L(\phi)^{(2)}) \ .$$

The time step is limited by

$$\Delta t \leq c\Delta t_{F.E.} \tag{7.39}$$

where $\Delta t_{F.E.}$ is the forward-Euler time step limited by the Courant-Friedrichs-Levy condition and $c$ is a constant less than or equal to one.

A higher order, multistep method is derived by letting $m > 1$ and choosing appropriate constants for $\alpha$ and $\beta$. For this study, two step and three step, strong stability preserving (SSP) coefficients were chosen from Gottlieb et al. [3]. The coefficients for SSP-RK 2 and SSP-RK 3 are optimal in the sense that the scheme is stable when $c = 1$ if the forward-Euler time step is stable for hyperbolic problems. In practice, for the Navier-Stokes equations, the value of $c$ is taken less than one.

Choosing an explicit time stepping scheme, rather than an implict one, creates a challenge for solving the set of equations. The density at the $n + 1$ timestep, which is required to determine the cardinal variables, requires an estimation. Taking the estimated density for $\overline{\rho}^{n+1}$ to be $\overline{\rho}^*$, the estimation can be as simple as $\overline{\rho}^* = \overline{\rho}^n$. Note that the 2d and 3d filter distinction is dropped for the remainder of this discussion for the sake of simplicity. A slightly more complicated procedure involves a forward-Euler step of the continuity equation to obtain $\overline{\rho}^*$. This is written as,

$$\overline{\rho}^* = \overline{\rho}^n - \Delta t \frac{S_k}{V} n_{kj} \left(\overline{\rho}\widetilde{u_j}\right) \ . \tag{7.40}$$

Ideally, one would like to know $\overline{\rho}^{n+1}$ rather than an estimate. While more details will be discussed in Section **??**, one recognizes that $\rho$ is a function of the same variables that are being updated in time, namely, the mixture fraction, $f$, and enthalpy, $h$. This quandary is a result of the explicit time stepping method will not be resolved for variable density flows without using a fully implicit method. Explicit methods, however, do have advantages, especially for large scale parallel computations. Specifically, explicit methods are easier to load balance because the amount of work required for each processor is readily determined a priori, which makes for an efficient parallel computation. Explicit methods are also easier to code into a computer and to debug. For these reasons, the current algorithm discussion is limited to explicit methods only.

The explicit algorithm for solving the set of filtered equations is shown in Algorithm 1.

### 7.2.4 Direct Quadrature Method of Moments

The direct quadrature method of moments (DQMOM) is a recently-developed moment method for tracking distributions. It has been applied to distributions of evaporating droplets, soot particle distributions, fluidized beds, and subgrid chemistry PDFs. The method is similar to the quadrature

**Algorithm 1** Explicit LES algorithm.

**for** $t = t_{min}...t_{max}$ **do**
  **for** $RK_{step} = 1...N$ **do**
    Solve for scalars products $(\overline{\rho}\widetilde{f})^{n+1}$ and $(\overline{\rho}\widetilde{h})^{n+1}$.
    Estimate $\overline{\rho}^* = \overline{\rho}^{n+1}$ from Equation 7.40
    **if** $\overline{\rho}^* < \overline{\rho}_{min}$ or $\overline{\rho}^* > \overline{\rho}_{max}$ **then**
      $\overline{\rho}^* = \overline{\rho}^n$
    **end if**
    Compute $\widetilde{f}^{n+1} = (\overline{\rho}\widetilde{f})^{n+1}/\overline{\rho}^*$ and $\widetilde{h}^{n+1} = (\overline{\rho}\widetilde{h})^{n+1}/\overline{\rho}^*$
    Compute $\overline{\rho}^{n+1} = f(\widetilde{f}^{n+1}, \widetilde{h}^{n+1})$
    Compute $\widetilde{\mathbf{u}}^*$, the unprojected velocities
    Perform RK averaging if needed
    Compute correct pressure from pressure poisson equation
    Project velocities with correct pressure to get $\widetilde{\mathbf{u}}^{n+1}$
  **end for**
**end for**

method of moments (QMOM), in that it uses quadrature to provide closure for the moment transport equation; however, it differs in that the moment transport equations are not actually solved, unlike the quadrature method of moments (QMOM). Rather, the DQMOM tracks the quadrature weights and weighted abscissas representing the NDF directly, rather than using the product-difference algorithm to transform between a set of moments and quadrature weights and abscissas that would best represent that distribution.

The basic outline of the method, given below, defines and covers these fundamental steps and concepts:

1. Number Density Function (NDF)
2. Moments
3. Moment Methods for NDF Transport
4. Quadrature
5. Direct Quadrature Method of Moments

The number density function is the starting point, as it is the function of interest that is being tracked. Moments are defined, and moment methods are explained and applied to the NDF transport equation to yield the moment transport equation. Furthermore, quadrature is defined and applied to approximate the moments, which leads to a quadrature-approximated moment transport equation. This equation leads to the fundamental equations governing DQMOM.

### Number Density Function

Using the direct quadrature method of moments (DQMOM) in Arches, the dispersed phase is represented as a number density function (NDF), which is tracked in a stationary Eulerian reference frame. The NDF is denoted as $f$ and represents the number of particles at a particular point in space and time. Using DQMOM, this NDF is parameterized on several different variables - independent

variables for the particles. These are called "internal coordinates," and are denoted by $\boldsymbol{\xi}$ (where boldface denotes a vector quantity). In this case the NDF is written as $f(\boldsymbol{\xi}; \boldsymbol{x}, t)$.

The starting point for the DQMOM equations is the NDF transport equation; this is derived in several places and will not be derived here. The NDF transport equation is:

$$\begin{aligned} \frac{\partial f(\boldsymbol{\xi}; \boldsymbol{x}, t)}{\partial t} + \frac{\partial}{\partial x_i}\left(\langle u_i | \boldsymbol{\xi}; \boldsymbol{x}, t\rangle f(\boldsymbol{\xi}; \boldsymbol{x}, t)\right) & \\ + \frac{\partial}{\partial \xi_j}\left(\langle G_j | \boldsymbol{\xi}; \boldsymbol{x}, t\rangle f(\boldsymbol{\xi}; \boldsymbol{x}, t)\right) &= h(\boldsymbol{\xi}; \boldsymbol{x}, t), \end{aligned} \tag{7.41}$$

where $G_j$ is the velocity of the NDF in phase-space (that is, internal coordinate-space), and is defined by:

$$G_j = \frac{d\xi_j}{dt} \tag{7.42}$$

(note this is analogous to spatial velocity $v$, defined by:

$$v_i = \frac{dx_i}{dt}, \tag{7.43}$$

and that $G_j$ takes the same form as Lagrangian particle models); also, $h$ is a birth and death term representing the appearance or disappearance of particles within the domain. Note also that the velocities $v_i$ and $G_j$ are conditioned on the value of the internal coordinates, as well as on space and time. This implies that the spatial and phase-space velocities are full distributions in $(\boldsymbol{\xi}, \boldsymbol{x})$ space, just as the NDF is.

The number of internal coordinates is denoted by $N_\xi$. If $N_\xi = 1$, the NDF is called "univariate"; if $N_\xi > 1$, then the NDF is called "multivariate".

### Moments

Using DQMOM, the particles are represented as an Eulerian distribution - that is, the particles are not treated in an individual sense, but in a statistical sense. In order to represent the NDF in the framework of a scalar CFD code, the NDF must be represented using a set of scalars that can be transported. One such set of scalars, the moments of the NDF, provide useful statistical information about the distribution. Additionally, a distribution can be approximately reconstructed from its moments. The first moment of the distribution represents the mean value; the second, the variance of the distribution; and so on. An arbitrary number of moments can be defined. For a univariate NDF, the $k^{th}$ moment of the NDF is defined as:

$$m_k = \frac{\int_{-\infty}^{+\infty} \xi^k f(\xi; \boldsymbol{x}, t)\, d\xi}{\int_{-\infty}^{+\infty} f(\xi; \boldsymbol{x}, t)\, d\xi} \tag{7.44}$$

Alternatively, if the NDF is a function of several internal coordinates, the $\boldsymbol{k}^{th}$ moment is defined

as:

$$m_{\boldsymbol{k}} = \frac{\int_{\infty}^{+\infty} \cdots \int \xi_1^{k_1} \ldots \xi_{N_\xi}^{k_{N_\xi}} f\left(\xi_1, \ldots, \xi_{N_\xi}\right) d\xi_1 \ldots d\xi_{N_\xi}}{\int_{\infty}^{+\infty} \cdots \int f\left(\xi_1, \ldots, \xi_{N_\xi}\right) d\xi_1 \ldots d\xi_{N_\xi}} \tag{7.45}$$

$$m_{\boldsymbol{k}} = \frac{\int_{-\infty}^{+\infty} \cdots \int \left[\left(\prod_{m=1}^{N_\xi} \xi_m^{k_m}\right) f(\boldsymbol{\xi}; \boldsymbol{x}, t)\, d\boldsymbol{\xi}\right]}{\int_{-\infty}^{+\infty} \cdots \int f(\boldsymbol{\xi}; \boldsymbol{x}, t)\, d\boldsymbol{\xi}} \tag{7.46}$$

where $\boldsymbol{k} = \left[k_1, k_2, \ldots k_{N_\xi}\right]$ is the multivariate moment index.

### Moment Methods for NDF Transport

The method of moments is a method of tracking the NDF of a system of particles. Because the NDF is a full, continuous distribution, it is difficult to track without assuming a functional form for it. Rather than assume a functional form, the moments of the NDF, which are simply scalars, are tracked instead. This method requires tracking various scalars, which is computationally feasible in a scalar framework and which greatly simplifies the process of tracking the NDF. However, the approach has a closure problem that prevents it from being used in practice for any but the most simple systems.

The transport equation for each moment must be written in terms of higher order moments, and the transport equations for these higher order moments must be written in terms of successively higher order moments, etc. Simplifications (models) must be used to express higher order moments only in terms of lower order moments being tracked as a part of the method of moments. Once this is accomplished, the set of moment transport equations becomes a closed set of equations.

### Quadrature

Quadrature approximates the integral of an unknown function with tabulated known values as a summation of a set of $N$ weighted abscissas. It determines a polynomial of degree $2N-1$ whose zeros are the $N$ weighted abscissas, and approximates the unknown function using this polynomial [Press 1992]. There are several common quadrature formulations, including the midpoint rule (the unknown function is assumed to be a constant, or zero-order polynomial), the trapezoid rule (the unknown function is assumed to be a straight line, or first-order polynomial), and Simpson's rule (the unknown function is assumed to be a second-order polynomial). Note that while the unknown function does not have to be a polynomial, the quadrature approximation becomes much better if it is (and exact if the unknown function is a polynomial of degree $2N-1$ or less). The general $N$-point quadrature formula can be written as:

$$\int_a^b w(r) g(r)\, dx \approx \sum_{\alpha=1}^{N} w_\alpha g(r_\alpha) \tag{7.47}$$

where $g(r)$ is an arbitrary function of the variable $r$. As $N$ increases, the quadrature approximation usually becomes more accurate. This equation can also be extended to a multivariate function $g(\mathbf{r})$, an arbitrary function of the $D$-element vector $\mathbf{r} = [r_1, r_2, \ldots, r_D]$ to yield:

$$\int_a^b w(\mathbf{r}) g(\mathbf{r}) d\mathbf{r} \tag{7.48}$$

The weights are common to all internal coordinates $\mathbf{r}$ because the weight function $w(\mathbf{r})$ is binned into $N$ discrete weights, and this weight function is common to all internal coordinates.

### Quadrature-Approximated Moment Transport Equation

When the quadrature approximation is applied to a multivariate NDF (where the NDF is the weight function), it yields:

$$\begin{aligned} f(\boldsymbol{\xi}; \mathbf{x}, t) &\approx \sum_{\alpha=1}^{N} \left( w_\alpha(\boldsymbol{x}, t)\, \boldsymbol{\delta}\left(\boldsymbol{\xi} - \langle \boldsymbol{\xi}(\boldsymbol{x}, t) \rangle_\alpha \right) \right) \\ &\approx \sum_{\alpha=1}^{N} \left( w_\alpha(\boldsymbol{x}, t) \prod_{j=1}^{N_\xi} \delta\left(\xi_j - \langle \xi_j(\boldsymbol{x}, t) \rangle_\alpha \right) \right) \end{aligned} \tag{7.49}$$

which makes the integral of the NDF:

$$\int_{-\infty}^{+\infty} \boldsymbol{\xi}^{\boldsymbol{k}} f \, d\boldsymbol{\xi} \approx \sum_{\alpha=1}^{N} \left\{ w_\alpha \left( \prod_{j=1}^{N_\xi} \langle \xi_j \rangle_\alpha^{k_j} \right) \right\} \tag{7.50}$$

This quadrature-approximated NDF can then be substituted into the moment definition to get a quadrature-approximated moment,

$$m_{\boldsymbol{k}} \approx \sum_{\alpha=1}^{N} \left\{ p_\alpha \left( \prod_{j=1}^{N_\xi} \langle \xi_j \rangle_\alpha^{k_j} \right) \right\} \tag{7.51}$$

where $p_\alpha$ is the probability of environment $\alpha$ (defined as $w_\alpha / \sum_{i=1}^{N} w_i$), $\boldsymbol{k}$ is the multivariate moment index vector, and $k_j$ is the $j^{th}$ element of vector $\boldsymbol{k}$ (with a total of $N_\xi$ values).

Starting with the NDF transport equation, the moment transform can be taken, yielding the moment transport equation. Next, the quadrature approximation can be plugged in, which yields the quadrature-approximated moment transport equation. After some algebraic manipulation, this yields:

$$\sum_{\alpha=1}^{N}\left[\prod_{j=1}^{N_\xi}\delta\left(\xi_j-\langle\xi_j\rangle_\alpha\right)+\sum_{m=1}^{N}\frac{\partial}{\partial\langle\xi_m\rangle_\alpha}\left(\prod_{j=1}^{N_\xi}\delta\left(\xi_j-\langle\xi_j\rangle_\alpha\right)\right)\langle\xi_m\rangle_\alpha\right]a_\alpha$$
$$-\sum_{\alpha=1}^{N}\sum_{n=1}^{N}\left[\frac{\partial}{\partial\langle\xi_n\rangle_\alpha}\left(\prod_{j=1}^{N_\xi}\delta\left(\xi_j-\langle\xi_j\rangle_\alpha\right)\right)\right]b_{n\alpha}=$$
$$\sum_{\alpha=1}^{N}\sum_{m=1}^{N_\xi}\sum_{n=1}^{N_\xi}\left[\frac{\partial^2}{\partial\langle\xi_m\rangle_\alpha\partial\langle\xi_n\rangle_\alpha}\left(\prod_{j=1}^{N_\xi}\delta\left(\xi_j-\langle\xi_j\rangle_\alpha\right)\right)\right]C_{mn\alpha}$$
$$+S_\xi+D_\xi+h, \quad (7.52)$$

where $a_\alpha$ and $b_{n,\alpha}$, defined as:

$$\frac{\partial}{\partial t}\left(w_\alpha\right)+\frac{\partial}{\partial x_i}\left(\langle u_i\rangle_\alpha w_\alpha\right)-\frac{\partial}{\partial x_i}\left(D_{x,\alpha}\frac{\partial}{\partial x_i}\left(w_\alpha\right)\right) = a_\alpha \quad (7.53)$$
$$\frac{\partial}{\partial t}\left(\varsigma_{n\alpha}\right)+\frac{\partial}{\partial x_i}\left(\langle u_i\rangle_\alpha\varsigma_{n\alpha}\right)-\frac{\partial}{\partial x_i}\left(D_{x,\alpha}\frac{\partial}{\partial x_i}\left(\varsigma_{n,\alpha}\right)\right) = b_{n\alpha}, \quad (7.54)$$
$$\text{for } j=1\ldots N_\xi$$

are the transport equations implemented to track the evolution of the NDF. The source terms $a_\alpha$ and $b_{n,\alpha}$ come from the solution to the linear system

$$\mathbf{Ax}=\mathbf{B}. \quad (7.55)$$

**Build of Ax=B**

There are $N_e(N+1)$ unknowns for $a_a$ and $b_{na}$, while there are just one linear equations for them. In order to build a complete linear system, $N_e(N+1)-1$ moments are needed. And DQMOM method is used to build this linear system. Using properties of delta functions, the $k^{th}$ moment for Eq.(7.53) is derived as

$$\sum_{a=1}^{N_e}\left[\left(\prod_{n=1}^{N}\left\langle\xi_n^{k_n}\right\rangle_a\right)-\sum_{m=1}^{N}k_m\left(\left\langle\xi_m^{k_m}\right\rangle_a\right)\left(\prod_{n=1,n\neq m}^{N}\left\langle\xi_n^{k_n}\right\rangle_\alpha\right)\right]a_a$$
$$+\sum_{a=1}^{Ne}\sum_{n=1}^{N}\left[k_n\left\langle\xi_n^{k_n-1}\right\rangle_a\left(\prod_{m=1,m\neq n}^{N}\left\langle\xi_m^{k_m}\right\rangle_a\right)\right]b_{a,n}=$$
$$\sum_{a=1}^{Ne}\sum_{m=1}^{N}\left[k_m\left(k_m-1\right)\left\langle\xi_m^{k_m-2}\right\rangle_a\times\left(\prod_{j\neq m,j=1}^{N_\xi}\left\langle\xi_j^{k_j}\right\rangle_a\right)\right]w_aC_{mma}+ \quad (7.56)$$
$$\sum_{a=1}^{Ne}\sum_{m=1}^{N}\sum_{n=1}^{N}\left[k_mk_n\left\langle\xi_m^{k_m-1}\right\rangle_a\left\langle\xi_n^{k_n-1}\right\rangle_a\times\left(\prod_{j=1,j\neq m}^{N}\left\langle\xi_j^{k_j}\right\rangle_a\right)\right]w_aC_{mna}+\bar{S}_k+\bar{D}_k$$

and

$$\bar{S}_k = -\sum_{a=1}^{N_e}\sum_{n=1}^{n}\left[\left(\prod_{m=1,m\neq n}^{N}\left\langle\xi_m^{k_m}\right\rangle\right)\left(k_n\left\langle\xi_n^{k_n-1}\right\rangle_a\right)\left(w_a\left\langle G_n\right\rangle_a\right)\right] \tag{7.57}$$

$$\bar{D}_k = \sum_{a=1}^{N_e}\sum_{n=1}^{n}\left(\prod_{m=1,m\neq n}^{N}\left\langle\xi_m^{k_m}\right\rangle_a\right)\left(k_n\left(k_n-1\right)\left\langle\xi_n^{k_n-2}\right\rangle\right)\left(w_a\Gamma_{\xi_n,a}\right) \tag{7.58}$$

where $k_m$ represent the $k^{th}$ moment for the $m^{th}$ internal coordinates. With a reasonable selection of the moments for each internal coordinates, a linear system with a rank of $N_e\left(N+1\right)$ can be constructed by

$$\left[A_1, A2\right] * \left[a_a, b_{an}\right]^{\mathrm{T}} = \left[C_k + \bar{S}_k + \bar{D}_k\right]^{\mathrm{T}} \tag{7.59}$$

**solution of DQMOM matrix**

The detailed solution method can be find in Charles and Julien's Thesis [11, 8]. The typical LU solution will offten met singular problems, so Rodney provided a method to rebuild matrix A and B to avoid this problem[1]. This method is named as "Optimize" in ¡DQMOM¿ block. However, the "Optimize" method is very time consuming.

**Simplest method in DQMOM**

In order to save CPU times for DQMOM method, some simplified method can be adopted to define the source term for 7.53, or $X$. For coal combustion case, when the following assumption was adopted

- Assuming that any scalars distribution of the coal particles can be perfectly represented by multi-delta functions, especially, the particle velocity distribution can be perfectly represented by $\langle v_a\rangle$, so that there is no diffusion terms. In other words, the third term at the right side and $D_\xi$ in Eq.(7.59), are omited. This is only correct for each single particle, so that integral of Eq.(7.52) with a moment becomes the Langrangian particle equations as:

$$\frac{d\xi^m}{dt} = S_{\xi^m} \tag{7.60}$$

where $\xi^m$ is the moment that we are interested. As long as we consider a reprentative particles for a group of particles, such diffusion term will exists, which are hard to be modeled. A reasonable way to eliminate the effect of diffusion term is to increase the phase numbers. (but this will meet other chanllenges, for e.g., for the dense phase, the interactions between two phases need to be considered, the computation cost would be hudge).

- For pulverized coal combustion system, the particles are very dispersed, so that any particle-particle interactions is small enough (comparing partile-gas interactions) to be omited. With this assumption, the source term, $S_\xi$ can be easily represented by the single particle models, so that Langrangian style models can be used.

- There is no particle death or birth, so that the weights do not change with the internal coordinates $\xi$. In other words, $a_a$ in Eq.(7.53) equals to 0 and the rank of the linear system is reduced from $N_e(N+1)$ to $N_eN$.

Taking these three assumptions, the linear system can be reduced further, since there is no diffusion terms, the right side of Eq.(ref{eq:kth_moment}) is reduced to $\bar{S}_k$. Then we just consider the first moments for $\xi_{n,a}$ and $0^{th}$ moments for other $\xi$(let $k_n = 1, and k_(m \neq n) = 0$ for the $p^{th}$ phase. This makes the linear system, Eq.(**??**) reduces to

$$0 + \ldots + 0 + b_{n,a} + 0 + \ldots 0 = w_a \langle G_n \rangle_a = w_a \frac{d\xi_{n,a}}{dt} \tag{7.61}$$

That means the linear system is already solved without any further calculations (no need of solving linear systems). and the transport equations of the weights and weighted abscissas finally are simplified to

$$\frac{\partial}{\partial t}(w_a) + \frac{\partial}{\partial x_i}(\langle v_i \rangle_a w_a) = 0 \tag{7.62}$$

$$\frac{\partial}{\partial t}(\zeta_{n,a}) + \frac{\partial}{\partial x_i}(\langle v_i \rangle_a \zeta_{n,a}) = w_a \frac{d\xi_{n,a}}{dt} \tag{7.63}$$

It should be noted these are instantaneous equations, and SGS model should be considered for LES simulation. If we know the diffusion models, and just with no birth/dirth of particles. There are also no need to solve the transport equations, if only the internal coordinates are independent, or if we can build the direct correlation among these dependent moments. For e.g., if we have to consider the particle surface and the particle diameter as well as the particle volume, these internal coordinates are correlated, and linear system is needed to be built for such dependent moments if the particle diameter, surface and volume change can't be decribed by three separate models. For the coal combustion systems we are considering, the internal coordinates are independent, so there is no need to solve the linear system. and the final transport equations are

$$\frac{\partial}{\partial t}(w_a) + \frac{\partial}{\partial x_i}(\langle v_i \rangle_a w_a) + \frac{\partial}{\partial x_i}\left(D_{x,p}\frac{\partial}{\partial x_i}(w_a)\right) = 0 \tag{7.64}$$

$$\frac{\partial}{\partial t}(\zeta_{n,a}) + \frac{\partial}{\partial x_i}(\langle v_i \rangle_a \zeta_{n,a}) + \frac{\partial}{\partial x_i}\left(D_{x,a}\frac{\partial}{\partial x_i}(\zeta_{n,a})\right) = w_a \frac{d\xi_{n,a}}{dt} + D_{\xi_{n,a}} \tag{7.65}$$

The chanllenge is that $D_{x,a}$ and $D_{\xi_{n,a}}$ almost can't be modeled (seems $D_{\xi_{n,a}} = 0$ when multi-delta functions adopted, according to Eq.(7.58). and we'd rather to consider more phases to elimate the diffusion terms.

Taking the assumptions mentioned above, the transport equations of PBE in form of PDF acutally degenerate to multi-phase Euler-Euler equations without consideration phase-phase interactions (except for gas-particle interactions). If the particle phase are enough so that each particle is represented by a phase, Eq(7.62) and Eq(7.65) are actually Lagrangian method. Or in other words, the simplest case we adopted with DQMOM method equvalent that we are using Lagrangian method to track infinitely large number of particles falling to only $Ne$ categories (each categories of particle has the same values for all $N$ internal coordinates).

**Summary**

In summary, the DQMOM solution procedure requires solving several transport equations, given by 7.53. These transport equations describe the changes in the quadrature weights and weighted abscissas used to approximate the NDF. The source terms for these transport equations, $a_\alpha$ and $b_{n,\alpha}$, come from the solution to a linear system, $\mathbf{Ax} = \mathbf{B}$, in which the vector $\mathbf{x}$ contains the source

terms. This linear system comes from the quadrature-approximated moment trasport equation, equation 7.52, of which there are $(N_\xi + 1)\, N$. It is because this set of equations is linear in the unknowns $a_\alpha$ and $b_{n,\alpha}$ that it can be re-cast in the form of a linear system.

### 7.2.5 Wall Heat Transfer Model in Arches

#### Basic assumptions and models for wall heat transfer

The basic assumptions are as follows:

- Wall is regarded as black body, the heat released by the wall is calculated by Stefan-Boltzmann law, this assumption is consistant to the assumption adopted in DO radiation method in ARCHES.
- There is no time derivation term for wall heat transfer, in other words, wall temperature comes to equilibrium state instantaneously at each time step. While a relaxation coefficient, $C_{rex}$, is adopted to smooth the wall temperature change, or $T_w = (1 - C_{rex})T_{w0} + C_{rex}T_w$.
- When considering convection heat transfer, the gas temperature is represent by the Cell-Center gas temperature adjecent to the wall, Pr and Re is calculated based on the velocities and properties of the CC value adjecent to the wall. But up to now, no convection heat transfer is considered in ARCHES.
- thermal resistance from fluid to the tubes is small enough to be neglected, so that inside tube wall temperature is set to a constant value. (¡tube_side_T¿ in arches.spec)

The wall heat transfer is calculated by

$$q_{net} = (T_w - T_{in})/R_{wc} \tag{7.66}$$

$$q_{net} = \epsilon_w(q_{in} - q_{rad\_w}) + q_{conv} \tag{7.67}$$

where $q_{net}$ is the heat flux absorbed by the wall. $q_{in}$ is the incidental radiation heat flux from the ambient gas phase to the wall surface, which is provided by radiation solver in ARCHES, $q_{rad\_w}$ is the heat flux of the black-body wall emission, and can be calculated by

$q_{rad\_w} = \sigma T_w^4$

$q_{conv}$ is the heat flux absorbed by the wall through convection, $T_w$ and $T_{in}$ are the wall surface temperature (represented by 'temperature' in ARCHES simulation) and fluid temperature inside wall tubes (¡tube_side_T¿), and $R_{wc}$ is the wall thermal resistance, $R_{wc} = l/\lambda$, $l$ is the ¡wall_thickness¿ in areches.spec, and $\lambda$ is ¡k¿ in arches.spec.

#### Algorithms of wall heat transfer solution in Arches

Eq.7.66 can be recasted by

$$T_w = T_{in} + q_{net}R_{wc} \tag{7.68}$$

Based on Eq.7.68, an iterative algorithm was proposed to solve $T_w$:

1. Taking an initial $T_w$ and calculate $q_{net}$ by Eq.(7.67)

2. Calculate $T_w$ by Eq.7.68. And return back to Step.1

Since $q_{net}$ is sensitive to $T_w$ due to Stefan-Boltzmann law, this iteration method is unstable and it's hard to get the final resolutions. So an relaxation coefficient was introduced to help converge. Then the final iteration method would be

1. calculate $q_{net}$ based on $T_{w0}$ by Eq.7.66
2. Calculate $T_{w1}$ with Eq.7.68
3. Calculate renewed $T_{w1}$ by $T_{w1} = (1 - C_{relx})T_{w0} + c_{relx}T'_w$. Then go back to step1 with $T_{w1}$

This Relaxation Iteration Method (RIM) can be used to get the final $T_w$ at each timestep. Although with small Relaxation Coefficient, RIM method may still be unstable if $R_{wc}$ is very large.

$T_w$ can also be calculated by another iterations method such as Dichotomy Iteration Method (DIM)as follows:

1. To set a maximum wall temperatrue $T_{wmax}$ and a minimum wall temperature $T_{wmin}$.
2. to calculate initial wall temperature $T_{w0} = (T_{wmax} + T_{wmin})/2$
3. To calculate $T_{w1}$ with Eq7.66, Eq7.67 and Eq7.68.
4. if $T_{w1} > T_{w0}$, then $T_{wmin} = T_{w0}$; if $T_{w1} < T_{w0}$, then $T_{wmax} = T_{w0}$. Then go to step.2 to make iteration, till $T_{w1} - T_{w0} < error$

The advantage of this method is that the iteration won't be oscilate and will always converge. Given a reasonable initial $T_{wmax}$ and $T_{wmin}$ (Which are ¡max_TW¿ and ¡min_TW¿ in arches.spec), the DIM will have very small cost.

**black body or grey body assumption?**

Up to now, the wall emissivity $\epsilon_w$ in Eq.7.67 is set to 1.0 in arches code, since that in DO model, the wall is assumed to be the black-body. If the grey body assumption has to be considered, we must make sure that radiation model (DO or RMCRT) also consider a greybody condition for the wall. Still, there is a trick to treat the wall as greybody in wall heat transfer model while keep adopting blackbody assumptions in Radiation model. The trick is as follows:

1. take an $\epsilon_w$ of less than one in Eq.7.67, and calculate the grey wall temperature $T_{w1}$ and $Q_{net}$ with the iteration method mentioned in Section7.2.5.
2. To modify the wall temperature from $T_{w1}$ to $T_w$, so that the following equations are satisfied: $q_{net} = q_{in} - \sigma T_w^4$. Generally the difference between $T_w$ and $T_{w1}$ is no more than 20K if $\epsilon_w = 0.8$. So the emissivity won't change the wall temperature a lot. However, it will be better if the wall reflection can be considered in the radiation model.

**numerical steadyness, especially for adiabatic wall BC or small thermal conductivities**

In Section.7.2.5, the calculation of $T_w$ is proposed for each times step. During LES simulation, $q_{in}$ will change with $T_w$ as well. The following conditions will cause the numerical simulation of $T_w$ unsteady

- A bad initial temperature assumption: a bad initial temperature assumption will casue a bad initial $q_{in}$, and thus a bad $T_w$, although $T_w$ is 'accurate' at this timestep, it's a fake wall temperature and may have negative effect on $q_{in}$ in next step.
- When simulation case is large and simulation domain is complex, the previous tests show that the simulation is sentitive to wall temperature. If $T_w$ changes frequently and changes in a large range, this may cause some numerical problems in Arches simulation.

with these consideration, it's better to change $T_w$ smoothly when $q_{in}$ is sensitive to $T_w$. So at the beginning of simulation, a small relaxation coefficient can be adopted to control the change of $T_w$. When simulation comes to a steady state, RIM and DIM moethod can be directly adopted to get the final $T_w$. However, the strategy of $T_w$ calculation depends on the real physical problems.

For adiabatic wall boundary conditions, since the thermal resisitance ($R_{wc}$) is infinitly large, numerical steady is a challenge when we use iterative method to calculate Eq.7.68, and DIM is a good way to solve this problem. In ARCHES, the DIM method is adopted and the adiabatic wall will be well solved. Still, a small relaxation coefficient, $C_{rex}$, is recommend to be adopted. For e.g., $C_{rex} = 0.05$, which will help to acquire a smooth change of wall temperature with time.

### 7.2.6 Digital Filter Generator for Turbulent Inlet Conditions

The digital filter method generates a set of both spatially and temporally correlated random data in order to create a synthetic turbulent inlet. In the Arches implementation, the data is generated a priori.

#### Digital Filter

Let $r_m$ be a set of random data such that $\rangle r_m \langle = 0$ and $\{r_m^2\{ = 1$. Then there exists a set of filter coefficients $b_n$ for the convolution such that

$$u_m = \sum_{-N}^{N} b_n r_{m+n} \tag{7.69}$$

where $N$ is the support of the filter. If 7.69 is rearranged in the form of an autocorrelation function then it follows that

$$\frac{u_m \bar{u}_{m+k}}{u_m \bar{u}_m} = \sum_{j=-N+k}^{N} b_j b_{j-k} \Big/ \sum_{j=-N}^{N} b_j^2 \tag{7.70}$$

For extending filter coefficients to three dimensions a simple convolution of the three one-dimensional filters is used.

$$b_{ijk} = b_i \cdot b_j \cdot b_k \tag{7.71}$$

Now consider a fully developed flow. A fit line for its autocorrelation function can be expressed as an exponential decay

$$R_{uu}(r) = \exp\left(-\frac{\pi r^2}{4L^2}\right) \tag{7.72}$$

where L is the integral length scale. If the grid spacing is $\Delta x$ then the length scale can be set as $L = n\Delta x$, and the discrete autocorrelation function is then

$$R_{uu}(k\Delta x) = \exp\left(-\frac{\pi(k\Delta x)^2}{4(n\Delta x)^2}\right) = \exp\left(-\frac{\pi k^2}{4n^2}\right) \tag{7.73}$$

Now apply 7.70 to relate the discrete autocorrelation function to the filter coefficients

$$R_{uu}(k\Delta x) = \frac{u_m \bar{u}_{m+k}}{u_m \bar{u}_m} = \sum_{j=-N+k}^{N} b_j b_{j-k} / \exp\left(-\frac{\pi k^2}{4n^2}\right) \tag{7.74}$$

There is an approximate solution to the filter coefficients available

$$b_k \approx \tilde{b}_k / \left(\sum_{-} N^N \tilde{b}_j^2\right)^{1/2} \quad \text{with} \quad \tilde{b}_k = \exp\left(-\frac{\pi k^2}{2n^2}\right) \tag{7.75}$$

which is a valid approximation as long the the filter support is sufficiently large, $N \geq 2n$. This maintains the the support of the filter is large enough to capture twice the length scale. This is a brief explanation of the method from Klein's paper [4]. Note that as $N \to 0$ this method approaches white noise.

**Implementation Algorithm**

The steps in generating the inlet data are listed here directly from Klein's procedure for isotropic flow [4].

1. Choose a length scale for each coordinate direction as$L_y = n_y \Delta y, L_z = n_z \Delta z$ and a timescale for the flow direction (or length scale by Talyor's hypothesis) $T_x = n_t \Delta t$ (or $L_x = n_x \Delta x$ ), set the size of the support $N_\alpha \geq 2n_\alpha$ for $\alpha = x, y, z$ . Also prescribe a mean flow profile.

2. Initialize three random fields $\mathcal{R}_\alpha$ of dimensions $[-N_x : N_x, -N_y + 1 : M_y + N_y, -N_z + 1 : M_z + N_z]$ , with $M_y \times M_z$ denoting the resolution of the grid of the inflow plane. The size is actually $[2N_x + 1, 2N_y + M_y, 2N_z + M_z]$.

3. Calculate the filter coefficients b(i,j,k), using 7.75

4. Loop over the inlet dimensions j and k and apply the filter operation for each velocity fluctuation
$$\mathcal{U}_\alpha(j,k) = \sum_{i'=-N_x}^{N_x} \sum_{j'=-N_y}^{N_y} \sum_{k'=-N_z}^{N_z} b(i',j',k') \mathcal{R}_\alpha(i', j+j', k+k') \tag{7.76}$$

5. Using the time-averaged Reynold's stress tensor ($R_{ij}$) solve for the transformation matrix $a_{ij}$
$$a_{ij} = \begin{bmatrix} (R_{11})^{1/2} & 0 & 0 \\ R_{21}/a_{11} & (R_{22} - a_{21}^2)^{1/2} & 0 \\ R_{31}/a_{11} & (R_{32} - a_{21}a_{31})/a_{22} & (R_{33} - a_{31}^2 - a_{32}^2)^{1/2} \end{bmatrix} \tag{7.77}$$

6. Now solve for each of the velocity components
$$u_i = \bar{u}_i + a_{ij} \mathcal{U}_j \tag{7.78}$$

7. Shift the random data set by the flow direction $\mathcal{R}_\alpha(i,j,k) = \mathcal{R}_\alpha(i+1,j,k)$ , and fill the last plane $\mathcal{R}_\alpha(N_x, j, k)$ with new random numbers.

8. Repeat steps 4 through 7 for each timestep

In Arches two major changes are made to this algorithm. First the data is all pre generated into a table that is loaded when the simulation starts. This happens because the summation in 7.76 can become very expensive at high resolution and occurs at each point in the boundary, and some issues with trying to deal with the random field at patch boundaries. The second is that the option to use Taylor's hypothesis of frozen turbulence is utilized so that the length scale is specified in the flow direction, rather than the time scale. Then the same flow profile is reused for several timesteps, this period is either specified as the number of steps or as a time.

## 7.3 Uintah Specification

### 7.3.1 Basic Inputs

Choosing the ARCHES component is done through the simulation controller using the *Simulation-Component* tag. In this case (similar to the other Uintah components) the Arches component is specified as

```
<SimulationComponent type="arches" />
```

as a child of the `<Uintah_Specification>` section.

Most other Arches specifications are located in the *CFD→ARCHES* section of the input file. Unless otherwise specified, the system of units for all Arches input parameters are SGI.

### 7.3.2 Boundary Conditions

Boundary conditions for Arches are specified using the Uintah boundary condition mechanism. As such, all boundary conditions for domain faces are specified in the `<Grid>`→`<BoundaryConditions>` section of the input file. Generic details of the UCF boundary condition mechanism may be found in 2.9.

Generally, boundary conditions are required for every transport equation except at periodic boundaries. Specifically, every equation specified in the `<TransportEqn>` section of the input file requires domain boundary specification on every domain face.

The generic options for the boundary condition type include `Dirichlet` and `Neumann` conditions. For the `Dirichlet` condition, transported variables at domain boundaries satisfy the condition,

$$\phi_b = \alpha_D \tag{7.79}$$

where $\phi_b$ is the transported variable at the domain extent and $\alpha_D$ is the value specified in the `<BCType>`→`<value>` tag. The `Neumann` condition specifies the gradient of $\phi$ at the boundary and thus satisfies the condition,

$$\frac{\partial \phi}{\partial n}|_b = \alpha_N \tag{7.80}$$

where $n$ is the normal direction of the boundary and $\alpha_N$ is the value specified in the `<BCType>`→`<value>` tag.

Momentum boundary conditions are specified using descriptive tags that in turn choose the appropriate Nuemann, Dirichlet, or some other boundary condition depending on the tag. Note that some momentum boundary conditions have implications for all transported variables despite any previously specified boundary condition for that specific scalar. This is described below.

Boundary conditions for momentum include (with the corresponding `<BCTyep var>` attribute listed),

- Specified velocity inlet (`VelocityInlet`)
- Specified mass flow rate (`MassFlowInlet`)
- Specified swirl inlet (`Swirl`)
- Velocity inlet read from file (`VelocityFileInput`)
- Digital Filter turbulence inlet from file (`TurbulentInlet`)
- Pressure boundary condition (`PressureBC`)
- Outlet boundary condition (`OutletBC`)
- Wall boundary condition (`WallBC`)

Details of each boundary condition follow.

### Velocity Inlet

The velocity inlet specification requires that `<vecvalue>` be specified as a child of `<BCType>`, where `<vecvalue>` represents a three component vector of velocity at the inlet. For example,

```
<Face circle="x-" radius="0.5" centroid="0 0.5 0.5">
        <BCType var="VelocityInlet" id="0" label="my velocity inlet">
                <vecvalue>[0.1, 0.2, 0.2]</vecvalue>
        </BCType>
</Face>
```

specifies a non-normal velocity component in a circular region on the x-minus face. Note that the `label` attribute is generic here, allowing the user to specify any descriptive title to the boundary condition.

Velocity inlet boundary conditions simply set the velocity vector at domain boundary interface. To ensure a constant mass flow rate for the given velocity condition, the actual velocity at the domain interface may vary to compensate for varying density within the domain while maintaining a constant mass flow rate into the domain. This step is necessary because density is a cell centered quantity and interpolation of density to the face may potentially change the mass flow rate if the velocity is not adjusted accordingly. This is accomplished by computing the following for the normal velocity component,

$$u_n = 2.0(\rho u)_b/(\rho_i + \rho_b), \tag{7.81}$$

where $n$ indicates the normal interfacial velocity, $b$ is the boundary density, and $i$ is the density from the first interior cell in the normal direction of the boundary. Note that a simple linear interpolant has been used to compute the boundary density.

### Mass Flow Inlet

The mass flow inlet boundary condition requires that `<value>` be specified as a child of `<BCType>`, where `<value>` is the mass flow rate in units of `[kg/s]`. For example,

```
<Face circle="x-" radius="0.5" centroid="0 0.5 0.5">
        <BCType var="MassFlowInlet" id="0" label="my mass flow inlet">
                <value>0.4</value>
        </BCType>
</Face>
```

specifies a mass flow rate in a circular region on the x-minus face. Note that the `label` attribute is generic here, allowing the user to specify any descriptive title to the boundary condition.

A velocity condition is set from the mass flow rate condition using the area and density of the boundary geometry. Note that the area of the geometry is computed from the true area of assembly of the grid cells and not the ideal geometry area. Once velocities are computed, the mass flow rate boundary condition proceeds as the previously described velocity inlet condition.

### Swirl Inlet

The swirl condition specifies a mass flow rate condition in addition to adding a swirling component from the non-normal velocity components. This specification requires that the `<value>` (mass flow rate, `[kg/hr]`), `<swirl_no>` (swirl number), and `<swirl_centroid>` be specified as a child of `<BCType>`. For example,

```
<Face circle="x-" radius="0.5" centroid="0 0.5 0.5">
        <BCType var="Swirl" id="0" label="my swirled inlet">
                <value>1.0</value>
                <swirl_no>16.0</swirl_no>
                <swirl_centroid>[0, 0.5, 0.5]</swirl_centroid>
        </BCType>
</Face>
```

specifies a mass flow inlet in a circular region on the x-minus face with a swirl number of 16 and a centroid coincident with the circle of the boundary geometry. Note that the swirl condition currently is applied across the entire geometry.

### Velocity File Input Inlet

Work in progress....please check back later.

### Turbulent Inlet

The turbulent inlet loads a pre-generated table consisting of cell indices for each realization time index provided. It requires a period, which specifies the amount of simulation time to use the same profile, a vector value to set up the iterator, and the input file to load. An example specification is

```
<Face circle="x-" radius="0.25" origin="0 0.5 0.5">
  <BCType id-"all" var="TurbulentInlet" label="my_turbulent_inlet">
    <value> [1.0, 0.0, 0.0] </value>
    <period> 10 </period>
    <inputfile> tubulent_input_file.gz </inputfile>
  </BCType>
</Face>
```

This loads the table and changes the time index of the realization to use once every 10 simulation timesteps. The period is an optional parameter that defaults to 1. If a timescale is used for generation the period should be one, but if a length scale is used for the flow direction, then the period needs to be such that sufficient time elapses to convect the velocity through one cell. A `<timeperiod>` specification can also be used, which is more convenient for a variable `dt` problem. The input file needs to have the number of total realizations generated and the non axial dimensions of the box around the inlet in the first line. The next line needs to be the minimum (i, j, k) cell index of the bounding box of the inlet, all of the indices of the velocity vectors stored are in relation to the corner being at (0, 0, 0), so this adjusts from the simulation's (i, j, k) to match. Then a list of the (t, j,k) points and (u, v, w) vectors, varying first by k, then j, then t (for a x- or x+ face). For other faces either (t, i, k) or (t, i, j) points are listed instead. It is recommended to use the `DigitalFilterGenerator` executable for creating a table.

#### Pressure Boundary Condition

A pressure (open) boundary condition is specified using `PressureBC` attribute. For example,

```
<Face side="y+">
        <BCType var="PressureBC" id="0" label="y+ pressure bc"/>
</Face>
```

specifies a pressure condition on the `y+` face of the domain.

#### Flow Outlet Boundary Condition

A flow outlet boundary condition is specified using `OutletBC` attribute. For example,

```
<Face side="y+">
        <BCType var="OutletBC" id="0" label="x+ outlet bc"/>
</Face>
```

specifies an outlet condition on the `x+` face of the domain.

#### Wall Boundary Condition

A wall boundary condition is specified using `WallBC` attribute. For example,

```
<Face side="y+">
        <BCType var="WallBC" id="0" label="x- wall bc"/>
</Face>
```

specifies a stationary wall condition on the `x-` face of the domain.

### 7.3.3 Time Integrator

#### Explicit Time Integrator

Arches is commonly run in a fully explicit time-stepping mode. That is, the update in time for any variable $\phi$ is expressed as

$$\phi^{t+\Delta t} = \frac{1}{\rho^*}\left((\rho\phi)^t + \Delta t RHS^t\right) \ , \tag{7.82}$$

where $RHS$ represents all forcing terms in the transport equation for $\phi$ at time level $t$. For the purposes of this discussion, we have dealt with the implicit nature of the density term by simply

assuming we have a density approximation, called $\rho^*$, that suits the current update (for details of the $\rho$ issue, see Section 7.2.3).

The explicit time integrator is activated (as a child node of <CFD>→<ARCHES>) by simply inserting the <ExplicitIntegrator> node. Within this node, the other solvers for the various transport equations will be defined along with a few parameters. The general structure will look something like this:

```
<ExplicitSolver>
           <!--Solver Options-->
           <option-1/>
           <option-2/>
           ....
           <!--Transport Equations-->
        <MomentumSolver>
        ...
        </MomentumSolver>

        <PressureSolver>
        ...
        </PressureSolver>

        <MixtureFractionSolver>
        ...
        </MixtureFractionSolver>

        <EnthalpySolver>
        ...
        </EnthalpySolver>
</ExplicitSolver>
```

The options for each transport equation will be described in Section 7.3.5.

Options for the <ExplicitSolver> section include:

1. **Initial time step**: <initial_dt>
   **Input type**: *Required, double*
   **Default**: *NA*
   **Description**: The explicit solver can be stepped forward in time by using a fixed time step or letting the code estimate a time step via a CFL condition (see Section 7.2.3). In either case, an initial time step must be specified.

2. **Variable time step**: <variable_dt>
   **Input type**: *Required, boolean*
   **Default**: *NA*
   **Description**: One may either step at a fixed time step with $\Delta t$ equal to the *intial_dt* tag or let the code guess a stable time step according to a CFL condition. It is recommended that one sets the *variable_dt* to *true* as it helps maintain stability during the time integration.

3. **Time integration order**: <timeIntegratorType>
   **Input type**: *Required, string*
   **Default**: *FE*
   **Description**: Current options include one of the following:

   - FE, 1st order Forward-Euler

- RK2SSP, Second Order, Strong-Stability Preserving Runge-Kutta
- RK3SSP, Third Order, Strong-Stability Preserving Runge-Kutta

See Section 7.2.3 for full details.

4. **Stability option for the density guess**: <restartOnNegativeDensityGuess>
   **Input type**: *Optional, boolean*
   **Default**: *false*
   **Description**: This parameter restarts a time step, regardless of the time integrator order, if the predicted density guess (see Section 7.2.3) from the continuity equation is negative and therefore unphysical. If this option is true, the time step is reduced by half and the time step is restarted with the new, smaller time step. The process with repeat until a) the density guess is physical or b) the code goes unstable. Instability usually will occur in the implicit pressure projection. If b) occurs, it is advised to set this option to *false*. By default this option is *false* and is not required. Note that in cases where this parameter is *false* and a negative density guess occurs, the density from the previous time step is used.

5. **Message control on density guess**: <NoisyDensityGuess>
   **Input type**: *Optional, boolean*
   **Default**: *true*
   **Description** : The negative density guess warning prints for every cell with a negative density guess. One may want to suppress the warning and can do so with this option. When used, a warning is printed for every patch rather than every cell.

6. **Turbulence model calculation frequency**: <turbModelCalcFreq>
   **Input type**: *Optional, integer*
   **Default**: *1*
   **Description**: This parameter allows one to control the frequency of the execution of the turbulence model. One may want to decrease the frequency for efficiency reasons.

7. **Turbulence model calculation frequency on time integrator sub-steps**: <turbModelCalcForAllRKSteps>
   **Input type**: *Optional, boolean*
   **Default**: *true*
   **Description**: If *false*, the turbulence closure will only be computed for the first time sub-step and then applied for all subsequent time sub-steps. By default, this parameter is *true*.

8. **Additional time step constraint**: <scalarUnderflowCheck>
   **Input type**: *Optional, boolean*
   **Default**: *false*
   **Description**: Guaranteeing stability for a problem with large length and time scales is difficult. As previously mentioned, a guess at a stable time step is made using a CFL condition. The scalar underflow check option uses additional information about the local flow information to compute an additional time step guess. The minimum of this estimation and the CFL condition is used to step the equations forward in time. Here, a time step is computed from the inverse of the continuity equation by considering outward mass fluxes only. In other words, given the local velocity state, there is a limit to the amount of mass that can leave any given cell. This limit is computed from

$$\frac{\partial \rho}{\partial t} = -\nabla \cdot (\rho U)|_{+}$$

when only outward facing fluxes are considered (as indicated by the $|_+$ symbol). Thus one can rearrange this equation to give an estimate for $\Delta t$ as,

$$\Delta t \approx \partial t = \frac{\partial \rho}{\nabla \cdot (\rho U)|_+}$$

This option is often helpful in helping with stability if your simulation is experiencing underflow ($< 1.0$) or overflow ($> 1.0$) errors from the mixture fraction scalar.

9. **Extra pressure projection option**: <extraProjection>
   **Input type**: *Optional, boolean*
   **Default**: *false*
   **Description**: This option performs a second pressure solve and projection step. In general, this option is not needed. By default the value is *false*.

#### Implicit Time Integrator

Currently the implicit time integrator is not supported.

### 7.3.4 Scalar Transport Equations

A set of user defined scalars can be defined through the input file. The scalar can be passive (it doesn't feed back to the CFD) or active (plays a role in the CFD solution). An example of an active scalar would be a mixture fraction variable that is used for the chemistry table look-up.

The basic form of the scalar transport is

$$\frac{\partial \rho \phi}{\partial t} + F_{conv} = F_{diff} + \sum_{i=1}^{N} S_i, \tag{7.83}$$

where $\phi$ is the scalar, $F_{conv}$ is the total convection term, $F_{diff}$ is the total diffusion term, and $S_i$ is the set of $N$ source terms associated with this scalar. If $N = 0$, then the scalar is a conserved scalar.

Scalar equations are defined in the <TransportEqn> node of the input file. The generic specification looks like

```
<TransportEqns>
  <Eqn label="my_great_scalar" type="CCscalar">
          ...equation options...
  </Eqn>
  <Eqn label="another_great_scalar" type="CCscalar">
          ...equation options...
  </Eqn>
</TransportEqns>
```

In the above example, two scalar equations were defined with a set of solver options. The *label* specification is used as the equation identifier, and must be unique for every equation. Currently, only one type of scalar is supported (*CCscalar*). Beside the actual definition of the scalar and its respective options, the only other requirement for each scalar is that boundary conditions are defined for all boundary condition specification (see Section 7.3.2).

#### Options for Scalar Transport Equations

Options for an individual scalar equation (<Eqn>) includes:

1. **Turbulent Schmidt/Prandtl Number**: <turbulentPrandtlNumber>
   **Input type**: *Optional, double*
   **Default**: 0.4
   **Description**: The total diffusion coefficient is computed as

$$D_T = \rho D_{molecular} + \frac{\mu_t}{Sc_t}$$

   where $D_{molecular}$ is the molecular diffusion coefficient, $mu_t$ is the turbulent viscosity (from the turbulence model) and $Sc_t$ is the turbulent Schmidt number.

2. **Constant Molecular Diffusion Coefficient**: <D_mol_constant value="DOUBLE">
   **Input type**: *Optional, double*
   **Default**: *NA*
   **Description**: Sets a constant molecular diffusivity for the scalar. The *value* attribute specifies the constant. Note that if *D_mol_constant* or *D_mol* is not specified, then the molecular diffusion is assumed zero.

3. **Variable Molecular Diffusion Coefficient**: <D_mol label="STRING">
   **Input type**: *Optional, string*
   **Default**: *NA*
   **Description**: Points the equation to the correct label for the molecular diffusion coefficient. The label must be defined elsewhere in a property model, table, etc. If the molecular diffusivity is a table lookup variable, the label must be saved in the uda (as specified in the <DataArchiver> section) for the variable to exist (a bit hackish but should suffice for now.) Note that if *D_mol_constant* or *D_mol* is not specified, then the molecular diffusion is assumed zero.

### 7.3.5 Transport Equation Options

In the current configuration of Arches, transport equations are activated by specifying the option for each equation in a equation-specific node under the *ExplicitSolver* node. Currently, all equation nodes are required except for the enthalpy solver node.

#### Momentum Solver

The moment solver refers to solution of $\rho\mathbf{U}$, where $\mathbf{U}$ is the vector quantity of velocity. As mentioned above, the components are solved in a staggered, finite volume configuration. By default, the required <MomentumSolver> node must be present in the <ExplicitSolver> node.

The options for the moment solver include:

1. **The order of the convection scheme**: <convection_scheme>
   **Input type**: *Required, string*
   **Default**: *NA*
   **Description**: The two options that currently are implemented for the convection term in the moment equation are first-order upwind (set as *upwind*) and second-order central difference

(set as *central*). Both types of discretization can be found in any common CFD text. It is recommended that one use the *central* option as it has desirable energy conservation properties (see Moronishi [add reference] ).

2. **Filter the divergence of $\rho$U**: <filter_divergence_constraint>
   **Input type**: *Optional, boolean*
   **Default**: *false*
   **Description**: This options turns on the filtering of the divergence constraint used in the pressure solver. When false, the divergence is unfiltered.

**Pressure Solver**

Arches is solved in an incompressible manner, in the sense that there is a degree of pressure-velocity decoupling which is resolved through an implicit pressure projection. This results in the classic Poisson equation for pressure than requires solution. By default, the required <PressureSolver> node must be present in the <ExplicitSolver> node.

The options for the pressure solver include:

1. **Perform only the last projection**: <do_only_last_projection>
   **Input type**: *Optional, boolean*
   **Default**: *false*
   **Description**: For multi-step time schemes, only perform the projection on the last time sub-step. The result is that intermediate time steps do not conserve mass.

2. **Normalize the pressure with the reference pressure**: <normalize_pressure>
   **Input type**: *Optional, boolean*
   **Default**: *false*
   **Description**: When true, this option subtracts the reference pressure, set in <PhysicalProperties>, from the current value of pressure for each time step.

3. **Solver choice for the pressure Poisson equation**: <linear_solver>
   **Input type**: *Required, string*
   **Default**: *NA*
   **Description**: Arches uses external linear solver packages to solve the pressure Poisson equation. Currently, there are two solver that have an interface to the pressure equation; *hypre* or *petsc*. A solver must be specified and specifics of the solver follow in the <parameter> section (detailed next).

4. **Solver parameters for the pressure Poisson equation**: <parameters>
   **Input type**: *Required, NA*
   **Default**: *NA*
   **Description**: The solver parameters, as children of the *parameters* node, include the following:

   - <solver>, Required solver parameter. Options include: *cg*.
   - <max_iter>, Required maximum iterations for the solver.
   - <preconditioner>, Required preconditioner. Options include: *jacobi*, *pfmg*.
   - <res_tol>, Required tolerance of the residual ($res = b - Ax$).

### Mixture Fraction Solver

In Arches current configuration, specific scalar variables that require specification. The mixture fraction equation is one equation that requires definition in order to run any simulation. This is true independent of the relation of the local density with the mixture fraction value.

In general, the mixture fraction equation is a conserved scalar equation that is used as a parameter to map the thermo-chemical state of the gas. In its most simple form, the mixture fraction and density are related by a linear relationship (such as mixing of two non-reacting gasses with different densities). More complicated state-space relationships are also available, which include subgrid reaction and turbulence mixing.

By default, the required <MixtureFractionSolver> node must be present in the <ExplicitSolver> node. Parameters within the mixture fraction node include;

1. **Initial value of of the mixture fraction in the domian**: <initial_value>
   **Input type**: *Optional, double*
   **Default**: *0.0*
   **Description**: One may set the mixture fraction everywhere inside the domain to a constant value. Boundary condition values are set elsewhere.

2. **Convection scheme**: <convection_scheme>
   **Input type**: *Required, string*
   **Default**: *central-upwind, flux-limited*
   **Description**: The choice of the convection scheme can affect the stability of the algorithm. The *central-upwind* option is the second upwind differencing scheme defined by Roache (ADD REFERENCE). The *central-upwind* option is kept separate from the *flux_limiter* option for historical reasons. One may choose the type of limter for the *flux_limited* specification by setting the <limeter_type> option. The currently available limiters include:

   - *superbee*
   - *vanLeer*
   - *upwind* - This option is the standard upwind scheme and is very stable, yet of low-order.
   - *none* - This option uses central differencing and will add noise, and possibly cause instabilities, if used.

   The super-bee scheme is used if the limiter type is not specified.

   One may also control the limiter type near boundary conditions by setting the <boundary_limiter_type>. Options include:

   - *central-upwind* The second upwind differencing scheme of Roache
   - *upwind* First order upwind scheme

   These options are necessary due to the wide stencil of some limiter types.

### Enthalpy Solver

Arches solves a filtered enthalpy transport equation for tracking energy in the system. Modes of heat transfer include convection, diffusion and radiative heat transfer. The energy equation is

activated by specifying the <EnthalpySolver> section within the <ExplicitSolver> node. Note that by neglecting this node, isothermal flow is assumed.

Current options for the enthalpy solver include:

1. **Convection scheme**: <convection_scheme>
   **Input type**: *Required, string*
   **Default**: *central-upwind, flux-limited*
   **Description**: The choice of the convection scheme can affect the stability of the algorithm. The *central-upwind* option is the second upwind differencing scheme defined by Roache (ADD REFERENCE). The *central-upwind* option is kept separate from the *flux_limiter* option for historical reasons. One may choose the type of limter for the *flux_limited* specification by setting the <limeter_type> option. The currently available limiters include:

   - *superbee*
   - *vanLeer*
   - *upwind* - This option is the standard upwind scheme and is very stable, yet of low-order.
   - *none* - This option uses central differencing and will add noise, and possibly cause instabilities, if used.

   The super-bee scheme is used if the limiter type is not specified.

   One may also control the limiter type near boundary conditions by setting the <boundary_limiter_type>. Options include:

   - *central-upwind* The second upwind differencing scheme of Roache
   - *upwind* First order upwind scheme

   These options are necessary due to the wide stencil of some limiter types.

2. Discrete Ordinates Radiation Model: <DORadiationModel>
   **Input type**: *NA*
   **Default**: *NA*
   **Description**: The discrete ordinates method is based on the numerical solution of the radiation transport equation (RTE) along specified directions. The total solid angle about a location is divided into a number of ordinate directions, each assumed to have uniform intensity. Each transport equation that is solved corresponds to an ordinate direction selected from an angular quadrature set that discretizes the unit sphere and describes the variation of directional intensity throughout the domain.

   If the <DORadiationModel> section is found, the model is activated and the radiative source is automatically added to the transport equation. If the section is absent, the calculation is assumed to have no radiative energy transport. Options for the discrete radiation model include

   - Optical path length: <opl>
     **Input type**: *Required, double*
     **Default**: *NA*
     **Description**: The optical path length for the radiation model.

- Number of ordinate directions: <ordinates>
  **Input type**: *Optional, integer*
  **Default**: *2*
  **Description**: The discrete ordinates method uses quadrature methods to represent the divergence of the radiative heat flux. The quadrature order is defined by the number of ordinate directions, $n$. The number of equations to be solved depends directly on $n$. The DO scheme is often referred to as an $Sn$ scheme where $n$ is the number of ordinate directions.
- Property Model: <property_model>
  **Input type**: *Optional, double*
  **Default**: *radcoef*
  **Description**: This option defines the model for computing the radiation properties. Options include:
  - *radcoef*
  - *patchmean*
  - *wsggm*
- Linear Solver: <linear_solver>
  **Input type**: *Required, string*
  **Default**: *NA*
  **Description**: This options sets the linear solver for the radiation calculation.

### 7.3.6 Initial and Boundary Conditions

#### Solid Intrusions (MPMArches only)

One may specify solid intrusions that intersect the physical domain. Intrusions will be treated with a stationary wall boundary condition.

The options for the `<Arches>`→`<BoundaryConditions>`→`<intrusions>` section are as follows:

1. Geometry object: `<geom_object>`
   **Input Type**: *XML node*
   **Default**: NA
   **Description**: All geometry objects are specified within this single `<geom_object>` tag.

#### Species Efficiency Calculations

The scalar efficiency calculator computes an overall balance on an atomic or molecular species. The output is a data file placed within the UDA with the prescribed label. The format of the data file consists of a column of time values and a column of efficiency values. To save the output to the UDA, one must specify the

```
<save label="user_defined_efficiency"/>
```

in the data archiver section of the input file. Note that the user_defined_efficiency name is the unique name given to the scalar efficiency (see below).

The total scalar species is computed as

$$\eta_{eff} = \frac{\dot{m}_{out} - \dot{m}_{in}}{\dot{m}^*_{in}} \tag{7.84}$$

where the superscript $*$ in the denominator signifies that the inflow is only summed over inlet boundary conditions. The value of $\dot{m}_{in}$ in the numerator is summed over all boundary condition types. The outflow, $\dot{m}_{out}$ is only summed over pressure and outlet type boundary conditions.

The generic setup of the `<ScalarEfficiency>` node should appear as

```
<ScalarEfficiency>
  <scalar label="my_carbon_efficiency" fuel_ratio="0.78" air_ratio="0.0">
    <species label="CO2" mol_ratio="0.27272727"/>
    <inlet>fuel_inlet</inlet>
  </scalar>
</ScalarEfficiency>
```

The `<ScalarEfficiency>` node is a child of the `<BoundaryCondition>` node. Parameters within this node include;

1. Scalar: `<scalar>`
   **Input Type**: *XML node*
   **Default**: NA
   **Description**: All information about a single atomic or molecular efficiency is encapsulated within this node. The following attributes of `<scalar>` are required

   - *label*: The user-given name of this species balance. The efficiency output data file will assume this name.
   - *fuel_ratio*: The mass ratio of species to total species mass in the fuel inlet (kg species/total fuel kg)
   - *ox_ratio*: The mass ratio of species to total species mass in the air (oxidizer) inlet (kg species/total oxidizer hg)

2. Species: `<scalar>`→`<species>`
   **Input Type**: *XML node*
   **Default**: NA
   **Description**: This node specifies the specific species that will be used to compute the balance. Note that multiple `<species>` may be specified in this section if needed. The following attributes are required for this node:

   - *label*: The label of the species.
   - *mol_ratio*: The molecular weight ratio of the species to the molecule. For example, in the case where we are computing a balance on the carbon atom and the output species is $CO_2$, we would enter 0.2727 because ratio = 12.0/44.0.

3. Inlet: `<scalar>`→`<inlet>`
   **Input Type**: *string*
   **Default**: *none*
   **Description**: This identifies which geometric objects of which inlets (ie, the label attribute of the geometry object) are associated with this scalar balance. One may have multiple `<inlet>`'s specified.

### 7.3.7 Properties, Reaction and Sub-Grid Mixing

Typically, subgrid mixing and reaction processes of the gas phase are described using pre-computed and tabulated mixing and reaction chemistry tables. These tables consists of the thermo-chemical state-space (density, temperature, species) as a function of a few independent parameters (e.g., mixture fraction, heat loss, scalar variance). In the `<Properties>` child of the `<ARCHES>` node, the inputs for the mixing and reaction models are specified, including additional information affecting various gas property models (e.g., empirical soot model).

To use tabulated chemistry, one must specify

1. Transport equations for any independent mixture fraction or other grid resolved species equation that parameterizes the table
2. Boundary conditions for every independent variable, including heat loss, scalar variance, and mixture fraction/species equations
3. A table specific node in the `<Properties>` section of the input file defining which table mechanism is being used

Information on specifying transport equations and boundary conditions is found in Section (ADD SECTION).

ARCHES currently has three tabulated property formats that can be interpreted:

1. Classic Mixing Table
2. New Static Mixing Table
3. TabProps

One way conversion from New Static Mixing Table to Classic Mixing Table to TabProps format is possible. To convert from a New Static Mixing Table to a Classic Table, use the `newstatic_to_classic.m` matlab script found in the `matlab` directory of the ARCHES input directory. Classic formatted tables can be pre-processed by the standalone TabProps library to produce a format readable by TabProps during runtime. Information on TabProps is found at
`https://software.crsim.utah.edu/trac/wiki/TabProps`.

Tables formatted in the Classic form are activated by using the following

```
<ClassicTable>
  <inputfile>REQUIRED STRING<inputfile/>
  <cold_flow>OPTIONAL BOOLEAN</cold_flow>
  <noisy_hl_warning>OPTIONAL BOOLEAN</noisy_hl_warning>
  <hl_scalar_init>OPTIONAL DOUBLE</hl_scalar_init>
  <coal  fp_label="REQUIRED STRING" eta_label="REQUIRED STRING"/>
</ClassicTable>
```

The `<cold_flow>` tag is used when the table does not involve a typical combustion process, as in mixing of two non-reacting streams or acid-base chemistry. The `<noisy_hl_warning>` option provides verbose output during runtime if computed heat losses are exceeding the table bounds. The `<hl_scalar_init>` option initializes the domain at the first time step with the specified value for heat loss.

If one is using a coal table which requires a transformation of mixture fractions, the <coal> tag with its attributes defines how the third mixture fraction is computed from the following relationship,

$$f = \frac{f_p}{(1 - \eta_c)}. \tag{7.85}$$

This identifies the mapping for the transformation of the transported $f_p$ (primary mixture fraction) and $eta_c$ (coal gas mixture fraction) equations that must be defined in the <TransportEqn> section of the input file.

Tables formatted in the TabProps form are activated by using the following

```
<TabProps>
  <inputfile>REQUIRED STRING</inputfile>
  <cold_flow>OPTIONAL BOOLEAN</cold_flow>
  <hl_scalar_init>OPTIONAL DOUBLE</hl_scalar_init>
  <noisy_hl_warning>OPTIONAL NO_DATA<noisy_hl_warning>
  <lower_hl_bound>OPTIONAL DOUBLE</lower_hl_bound>
  <upper_hl_bound>OPTIONAL DOUBLE</lower_hl_bound>
  <coal  fp_label="REQUIRED STRING" eta_label="REQUIRED STRING"/>
</TabProps>
```

where the same description of the tags for the Classic Tables applies to the TabProps tables. Note that the TabProps tables do not automatically determine the bounds for the heat loss grid, thus one must explicitly set the lower and upper bounds for heat loss if they differ from $(-1, +1)$ for strict error checking.

For both the Classic and TabProps tables, any dependent variable in the table is saved by specifying the following in the <DataArchiver> section of the input file:

```
<DataArchiver>
    <save name=STRING table_lookup="true">
</DataArchiver>
```

where the additional table_lookup attribute is specified to signify to ARCHES that the variable is found in the table. Note that the string name must match exactly the name in table.

In addition to the pre-tabulated chemistry, cold-flow mixing is available for simple two-stream mixing. The cold two-stream mixing is activated by

```
<ColdFlow>
  <mixture_fraction_label>REQUIRED STRING</mixture_fraction_label>
  <Stream_1>
    <density>REQUIRED DOUBLE POSITIVE</density>
    <temperature>REQUIRED DOUBLE POSITIVE</temperature>
  </Stream_1>
  <Stream_2>
    <density>REQUIRED DOUBLE POSITIVE</density>
    <temperature>REQUIRED DOUBLE POSITIVE</temperature>
  </Stream_2>
</ColdFlow>
```

where <mixture_fraction_label> is the name given to the mixture fraction equation defined in the <TransportEqns> section. A mixture fraction of, $f = 1$ represents a mixture of pure Stream 1 and conversely a mixture fraction of, $f = 0$, is pure Stream 2.

### Classic Table Format

The Classic Mixing Tables can function with an arbitrary number of independent variables. The parser for the tables treats # as a commented line, so any header information in the table should be precluded by this. The header information can also include various table constants with `#KEY VARNAME=VALUE`. After the header the sizes of the table and its variables must be specified in a specific order. This is:

1. Number of independent variables
2. Names of independent variables
3. Grid Sizes of independent variables
4. Number of dependent variables
5. Names of dependent variables
6. Units of dependent variables

This is followed by a list of grid points for each independent variable, listed *backwards* from N through independent variable 2. The table format allows for the grid points used for independent variable 1 to vary as a function of the value of last independent variable. The grid point list for independent variable 1 is listed at the start of each block of data where the $N^{th}$ variable varies. The blocks of data for the dependent variable are listed similarly to how Matlab displays a full N-D matrix. Independent variable 1 varies across the row, and independent variable 2 varies down the columns. The next block of data should then change the value of the third independent variable, and vary independent variables 1 and 2 as before. This is repeated up to the $(N-1)^{th}$ independent variable. Then the list of the grid points for the first independent variable for the next value of the $N^{th}$ independent variable is listed , then the other variables are listed as before, and repeated until the last value of the $N^{th}$ independent variable is used. This is repeated for all of the dependent variables.

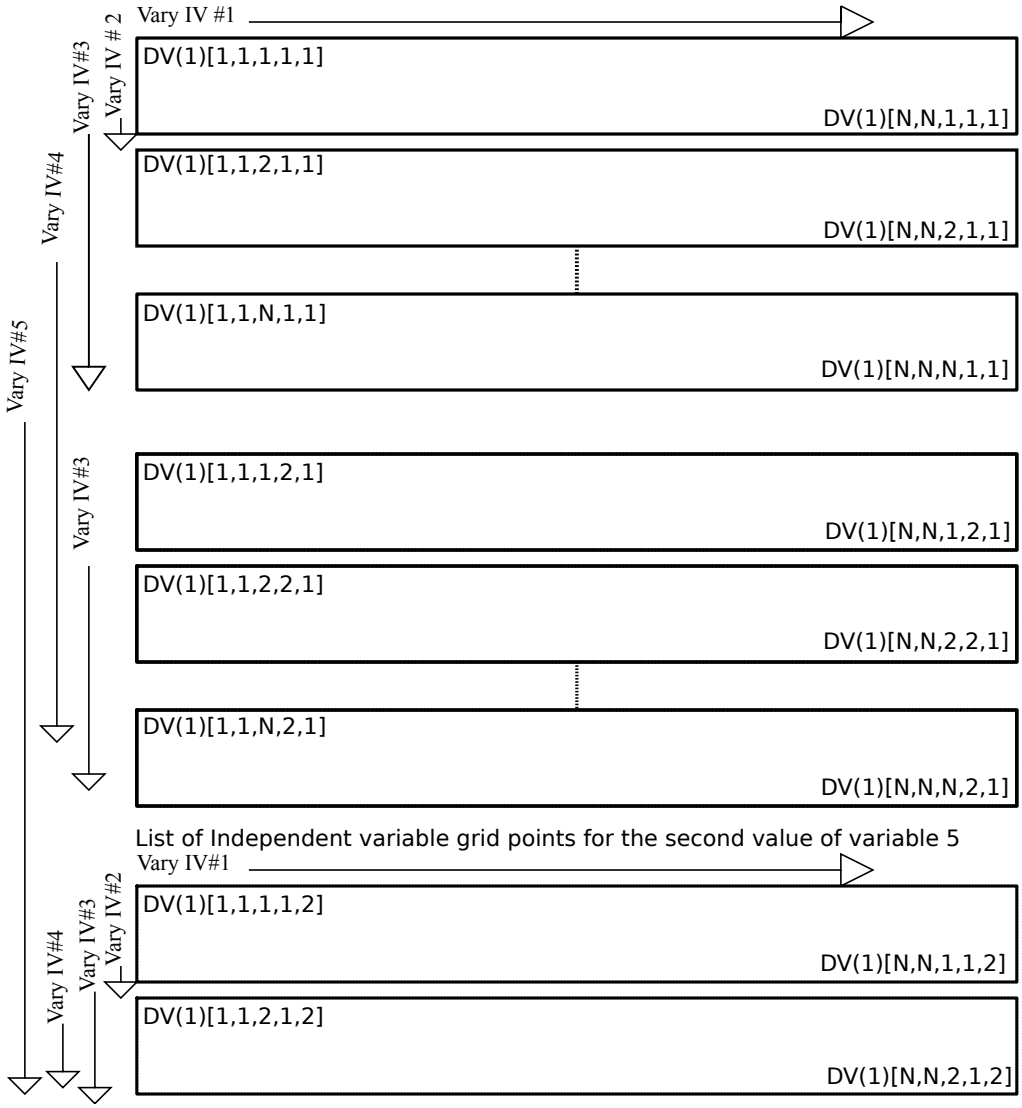
Refer to `src/StandAlone/inputs/ARCHES/ClassicMixingTables/` for example tables. A diagram of a 5 dimensional table layout (i.e. for a flamelet model) is also included in figure 7.2.

#Header
#Header
#Header
#KEY Table Constant(s) (i.e. H_ox)

Number of indepedent variables
Names of independent variables
Grid Sizes
Number of dependent variables
Names of dependent variables
Units of dependent variables

List of independent variable 5 grid points
List of independent variable 4 grid points
List of independent variable 3 grid points
List of independent variable 2 grid points

List of independent variable 1 grid points for first value of variable 5




List of Independent variable grid points for the second value of variable 5




Figure 7.2: Example 5D Table Layout

### 7.3.8 Direct Quadrature Method of Moments

The direct quadrature method of moments (DQMOM) is implemented in Arches, and various parameters, optional and required, are set in the input file. DQMOM involves transporting several transport equations, namely transport equations for the weights $w_\alpha$ and abscissas $\langle \xi_j \rangle_\alpha$, or, alternatively, the weights $w_\alpha$ and weighted abscissas $\varsigma_{j,\alpha}$. Information about these transport equations must be specified in the DQMOM tags.

Three other things need to be specified, namely: a set of moments with which to generate a set of quadrature-approximated moment transport equations, which make up the linear system $\mathbf{Ax} = \mathbf{B}$; parameters related to the solver for the linear system $\mathbf{Ax} = \mathbf{B}$; and finally, physical models implemented to describe the evolution of the NDF in state-space. The input file has a DQMOM section denoted by the tag `<DQMOM>`.

The basic structure is as follows:

```
<DQMOM>
        <LinearSolver>
                <!-- Linear solver options -->
                ...
        </LinearSolver>

        <Models>
                <!-- the coal models used in DQMOM are specified here -->
                ...
        </Models>

        <VelModel>
                <!-- information about the particle velocity model
                        is specified here -->
                ...
        </VelModel>

        <Weights>
                <!-- weight transport equation information set here -->
                ...
        </Weights>

        <Ic label="...">
                <!-- weighted abscissa (for each internal coordinate)
                        transport equation information set here -->
                ...
        </Ic>

        <Moment><m>[...]</m></Moment>
        ...
        <Moment><m>[...]</m></Moment>
</DQMOM>
```

Three additional tags go between the `<DQMOM>` tags. These are:

1. **Number of quadrature nodes**: `<number_quad_nodes>`
   **Input type**: *Required, positive integer*
   **Description**: Denotes the number of quadrature nodes (also called environments) with which to represent the NDF. The NDF is represented by a set of delta function, and each quadrature node represents an additional delta function.

2. **Adiabatic Gas, Non-Adiabatic Particles**: `<adiabGas_nonadiabPart>`
   **Input type**: *Optional, boolean*
   **Default**: *false*
   **Description**: This parameter is used to indicate that the gas is adiabatic (in which case, it is *true*) and therefore heat loss should not be looked up from the table. The default is to assume *false*, so that the heat loss will be used as a variable if it is so determined by the remainder of the input file.

3. **Save moments**: `<save_moments>`
   **Input type**: *Optional, boolean*
   **Default**: *false*
   **Description**: This boolean determines whether or not the moments specified in the `<Moments>` tags (see below) should be calculated in order to be saved out as variables. If true, all moments specified in the `<Moments>` tags will be calculated. In order to actually save them, however, they must also be added to the `<save>` labels.

**Linear Solver, <LinearSolver>**

1. **Tolerance**: `<tolerance>`
   **Input type**: *Optional, positive double*
   **Default**: 1.0e-5
   **Description**: Sets the linear solver tolerance for DQMOM. For the linear system

$$\mathbf{A}\mathbf{x} = \mathbf{B}, \tag{7.86}$$

   the residual vector $\mathbf{R}$ is defined as

$$\mathbf{R} = \mathbf{A}\mathbf{x} - \mathbf{B} \tag{7.87}$$

   and the normalized residual vector $\mathbf{R}^\star$ is defined as

$$\mathbf{R}^\star = \frac{\mathbf{A}\mathbf{x} - \mathbf{B}}{\mathbf{x}}. \tag{7.88}$$

   The tolerance set by this tag will enforce the following condition

$$\begin{cases} \text{if} \|\mathbf{R}^\star\| \leq \text{tol} & \text{Use solution obtained from } \mathbf{x} = \mathbf{A}^{-1}\mathbf{B} \\ \text{if} \|\mathbf{R}^\star\| > \text{tol} & \text{Throw out solution from } \mathbf{x} = \mathbf{A}^{-1}\mathbf{B} \text{ and use } \mathbf{x} = \mathbf{0} \end{cases} \tag{7.89}$$

2. **Type**: `<type>`
   **Input type**: *Required, string*
   **Default**: *"LU"*
   **Description**: The linear solver type; options are:

   - "Lapack-invert" - uses a Lapack routine to invert $\mathbf{A}$, which is then multiplied by $\mathbf{B}$
   - "Lapack-svd" - uses a Lapack routine to calculate the singular value decomposition (SVD) of $\mathbf{A}$
   - "LU" - LU decomposition solver using the Crout's Method algorithm
   - "Optimize" - uses sets of optimal moments and optimal abscissas to get a well conditioned matrix

3. **Maximum allowable condition number for A**: `<maxConditionNumber>`
   **Input type**: *Optional, positive double*
   **Default**: *1.0e+16*
   **Description** : If the condition number of **A** is greater than the maximum allowable condition number, the resulting solution vector is discarded and $\mathbf{x} = \mathbf{0}$ is used; this prevents large numerical errors from contaminating the solution.

4. **Calculate condition number? (boolean)**: `<calcConditionNumber>`
   **Input type**: *Optional, boolean*
   **Default**: *false*
   **Description** : If *true*, the condition number is calculated using singular value decomposition. Note, this calculation is very expensive and should probably only be done when the "Lapack-svd" solver option is being used. An error is thrown if this is true and the "LU" (Crout's Method) solver option is chosen.

5. **Using the "Optimize" solver**: `<Optimization>`
   **Description** : This section must be specified to use the "Optimize" solver. The section must contain specification of the optimal abscissas:
   `<Optimal_abscissas>`
   **Input type**: *Required, Vector*
   **Default**: *none*
   **Description** : The optimal abscissas should be specified as a vector (within brackets) and separated by comas. The set of moments chosen by the user should be consistent with the set of otpimal absissas. The number of optimal abscissas to specify is Nic*Nqn. The user can use the matlab script located in $/src/StandAlone/inputs/ARCHES/matlab/find_optmoments.m$ to determine a set of optimal moments and absissas given a number of internal coordinates and quadrature nodes. After running the script, the set of optimal abscissas is stored in Xgood and the corresponding set of optimal moments is stored in Momentsgood. Note, the order of the abscissas and the moments matters and should not be changed. Also, the following options are irrelevent for the "Optimize" solver: tolerance, maxConditionNumber, calcConditionNumber.

### Coal Models for DQMOM, <Models>

See the 7.3.9 section below for details about the `<Models>` tag.

### Velocity Model

This section sets parameters for the Balachandar velocity model.

1. **Velocity model**: `<VelModel>`
   **Attributes**: label, type
   **Description**: Attributes determine the label the velocity model is given, and the type of velocity model being used. The label is an arbitrary string. The type takes on these values:

   - `Balachandar` - Use the Balachandar particle velocity model
   - `Dragforce` - Use a drag force law, and track particle velocity as an internal coordinate

2. **Viscosity**: `<kinematic_viscosity>`
   **Input type**: *Required, positive double*
   **Default**: *1.0e-5*
   **Description**: The kinematic viscosity.

3. **Length scale**: `<L>`
   **Input type**: *Required, positive double*
   **Default**: *1.0*
   **Description**: The integral scale of the simulation.

4. **Density ratio**: `<rho_ratio>`
   **Input type**: *Required, positive double*
   **Default**: *1000*
   **Description**: The ratio of particle density to fluid density, $\frac{\rho_{\text{particle}}}{\rho_{\text{fluid}}}$.

5. **Regime**: `<regime>`
   **Input type**: *Required, integer (1 or 3)*
   **Default**: *1*
   **Description**: The flow regime for the Balachandar particle velocity model.

6. **Eta**: `<eta>`
   **Input type**: *Required, positive double*
   **Default**: *1.0e-5*
   **Description**: The Kolmogorov scale.

7. **High clip value (via upper limit multiplier)**: `<upper_limit_multiplier>`
   **Input type**: *Optional, double*
   **Default**: *2.0*
   **Description**: This factor is not actually used. It is used to set a high clip value. The way it stands in the code, this is, for some reason, multiplying the upper limit multiplier and the high clip value for the particle length - which doesn't make sense.

8. **Low clip value**: `<clip_low>`
   **Input type**: *Optional, double*
   **Default**: *0.0*
   **Description**: Sets the low clip value for the particle velocity.

9. **Particle velocity boundary conditions**: `<partvelBC_eq_gasvelBC>`
   **Input type**: *Optional, double*
   **Default**: *false*
   **Description**: If this value is true, the particle velocity boundary conditions are the same as the gas velocity boundary conditions.
   WARNING: This should NOT be turned on! It will cause problems!

10. **Minimum velocity ratio**: `<min_vel_ratio>`
    **Input type**: *Optional, positive double*
    **Default**: *0.1*
    **Description**: This keeps the particle velocity from becoming too different from the gas velocity.

11. **Number of iterations**: `<iter>`
    **Input type**: *Optional, positive integer*
    **Default**: *15*
    **Description**: An iterative procedure is required to determine the particle velocity, since it appears in the Balachandar particle velocity model implicitly. This tag controls the maximum iterations that may be run.

12. **Tolerance**: `<tol>`
    **Input type**: *Optional, positive double*
    **Default**: *1e-15*
    **Description**: This defines a maximum residual for the expression for the particle velocity appearing in the Balachandar particle velocity model; i.e., if $[(LHS - RHS) \leq \text{tol}]$ then the iterative procedure finishes.

**Weight Transport Equation Options, <Weights>**

1. **Do Diffusion?**: `<doDiff>`
   **Input type**: *Optional, boolean*
   **Default**: *false*
   **Description**: Boolean to determine whether transport equation's RHS will include a diffusion term. If true, it uses a turbulent Prandtl number to determine the diffusivity. The turbulent Prandtl number can be set, but is 0.4 by default. See `<turbulentPrandtlNumber>` tag below.

2. **Do Convection?**: `<doConv>`
   **Input type**: *Optional, boolean*
   **Default**: *false*
   **Description**: Boolean to determine whether transport equation's RHS will include a convection term. Because the weights are DQMOM scalars, and the weight transport equations (originally) come from the NDF transport equation, the velocity used for the convection term is the particle velocity, conditioned on particle size. This velocity comes from the DQMOM particle velocity model (see the `<VelModel>` tag above).

3. **Convection scheme**: `<conv_scheme>`
   **Input type**: *Optional, string*
   **Default**: *upwind*
   **Description**: Tag to determine the convection scheme used. Valid options are:

   - `upwind` - use the upwind convection scheme
   - `super_bee` - use the super-bee convection scheme

4. **Turbulent Prandtl Number**: `<turbulentPrandtlNumber>`
   **Input type**: *Optional, double*
   **Default**: *0.4*
   **Description**: If diffusion is turned on, a turbulent Prandtl number is required to determine the diffusion coefficient. The Prandtl number is defined as:

$$Pr = \frac{\nu}{D} \tag{7.90}$$

   where $\nu$ is the fluid viscosity, and $D$ is the scalar diffusivity. This is equivalent to the turbulent Schmidt number.

5. **Initialization function**: `<initialization>`
   **Attributes**: type
   **Default**: none
   **Description**: There are several options for initialization functions. These are as follows:

   - *Constant value initialization*, `constant` - Initializes the weights to be a constant throughout the domain; the constant value is the same for all weights of all environments. The block looks like:

     ```
     <initialization type="constant">
             <constant>...</constant>
     </initialization>
     ```

     where the `<constant>` tags encapsulate a double.

   - *Environment-specific constants*, `env_constant` - Initializes the weights to be a constant throughout the domain; the constant value is different for the weights of each environment. The block looks like:

     ```
     <initialization type="env_constant" qn="..." value="...">
     ```

     where "qn" is an integer specifying an environment (or quadrature node), and "value" is a double, and is the value to which the weight of that environment will be initialized.

   - *Step-function initialization*, `step` - Initializes the weights to be a step function. Several attributes of the step function must be specified; these include:
     - `<step_direction>` - Direction in which the step occurrs (x, y, z) (Required, no default)
     - `<step_value>` - The step goes from 0 to `<step_value>` (double) (Required, no default)
     - `<step_start>` - Physical location at which step begins (double) (No default)
     - `<step_end>` - Physical location at which step ends (double) (No default)
     - `<step_cellstart>` - Cell location at which step begins (integer) (No default)
     - `<step_cellend>` - Cell location at which step ends (integer) (No default)

     The step function initialization block looks like:

     ```
     <initialization type="step">
             <step_direction> ... </step_direction>
             <step_value> ... </step_value>
             <step_start> ... </step_start>
             <step_end> ... </step_end>
     </initialization>
     ```

     or, alternatively, the start and end of the step could be specified using the cell locations, which would look like:

     ```
     <initialization type="step">
             <step_direction> ... </step_direction>
             <step_value> ... </step_value>
             <step_cellstart> ... </step_cellstart>
             <step_cellend> ... </step_cellend>
     </initialization>
     ```

- *Environmental step-function initialization*, `env_step` - Initializes the weights to step functions, with the value of the step function different for each environment. The attributes are the same as for the step-function initialization, except that the `<step_value>` tag becomes a set of `<env_step_value>` tags. These tags have the following attributes:
  - `qn` - the environment (or quadrature node) to which the `<env_step_value>` tag applies
  - `value` - the value to which this environment's step function should be initialized

  so that the tags would look like:

```
<initialization type="env_step">
        <step_direction> ... </step_direction>
        <step_start> ... </step_start>
        <step_end> ... </step_end>
        <env_step qn=" ... " value = " ... " >
        ...
        <env_step qn=" ... "  value=" ... " >
</initialization>
```

6. **Scaling constant**: `<scaling_const>`
   **Input type**: *Required, double*
   **Default**: *1.0*
   **Description**: Value by which to scale the weights. The actual values of the weights are very high, as the weights represent numbers of particles (typically greater than 1e6). Because the values of the weights are going into the **A** matrix, large weight values can cause **A** to become very ill-conditioned. Scaling the values of the weights can help control condition numbers for **A**.

7. **Clipping**: `<Clipping>`
   **Description**: Values of weights can attain non-physical values (e.g. negative numbers of particles, large and physically unrealistic numbers of particels, etc.). Clipping is implemented to prevent non-physical values. The weights have three types of clipping:

   - `<low>` - Low clipping represents the lowest value the weights can attain (in most cases, 0)
   - `<small>` - As a result of the weight/weighted abscissa formulation of DQMOM, the abscissa values are not calculated by themselves; they are bound to the weights. Many models thus require dividing the weighted abscissas by the weights. When the weights become very small, this leads to physically unrealistic abscissa values. The small clipping sets a minimum value at which weights can divide weighted abscissas; when the weights are lower than this small clipping, the abscissas are set to 0.
   - `<high>` - High clipping represents an upper limit on values of weights.

### Weighted Abscissa Transport Equation Options, <Ic>

Many of the options for the weighted abscissa transport equation options are the same as the weight transport equations. Only the differences are highlighted in this section.

1. **Initialization**: **Description**: The initialization function options are more limited for weighted abscissas than for weights. If abscissas coincide, this creates singularities in **A**. Thus, the constant and step function initialization functions must be environment-specific

2. **Scaling constant**: **Description**: The scaling constant is for the abscissa, and not for the weighted abscissa. Because the weight is scaled, there are *two* scaling constants for each weighted abscissa: one scaling constant for the weights, the other for the abscissa.

3. **Clipping**: **Description**: The clipping values are set for the abscissas, and not the weighted abscissas. Additionally, there is no "small" clipping value, as there is no dividing by the abscissas.

4. **Models**: `<model>` **Attributes**: label
   **Description**: Attribute determines which model (identified by the label) should be associated with this internal coordinate. The label is an arbitrary string that must match a label for a model.

### Moments, <Moment>

An important part of DQMOM is selecting a set of moments that will be used to provide closure for the NDF. These moments are then used to construct the quadrature-approximated moment transport equations, which are written in the linear form $\mathbf{Ax} = \mathbf{B}$. The moments selected have a strong influence on the behavior of the matrix **A**.

1. **Moment**: `<Moment>`
   **Input type**: *Required, multiple integers*
   **Default**: N/A
   **Description**: The moments are specified as sets of integer vectors. The `<Moment>` block looks like:

   ```
   <Moment><m>[ ... , ... , ... ]</m></Moment>
   ...
   <Moment><m>[ ... , ... , ... ]</m></Moment>
   ```

   Note that the moment index vectors must be of size $N_\xi$, where $N_\xi$ is the number of internal coordinates, and there must be $(N_\xi + 1)\, N$ total moments, where $N$ is the number of environments (or quadrature nodes).

### Verification, <Verify_Linear_Solver>, <Verify_AB_Construction>

Of major importance is the verification procedure for DQMOM. There are two important processes in the DQMOM code that must be verified: the construction of $\mathbf{Ax} = \mathbf{B}$, and the linear solver's solution to $\mathbf{Ax} = B$. Both have verification mechanisms built in. Verification is not something that can be "turned on" in the input file - it must be turned on using compiler directives. The file Directives.h in the Arches component directory contains several directives related to verification, two of which are `VERIFY_LINEAR_SOLVER` and `VERIFY_AB_CONSTRUCTION`. For verification, some additional information must be put into the input file. This information is specified for each verification procedure. Most of these tags point to files containing information which is used to construct $\mathbf{Ax} = \mathbf{B}$. These files are available in the Arches `inputs/` directory.

1. **Verification of Linear Solver**: `<Verify_Linear_Solver>`

    - `<A>` - File containing A matrix
    - `<X>` - File containing X vector (verification solution)
    - `<B>` - File containing B vector
    - `<normR>` - File containing norm of the residual vector
    - `<norms>` - File containing several norms of vectors
    - `<dimension>` - The dimension of the linear system being solved
    - `<tolerance>` - Error tolerance

2. **Verification of A, B construction process**: `<Verify_AB_Construction>`

    - `<A>` - File containing A matrix
    - `<B>` - File containing B vector
    - `<input>` - File containing various inputs (e.g. model terms)
    - `<moments>` - File containing moments
    - `<number_environments>` - Number of environments used to represent the NDF
    - `<number_internal_coordinates>` - Number of internal coordinates parameterizing the NDF
    - `<tolerance>` - Error tolerance

### 7.3.9 Models

The <Models> tags contain several tags that are not specific to models. The outline of a typical <Models> section looks like:

```
<Models>
        <model
        label = " ... "
        type  = " ... "   >

                <ICVars>

                        <variable
                        label = " ... "
                        role = " ... "  >

                </ICVars>

                <scalarVars>

                        <variable
                        label = " ... "
                        role   = " ... " >

                </scalarVars>
```

```
            <low_clip>  ... </low_clip>
            <high_clip> ... </high_clip>

        </model>

        ...

</Models>
```

The sections have the following significance:

- `<model>` - there is a `<model>` tag block for each coal model
- `<ICVars>` - this tag block contains all of the internal coordinate variables that the model depends on (for example, if a kinetic model depends on temperature, and temperature is being tracked as an internal coordinate variable, the temperature would be included in this `<ICVars>` block)
- `<scalarVars>` - this tag block contains all of the scalar variables that the model depends on
- `<low_clip>` - When values of the model are lower than `<low_clip>`, the value is clipped to `<low_clip>`
- `<high_clip>` - When values of the model are higher than `<high_clip>`, the value is clipped to `<high_clip>`

1. **Model**: `<model>`
   **Attributes**: label, type
   **Description**: Attributes determine the label the velocity model is given, and the type of velocity model being used. The label is an arbitrary string. The type takes on these values:
   - `KobayashiSarofimDevol` - Use the Kobayashi-Sarofim coal devolatilization model
   - `ConstantModel` - Use a constant model
   - `SimpleHeatTransfer` - Use a simple particle heat transfer model
   - `XDrag, YDrag, ZDrag` - Use a standard drag model for the x-, y-, and z-velocity of the particle
2. **Internal Coordinate Variable**: `<ICVars> <variable> </ICVars>`
   **Attributes**: label, role
   **Description**: Attributes set the label of the internal coordinate variable, and the role that the internal coordinate variable plays in the model. The label must match a label for one of the internal coordinate variables (see the "label" attribute of the `<Ic>` tag, above). The role must match one of the role names specified by the model, in the model code. These include:
   - KobayashiSarofimDevol
     - `raw_coal_mass`
     - `particle_temperature`
   - SimpleHeatTransfer

- `particle_length`
- `raw_coal_mass`
- `particle_temperature`

- XDrag / YDrag / ZDrag
  - `particle_length`
  - `particle_xvel / particle_yvel / particle_zvel`

3. **Scalar Variable**: `<scalarVars> <variable> </scalarVars>`
   **Attributes**: label, role
   **Description**: As with the above `<variable>` tag, this tag specifies information about scalar variables a model may depend on. The label attribute indicates the label of the scalar variable, and the role specifies the role that scalar variable will play in the model. The label attribute must match a label for one of the scalar variables. The role must match one of the role names specified by the model, in the model code. Currently, scalar variables do not play a role in any models currently implemented.

### 7.3.10 Digital Filter Generator

In order to utilize the digital filter turbulent inlet condition in 7.3.2, the executable `DigitalFilterGenerator` is the best way to generate the table needed. This executable lies in the `StandAlone` folder. Its usage is simply

```
./DigitalFilterGenerator input_file output_table_file
```

If not output table file is specified this writes out to `DFGInlet`. The input file is a small text file with the grid size and geometries specified on individual lines. Currently it must be specified in a specific order.

1. Grid resolution `<int> <int> <int>`
2. Lower grid points `<double> <double> <double>`
3. Upper grid points `<double> <double> <double>`
4. Inlet face `<string>` (x+, x-, y-, y+, z-, z+)
5. Inlet geometry shape `<string>` (box/circle/ellipse/annulus)
6. Geometry parameters `<double>` (varies on geometry, same order as BCGeom objects )
7. Number of realizations in file `<int>`
8. Lengthscales `<double> <double> <double>`
9. Accuracy of filter (min 2) `<double>`
10. Average velocity vector `<double> <double> <double>`
11. Average Reynold's stress tensor (lower diagonal) `<double> <double> <double> <double> <double> <double>` $R_{11},\ R_{21},\ R_{22},\ R_{31},\ R_{32},\ R_{33}$

12. Use spatial variations `<string>` (y, yes, n, no)
13. Variation direction `<string>` (x, y, z, none) used if a planar geometric relation is needed
14. Velocity variation type `<string>` (tanh, power, laminar, none)
15. Stress variation type `<string>` (channel, jet, none)
16. Lengthscale variation type `<string>` (channel, jet, none)

An example input file for the digital filter is located in `StandAlone/inputs/ARCHES/DigitalFilter/ChannelInletGenerator.txt`. It is recommended to gzip the file that is created to save space.

## 7.4 Examples

The following ARCHES examples illustrate the diverse set of problems that can be solved using the ARCHES component of Uintah code. The first two examples exemplify techniques used to verify various ARCHES algorithms that were implemented in the code. The following three [or more] examples illustrate the kinds of problems that ARCHES can solve. The input files used here can be used as templates to build similar input files for similar problems. Due to the complexity of ARCHES simulations, exact solutions (with the exception of MMS) do not exist. Hence the emphasis on model validation, or the comparison of simulation with experimental results. Model validation provides a framework that allows the simulation scientist to be confident in his or her results in the absence of analytical solutions. All modeling should be accompanied by some form of validation analysis.

### Almgren MMS

#### Problem Description

Methods of Manufactured Solutions (MMS) are verification tools that are used with computer codes such as ARCHES that seek to solve the Navier-Stokes Equations. They are extremely useful for finding programming errors and ensuring expected behavior of the computer code. The Almgren MMS is especially easy to implement because of the absence of source terms that must be added to the transport equations. ARCHES uses a second-order spatial discretization scheme and a first-order scheme in the temporal direction. Therefore, if the Almgren MMS problem is run in Arches at different mesh resolutions and the normalized error plotted on a semilog plot, the slope of the line should be 2. To facilitate this exercise, a shell script has been written to perform this analysis.

#### Simulation Specifics

**Component used:** ARCHES

**Input file name:** `almgrenMMS.ups`

**Command used to run input file:**

`./runAlmgren.sh`

If you examine the shell script you will see the following line of code: `mpirun -np 1 sus inputs/UintahRelease/ARCHES/a`
This is call to run the ARCHES via sus.

**Simulation Domain:** 1.0 x 1.0 x 3.0 m

**Cell Spacing:**
0.3125 x 0.3125 x 0.275 m

**Example Runtimes:**
113.2 seconds (1 processor, 2.4 GHz Intel Core 2)

**Physical time simulated:**
1.0 sec.

# Periodic Box Problem

## Problem Description

The Periodic Box Problem indicates how well ARCHES is modeling the kinetic energy contained in the turbulence modeled on the grid and at a sub-grid level. The LES algorithm transfers kinetic energy from cell to cell in the ARCHES structured grid. Turbulence models such as "compdynamicprocedure," "dynamicprocedure," and "smagorinsky" are used to model kinetic energy at the sub-grid level. Ideally, there would be a seamless transition between the resolved turbulence and sub-grid models at the Nyquist limit. **SHOW SAMPLE PLOT** Experience has shown that this is not normally the case. By plotting the kinetic energy as a function of the wave number, it is possible to determine how well the kinetic energy dissipation is being modeled by the code. The Periodic Box problem is initialized with a kinetic energy (turbulence) profile from Direct Numerical Simulation (DNS). As the simulation proceeds that energy is dissipated.

## Simulation Specifics

**Component used:** ARCHES

**Input file name:** `periodic.ups`

**Command used to run input file:**

This simulation, like many ARCHES simulations, requires another file, in addition to the input file called by sus. That file is the initial condition called upon in periodic.ups by

```
mpirun -np 1 sus inputs/UintahRelease/ARCHES/periodic.ups
```

**Simulation Domain:** 0.565 x 0.565 x 0.565 m

**Cell Spacing:**

0.0177 x 0.0177 x 0.0177 m

**Example Runtimes:**

2 minutes (1 processor, 2.4 GHz Intel Core 2 )

**Physical time simulated:**

0.1 sec.

# Helium Plume

## Problem Description

Helium plumes are classical experiments that allow for the easy capture of turbulent mixing data that can be used to validate the turbulent mixing models used in LES algorithms such as Arches. The non-reacting nature of the plume makes it easy to capture experimental data without damaging expensive equipment. Reacting flows require special mixing tables that contain temperature, pressure and composition as a function of the transported scalars in Arches. The coldFlowMixingModel is used to determine the mixing of **isothermal?** streams. After the mixing model is specific in the .ups file, the temperature and densities of the two mixing streams are specified.

## Simulation Specifics

**Component used:** ARCHES

**Input file name:** `helium_1m.ups`

**Command used to run input file:**

```
mpirun -np 8 sus inputs/UintahRelease/ARCHES/helium_1m.ups
```

**Simulation Domain:** 3.0 x 3.0 x 3.0 m

**Cell Spacing:**
0.06 x 0.06 x 0.0.6 m

**Example Runtimes:**
54 minutes (8 processors, 2.8 GHz Xeon)

**Physical time simulated:**
5.0 sec.

## Results

# Methane Plume

## Problem Description

This methane plume is geometrically identical to the Helium Plume problem. The difference is the addition of chemistry. Instead of unreacting, isothermal fluids mixing, a fuel is reacting to combustion products inside of the computational domain. This chemistry is captured via a mixing table. The input file is pointed to the mixing table which contains state variables and species mass fractions as a function of mixture fraction, heat loss, and mixture fraction variance.

## Simulation Specifics

**Component used:** ARCHES

**Input file name:** `helium_1m.ups`

Note that the input file is pointed to the mixing table (inputs/UintahRelease/ARCHES/CH4_equil_clipped.mxn.gz)

**Command used to run input file:**

```
mpirun -np 8 sus inputs/UintahRelease/ARCHES/helium_1m.ups
```

**Simulation Domain:** 3.0 x 3.0 x 3.0 m

**Cell Spacing:**

0.06 x 0.06 x 0.0.6 m

**Example Runtimes:**

2 hours 10 minutes (8 processors, 2.8 GHz Xeon)

**Physical time simulated:**

5.0 sec.

## Results

## Fast Cookoff

### Problem Description

The Fast Cookoff test is a procedure used for hazard classification of energetic materials. The object is immersed over a jet fuel pool fire and the reaction, if any, is observed. Current protocol requires that the full size article must be subjected to this test, making such procedures prohibitively expensive and unfeasible for articles such as solid rocket motors. An alternative procedure has been proposed, combining sub-scale experiments with computer simulation. Through validation and uncertainty quantification procedures, the computer simulation tool (ARCHES) can be used as a surrogate for full-scale experimental testing. This Fast Cookoff problem includes a reacting flow as well as an MPMARCHES object. After performing the simulation, the incident heat flux to the cylinder can be extracted.

### Simulation Specifics

**Component used:** MPMARCHES

**Input file name:** `fastcookoff.ups`

**Command used to run input file:**

Note that the input file is pointed to the mixing table (inputs/UintahRelease/ARCHES/sandia_jp8_flmlt_cg.mxn)

`mpirun -np 64 sus inputs/UintahRelease/ARCHES/fastcookoff.ups` To extract the incident heat flux to the cylinder, use the faceextract and timeextract utilities. ¡How to do this¿

**Simulation Domain:** 24.0 x 24.0 x 24.0 m

**Cell Spacing:**

0.24 x 0.24 x 0.24 m

**Example Runtimes:**

7 hours 27 minutes (64 processors, 2.8 GHz Xeon)

**Physical time simulated:**

10.0 sec.

### Results

# 7.5 References

# Bibliography


[1] Rodney O. Fox. Optimal moment sets for multivariate direct quadrature method of moments. *Ind. Eng. Chem. Res.*, 48:9686–9696, 2009.

[2] M. Germano, U. Piomelli, P. Moin, and W. H. Cabot. A dynamic subgrid-scale eddy viscosity model. *Physics of Fluids A:Fluid Dynamics*, 3(7):1760–1765, 1991.

[3] S. Gottlieb, C.-W. Shu, and E. Tadmor. Strong stability-preserving high-order time discretization methods. *SIAM Review*, 43(1):89–112, 2001.

[4] M. Klein, A. Sadiki, and J. Janicka. A digital filter based generation of inflow data for spatially developing direct numerical or large eddy simulations. *Journal of Computational Physics*, 186:652–665, 2003.

[5] D. K. Lilly. A proposed modifcation of the Germano subgrid-scale closure method. *Physics of Fluids A*, 4:633–635, 1992.

[6] P. Moin, K. Squires, W. Cabot, and S. Lee. A dynamic subgrid-scale model for compressible turbulence and scalar transport. *Physics of Fluids A*, 3(11):2746–2757, 1991.

[7] Y. Morinishi, T.S. Lund, O.V. Vasilyev, and P. Moin. Fully conservatice higher order finite difference schemes for incompressible flow. *Journal of Computational Physics*, 143:90–124, 1998.

[8] Julien Pedel. Large eddy simulations of coal jet flame ignition using the direct quadrature method of moments, June 2012.

[9] H. Pitsch and H. Steiner. Large-eddy simulation of a turbulent piloted methane/air diffusion flame (Sandia flame D). *Physics of Fluids*, 12(10):2541–2544, 2000.

[10] S. B. Pope. *Turbulent Flows*. Cambridge University Press, Cambridge, UK, 2000.

[11] Charles Martin Reid. *An instrumentalist approach to validation: a quantitative assessment of a novel coal gasification model*. Ph.d thesis, University of Utah, May 2012.

[12] J. Smagorinsky. General circulation experiments with the primitive equations. *Monthly Weather Review*, 91(3):99–106, 1963.

# Chapter 8

# ICE

## 8.1 Introduction

The work presented here describes a multi-material CFD approach designed to solve "full physics" simulations of dynamic fluid structure interactions involving large deformations and material transformations (e.g., phase change). "Full physics" refers to problems involving strong interactions between the fluid field and solid field temperatures and velocities, with a full Navier Stokes representation of fluid materials and the transient, nonlinear response of solid materials. These interactions may include chemical or physical transformation between the solid and fluid fields.

The theoretical and algorithmic basis for the multi-material CFD algorithm presented here is based on a body of work of several investigators at Los Alamos National Laboratory, primarily Bryan Kashiwa, Rick Rauenzahn and Matt Lewis. Several reports by these researchers are publicly available and are cited herein. It is largely through our personal interactions that we have been able to bring these ideas to bear on the simulations described herein.

An exposition of the governing equations is given in the next section, followed by an algorithmic description of the solution of those equations. This description is first done separately for the materials in the Eulerian and Lagrangian frames of reference, before details associated with the integrated approach are given.

### 8.1.1 Governing Equations

The governing multi-material model equations are stated and described, but not developed, here. Their development can be found in [6]. Here, our intent is to identify the quantities of interest, of which there are eight, as well as those equations (or closure models) which govern their behavior. Consider a collection of $N$ materials, and let the subscript r signify one of the materials, such that $\mathrm{r} = 1, 2, 3, \ldots, N$. In an arbitary volume of space $V(\mathbf{x}, t)$, the averaged thermodynamic state of a material is given by the vector $[M_\mathrm{r}, \mathbf{u}_\mathrm{r}, e_\mathrm{r}, T_\mathrm{r}, v_\mathrm{r}, \theta_\mathrm{r}, \boldsymbol{\sigma}_\mathrm{r}, p]$, the elements of which are the r-material mass, velocity, internal energy, temperature, specific volume, volume fraction, stress, and the equilibration pressure. The r-material averaged density is $\rho_\mathrm{r} = M_\mathrm{r}/V$. The rate of change of

the state in a volume moving with the velocity of r-material is:

$$\frac{1}{V}\frac{D_{\rm r}M_{\rm r}}{Dt} = \sum_{s=1}^{N}\Gamma_{\rm rs} \tag{8.1}$$

$$\frac{1}{V}\frac{D_{\rm r}(M_{\rm r}\mathbf{u}_{\rm r})}{Dt} = \theta_{\rm r}\boldsymbol{\nabla}\cdot\boldsymbol{\sigma} + \boldsymbol{\nabla}\cdot\theta_{\rm r}(\boldsymbol{\sigma}_{\rm r}-\boldsymbol{\sigma}) + \rho_{\rm r}\mathbf{g} + \sum_{s=1}^{N}\mathbf{f}_{\rm rs} + \sum_{s=1}^{N}\mathbf{u}_{\rm rs}^{+}\Gamma_{\rm rs} \tag{8.2}$$

$$\frac{1}{V}\frac{D_{\rm r}(M_{\rm r}e_{\rm r})}{Dt} = -\rho_{\rm r}p\frac{D_{\rm r}v_{\rm r}}{Dt} + \theta_{\rm r}\boldsymbol{\tau}_{\rm r}:\nabla\mathbf{u}_{\rm r} - \boldsymbol{\nabla}\cdot\mathbf{j}_{\rm r} + \sum_{s=1}^{N}q_{\rm rs} + \sum_{s=1}^{N}h_{\rm rs}^{+}\Gamma_{\rm rs} \tag{8.3}$$

Equations (8.1-8.3) are the averaged model equations for mass, momentum, and internal energy of r-material, in which $\boldsymbol{\sigma}$ is the mean mixture stress, taken here to be isotropic, so that $\boldsymbol{\sigma} = -p\mathbf{I}$ in terms of the hydrodynamic pressure $p$. The effects of turbulence have been explicitly omitted from these equations, and the subsequent solution, for the sake of simplicity. However, including the effects of turbulence is not precluded by either the model or the solution method used here.

In Eq. (8.2) the term $\sum_{s=1}^{N}\mathbf{f}_{\rm rs}$ signifies a model for the momentum exchange among materials. This term results from the deviation of the r-field stress from the mean stress, averaged, and is typically modeled as a function of the relative velocity between materials at a point. (For a two material problem this term might look like $\mathbf{f}_{12} = K_{12}\theta_1\theta_2(\mathbf{u}_1-\mathbf{u}_2)$ where the coefficient $K_{12}$ determines the rate at which momentum is transferred between materials). Likewise, in Eq. (8.3), $\sum_{\rm s=1}^{N}q_{\rm rs}$ represents an exchange of heat energy among materials. For a two material problem $q_{12} = H_{12}\theta_1\theta_2(T_2-T_1)$ where $T_{\rm r}$ is the r-material temperature and the coefficient $H_{\rm rs}$ is analogous to a convective heat transfer rate coefficient. The heat flux is $\mathbf{j}_{\rm r} = -\rho_{\rm r}b_{\rm r}\boldsymbol{\nabla}T_{\rm r}$ where the thermal diffusion coefficient $b_{\rm r}$ includes both molecular and turbulent effects (when the turbulence is included).

In Eqs. (8.1-8.3) the term $\Gamma_{\rm rs}$ is the rate of mass conversion from s-material into r-material, for example, the burning of a solid or liquid reactant into gaseous products. The rate at which mass conversion occurs is governed by a reaction model. In Eqs. (8.2) and (8.3), the velocity $\mathbf{u}_{\rm rs}^{+}$ and the enthalpy $h_{\rm rs}^{+}$ are those of the s-material that is converted into r-material. These are simply the mean values associated with the donor material.

The temperature $T_{\rm r}$, specific volume $v_{\rm r}$, volume fraction $\theta_{\rm r}$, and hydrodynamic pressure $p$ are related to the r-material mass density, $\rho_{\rm r}$, and specific internal energy, $e_{\rm r}$, by way of equations of state. The four relations for the four quantites $(T_{\rm r}, v_{\rm r}, \theta_{\rm r}, p)$ are:

$$e_{\rm r} = e_{\rm r}(v_{\rm r}, T_{\rm r}) \tag{8.4}$$

$$v_{\rm r} = v_{\rm r}(p, T_{\rm r}) \tag{8.5}$$

$$\theta_{\rm r} = \rho_{\rm r}v_{\rm r} \tag{8.6}$$

$$0 = 1 - \sum_{s=1}^{N}\rho_{\rm s}v_{\rm s} \tag{8.7}$$

Equations (8.4) and (8.5) are, respectively, the caloric and thermal equations of state. Equation (8.6) defines the volume fraction, $\theta$, as the volume of r-material per total material volume, and with that definition, Equation (8.7), referred to as the multi-material equation of state, follows. It defines the unique value of the hydrodynamic pressure $p$ that allows arbitrary masses of the multiple materials to identically fill the volume $V$. This pressure is called the "equilibration" pressure [8].

A closure relation is still needed for the material stress $\boldsymbol{\sigma}_{\rm r}$. For a fluid $\boldsymbol{\sigma}_{\rm r} = -p\mathbf{I} + \boldsymbol{\tau}_{\rm r}$ where the deviatoric stress is well known for Newtonian fluids. For a solid, the material stress is the Cauchy stress. The Cauchy stress is computed using a solid constitutive model and may depend on the the rate of deformation, the current state of deformation ($\mathbf{E}$), the temperature, and possibly a number of history variables. Such a relationship may be expressed as:

$$\boldsymbol{\sigma}_{\rm r} \equiv \boldsymbol{\sigma}_{\rm r}(\boldsymbol{\nabla}\mathbf{u}_{\rm r}, \mathbf{E}_{\rm r}, T_{\rm r}, \ \ldots) \tag{8.8}$$

The approach described here imposes no restrictions on the types of constitutive relations that can be considered. More specific discussion of some of the models used in this work is found in Sec. **??**

Equations (8.1-8.8) form a set of eight equations for the eight-element state vector, $[M_{\rm r}, \mathbf{u}_{\rm r}, e_{\rm r}, T_{\rm r}, v_{\rm r}, \theta_{\rm r}, \boldsymbol{\sigma}_{\rm r}, p]$, for any arbitrary volume of space $V$ moving with the r-material velocity. The approach described here uses the reference frame most suitable for a particular material type. As such, there is no guarantee that arbitrary volumes will remain coincident for materials described in different reference frames. This problem is addressed by treating the specific volume as a dynamic variable of the material state which is integrated forward in time from initial conditions. In so doing, at any time, the total volume associated with all of the materials is given by:

$$V_t = \textstyle\sum_{r=1}^{N} M_{\rm r} v_{\rm r} \tag{8.9}$$

so the volume fraction is $\theta_{\rm r} = M_{\rm r} v_{\rm r} / V_t$ (which sums to one by definition). An evolution equation for the r-material specific volume, derived from the time variation of Eqs. (8.4-8.7), has been developed in [6]. It is stated here as:

$$\begin{aligned} \frac{1}{V}\frac{D_{\rm r}(M_{\rm r} v_{\rm r})}{Dt} = f_{\rm r}^{\theta}\boldsymbol{\nabla}\cdot\mathbf{u} \ &+ \ \left[v_{\rm r}\Gamma_{\rm r} - f_{\rm r}^{\theta}\textstyle\sum_{s=1}^{N} v_{\rm s}\Gamma_{\rm s}\right] \\ &+ \ \left[\theta_{\rm r}\beta_{\rm r}\frac{D_{\rm r}T_{\rm r}}{Dt} - f_{\rm r}^{\theta}\textstyle\sum_{s=1}^{N}\theta_{\rm s}\beta_{\rm s}\frac{D_{\rm s}T_{\rm s}}{Dt}\right] . \end{aligned} \tag{8.10}$$

where $f_{\rm r}^{\theta} = \frac{\theta_{\rm r}\kappa_{\rm r}}{\sum_{\rm s=1}^{N}\theta_{\rm s}\kappa_{\rm s}}$, and $\kappa_{\rm r}$ is the r-material bulk compressibility.

The evaluation of the multi-material equation of state (Eq. (8.7)) is still required in order to determine an equilibrium pressure that results in a common value for the pressure, as well as specific volumes that fill the total volume identically.

A description of the means by which numerical solutions to the equations in Section 8.2 are found is presented next. This begins with separate, brief overviews of the methodologies used for the Eulerian and Lagrangian reference frames. The algorithmic details necesssary for integrating them to achieve a tightly coupled fluid-structure interaction capability is provided in Sec. 10.

## 8.2 Algorithm Description

The Eulerian method implemented here is a cell-centered, finite volume, multi-material version of the ICE (for Implicit, Continuous fluid, Eulerian) method [5] developed by Kashiwa and others at Los Alamos National Laboratory [7]. "Cell-centered" means that all elements of the state are colocated at the grid cell-center (in contrast to a staggered grid, in which velocity components may be centered at the faces of grid cells, for example). This colocation is particularly important in regions where a material mass is vanishing. By using the same control volume for mass and momentum it can be assured that as the material mass goes to zero, the mass and momentum also go to zero at the same rate, leaving a well-defined velocity. The technique is fully compressible, allowing wide generality in the types of problems that can be addressed.

Our use of the cell-centered ICE method employs time splitting: first, a Lagrangian step updates the state due to the physics of the conservation laws (i.e., right hand side of Eqs. 8.1-8.3); this is followed by an Eulerian step, in which the change due to advection is evaluated. For solution in the Eulerian frame, the method is well developed and described in [7].

In the mixed frame approach used here, a modification to the multi-material equation of state is needed. Equation (8.7) is unambiguous when all materials are fluids or in cases of a flow consisting of dispersed solid grains in a carrier fluid. However in fluid-structure problems the stress state of a submerged structure may be strongly directional, and the isotropic part of the stress has nothing to do with the hydrodynamic (equilibration) pressure $p$. The equilibrium that typically exists between a fluid and a solid is at the interface between the two materials: there the normal part of the traction equals the pressure exerted by the fluid on the solid over the interface. Because the orientation of the interface is not explicitly known at any point (it is effectively lost in the averaging) such an equilibrium cannot be computed.

The difficulty, and the modification that resolves it, can be understood by considering a solid material in tension coexisting with a gas. For solid materials, the equation of state is the bulk part of the constitutive response (that is, the isotropic part of the Cauchy stress versus specific volume and temperature). If one attempts to equate the isotropic part of the stress with the fluid pressure, there exist regions in pressure-volume space for which Eq. (8.7) has no physical solutions (because the gas pressure is only positive). This can be seen schematically in Fig. **??**, which sketches equations of state for a gas and a solid, at an arbitrary temperature.

Recall that the isothermal compressiblity is the negative slope of the specific volume versus pressure. Embedded structures considered here are solids and, at low pressure, possess a much smaller compressibility than the gasses in which they are submerged. Nevertheless the variation of condensed phase specific volume can be important at very high pressures, where the compressibilities of the gas and condensed phase materials can become comparable (as in a detonation wave, for example). Because the speed of shock waves in materials is determined by their equations of state, obtaining accurate high pressure behavior is an important goal of our FSI studies.

To compensate for the lack of directional information for the embedded surfaces, we evaluate the solid phase equations of state in two parts. Above a specified postive threshold pressure (typically 1 atmosphere), the full equation of state is respected; below that threshold pressure, the solid phase pressure follows a polynomial chosen to be $C^1$ continuous at the threshold value and which approaches zero as the specific volume becomes large. The effect is to decouple the solid phase specific volume from the stress when the isotropic part of the stress falls below a threshold value. In regions of coexistence at states below the threshold pressure, $p$ tends to behave according to the fluid equation of state (due to the greater compressibility) while in regions of pure condensed phase material $p$ tends rapidly toward zero and the full material stress dominates the dynamics as it should.

## 8.3 Uintah Specification

### 8.3.1 Basic Inputs

Each Uintah component is invoked using a single executable called *sus* , which chooses the type of simulation to execute based on the *SimulationComponent* tag in the input file. In the case of ICE simulations, this looks like:

```
<SimulationComponent type="ice" />
```

near the top of the inputfile. The system of units **must** be consistent (mks, cgs) and the majority of input files will be in Meter-Kilogram-Sec system. For small length scale simulations, it is advantageous to use "bomb units", which are a consistent set of units for microgram mass scales, centimeter

length scales and micosecond timescales. A conversion table of relevant physical quantities from mks to bomb units can be found in Appendix A.

### 8.3.2 Semi-Implicit Pressure Solve

The equation for the change in the pressure field $\Delta P$ during a given timestep is given by

$$\frac{dP}{dt} = \frac{\sum_{m=1}^{N} \frac{\dot{m}}{V\rho_m^o} - \sum_{m=1}^{N} \nabla \cdot \widehat{\theta_m \vec{U_m}}^{*f}}{\sum_{m=1}^{N} \frac{\theta_m}{\rho_m^o c_m^2}} \tag{8.11}$$

which can be written in matrix form $Ax = b$ and solved with a linear solver. Details on the notation, discretization of Eq. 8.11 and the formation of $A$ and $b$ can be found in

`src/CCA/Components/ICE/Docs/implicitPressSolve.pdf`

The linear system $Ax = b$ can be solved using the default Uintah:conjugate gradient solver (cg) (slow) or one of the many that are available through the scalable linear solvers and preconditioner package hypre [3]. Experience has shown that the most efficient hypre preconditioner and solver are the pfmg and cg respectively. Below are typical values for both the Uintah:cg and hypre:cg solver

```
<ImplicitSolver>
   <max_outer_iterations>         20    </max_outer_iterations>
   <outer_iteration_tolerance>    1e-8  </outer_iteration_tolerance>
   <iters_before_timestep_restart> 5    </iters_before_timestep_restart>
   <Parameters variable="implicitPressure">

    <tolerance>     1.e-10  </tolerance>

    <!-- CGSolver options -->
    <norm>     LInfinity  </norm>
    <criteria> Absolute   </criteria>

    <!-- Hypre options -->
    <solver>         cg      </solver>
    <preconditioner> pfmg    </preconditioner>
    <maxiterations>  7500    </maxiterations>
    <npre>           1       </npre>
    <npost>          1       </npost>
    <skip>           0       </skip>
    <jump>           0       </jump>
   </Parameters>
</ImplicitSolver>
```

If the user is interested in altering the tolerance to which the equations are solved they should look at

`<tolerance> and <outer_iteration_tolerance>`

| XML tag | Description |
|---|---|
| max_outer_iterations | maximum number of iterations in the outer loop of the pressure solve. |
| outer_iteration_tolerance | tolerance XXXXDX |
| iters_before_timestep_restart | number of outer iterations before a timestep is restarted |
| tolerance | XXXX |

### 8.3.3 Physical Constants

The gravitational constant and a reference pressure are specified in:

```
<PhysicalConstants>
   <gravity>            [0,0,0]   </gravity>
   <reference_pressure> 101325.0  </reference_pressure>
</PhysicalConstants>
```

### 8.3.4 Material Properties

For each ICE material the thermodynamic and transport properties must be specified, in addition to the initial conditions of the fluid inside of each geom_object. Below is the an example of how to specify an invisid ideal gas over square region with dimensions $6mX6mX6m$. The initial conditions of the gas in that region are $T = 300, \rho = 1.179, v_x = 1, v_y = 2, v_z = 3$ (Note, the pressure XML tag is not used as an initial condition and is simply there to make the user aware of what the pressure would be at that thermodynamic state.)

```
<MaterialProperties>
   <ICE>
     <material>
       <EOS type = "ideal_gas">                </EOS>
       <dynamic_viscosity>   0.0               </dynamic_viscosity>
       <thermal_conductivity>0.0               </thermal_conductivity>
       <specific_heat>      716.0              </specific_heat>
       <gamma>               1.4               </gamma>
       <geom_object>
           <box label="wholeDomain">
               <min>       [ 0.0, 0.0, 0.0 ]  </min>
               <max>       [ 6.0, 6.0, 6.0 ]  </max>
           </box>
           <res>                 [2,2,2]      </res>
           <velocity>      [1.,2.,3.]         </velocity>
           <density>       1.1792946927374306 </density>
           <pressure>      101325.0           </pressure>
           <temperature>   300.0              </temperature>
       </geom_object>
     </material>
  </ICE>
</MaterialProperties>
```

### 8.3.5 Equation of State

Below is a list of the various equations of state, along with the user defined constants, that are available. The reader should consult the literature for the theoretical development and applicability of the equations of state to the problem being solved. The most commonly used EOS is the ideal gas law

$$p = (\gamma - 1)c_v \rho T \tag{8.12}$$

and is specified in the input file with:

```
<EOS type="ideal_gas"/>
```

The Thomsen Hartka EOS for cold liquid water (1-100 atm pressure range) is specified with [16, 1]

```
<EOS type="Thomsen_Hartka_water">
  <a>  2.0e-7     </a>    <!-- (K/Pa)     -->
  <b>  2.6        </b>    <!-- (J/kg K^2) -->
  <co> 4205.7     </co>   <!-- (J/Kg K)   -->
  <ko> 5.0e-10    </ko>   <!-- (1/Pa)     -->
  <To> 277.0      </To>   <!-- (K)        -->
  <L>  8.0e-6     </L>    <!-- (1/K^2)    -->
  <vo> 1.00008e-3 </vo>   <!-- (m^3/kg)   -->
</EOS>
```

The input specification for the "JWLC", "JWL++" and "Murnaghan" equations of state from [11] are:

```
<EOS type = "JWLC">
  <A>   2.9867e11   </A>
  <B>   4.11706e9   </B>
  <C>   7.206147e8  </C>
  <R1>    4.95      </R1>
  <R2>    1.15      </R2>
  <om>    0.35      </om>
  <rho0>  1160.0    </rho0>
</EOS>

<EOS type = "JWL">
  <A>     1.6689e12 </A>
  <B>     5.969e10  </B>
  <R1>      5.9     </R1>
  <R2>      2.1     </R2>
  <om>      0.45    </om>
  <rho0>  1835.0    </rho0>
</EOS>

<EOS type = "Murnaghan">
  <n>     7.4       </n>
  <K>     39.0e-11  </K>
  <rho0>  1160.0    </rho0>
  <P0>    101325.0  </P0>
</EOS>

<EOS type = "BirchMurnaghan">
  <n>     7.4       </n>
  <K>     39.0e-11  </K>
  <rho0>  1160.0    </rho0>
  <P0>    101325.0  </P0>
</EOS>
```

The "hard sphere" or "Abel" equation of state for dense gases is

$$p(v - b) = RT \tag{8.13}$$

where b corresponds to the volume occupied by the molecules themselves [15]. Input parameters are specified using:

```
<EOS type="hard_sphere_gas">
   <b> 1.4e-3 </b>
</EOS>
```

Non-idea gas equation of state used in HMX combustion simulations the Twu-Sim-Tassone(TST) EOS is

$$p = \frac{(\gamma - 1)c_v T}{v - b} - \frac{a}{(v + 3.0b)(v - 0.5b)} \tag{8.14}$$

Input parameters are specified using:

```
<EOS type="TST">
  <a>       -260.1385968     </a>
  <b>       7.955153678e-4   </b>
  <u>       -0.5             </u>
  <w>       3.0              </w>
  <Gamma>   1.63             </Gamma>
</EOS>
```

The input parameters for the Tillotson equation of state [4] for soils :

```
<EOS type = "Tillotson">
  <a>       .5      </a>
  <b>       1.3     </b>
  <A>       4.5e9   </A>
  <B>       3.0e9   </B>
  <E0>      6.e6    </E0>
  <Es>      3.2e6   </Es>
  <Esp>     18.0e6  </Esp>
  <alpha>   5.0     </alpha>
  <beta>    5.0     </beta>
  <rho0>    1700.0  </rho0>
</EOS>
```

### 8.3.6 Specific Heat Models

In Uintah, temperature dependent specific heat models are available for ICE materials. Three models currently exist including the Debye model, a common gas component model, and a generalized polynomial model. NOTE: Not all of these models are energy conservative, most notably the component based model. The input specification belongs in the material definition and the input file and takes the form:

```
<SpecificHeatModel type="...">
  ...
</SpecificHeatModel>
```

The **Debye** specific heat model follows the Debye equation for temperature dependence in a solid latice:

$$C_v(T) = 9Nk_B \left(\frac{T}{T_D}\right)^3 \int_0^{\frac{T_D}{T}} \frac{x^4 e^x}{(e^x - 1)^2}$$

where $k_B$ is the Boltzmann constant. The input parameters are the number of atoms $N$ and the Debye temperature $T_D$ in Kelvin. these are specified in the input file as:

```
<SpecificHeatModel type="Debye">
  <Atoms> 3 </Atoms>
  <DebyeTemperature> 290 </DebyeTemperature>
</SepcificHeatModel>
```

The **Component** model is designed to allow the specification of the mole fraction of a number gas species, and the mixture specific heat will be calculated. Gas species supported include $CO_2$, $H_2O$, CO, $H_2$, $O_2$, $N_2$, OH, NO, O and H. Data comes from NASA's thermochemical code and includes fits that run from 300K to 5000K. Outside of these ranges the specific heat is clamped to these endpoints. The input file specification is:

```
<SpecificHeatModel type="Component">
  <XCO2> 0.5 <XCO2>
  <XH2O> 0.4 <XH2O>
  <XCO> 0.0 <XCO>
  <XH2> 0.0 <XH2>
  <XO2> 0.0 <XO2>
  <XN2> 0.0 <XN2>
  <XOH> 0.1 <XOH>
  <XNO> 0.0 <XNO>
  <XO> 0.0 <XO>
  <XH> 0.0 <XH>
</SepcificHeatModel>
```

The **Polynomial** model is designed to be a general polynomial that limits towards an upper asymptote. The form of the equation is:

$$C_v(T) = \frac{\sum_{i=0}^{n} a_i * T^i}{T^n}$$

where $n$ is the maximum order of the polynomial and $a_i$ are the fitting coefficients. There must be $n+1$ coefficients specified as well as a maximum order of the polynomial. Optionally, a minimum and maximum temperature may be assigned that will clamp the specific heat at each end. These default to 0K and 1,000,000K. The input file specification is:

```
<SpecificHeatModel type="Polynomial">
  <MaxOrder> 7 </MaxOrder>
  <Tmin>     250 </Tmin>
  <Tmax>     1e6 </Tmax>
  <coefficient> [1.2, 3.0, 4.0, 5.3, 6.7, 2.2, 4.1, 4.9] </cofficient>
</SepcificHeatModel>
```

### 8.3.7 Exchange Properties

The heat and momentum exchange coefficients $K_{rs}$ and $H_{rs}$, which determine the rate at which momentum and heat are transferred between materials, and are specified in the following format.

```
0->1,   0->2,  0->3
        1->2,  1->3
               2->3
```

For a three material problem the coefficients would be:

```
<exchange_properties>
   <exchange_coefficients>
      <momentum>  [0, 1e15, 1e15 ]     </momentum>
      <heat>      [0, 1e10, 1e10 ]     </heat>
   </exchange_coefficients>
</exchange_properties>
```

### 8.3.8 BoundaryConditions

Boundary conditions must be specified on each face of the computational domain $(x^-, x^+, y^-, y^+, z^-, z^+)$ for the variables $P, \mathbf{u}, \mathbf{T}, \rho, \mathbf{v}$ for each material. The three main types of numerical boundary conditions that can be applied are "Neumann", "Dirichlet" and "Symmetric". A Neumann boundary condition is used to set the gradient or $\frac{\partial q}{\partial n}|_{surface} = value$ at the boundary. The value of the primative variable in the boundary cell is given by,

$$q[\text{boundary cell}] = q[\text{interior cell}] - value * dn; \tag{8.15}$$

if we use a first order upwind discretization of the gradient. Dirichlet boundary conditions set the value of primative variable in the boundary cell using

$$q[\text{boundary cell}] = value; \tag{8.16}$$

```
<Grid>
  <BoundaryConditions>
    <Face side = "x-">
      <BCType id = "0"   label = "Pressure"     var = "Neumann">
                           <value> 0. </value>
      </BCType>
      <BCType id = "all" label = "Velocity"     var = "Neumann">
                           <value> [0.,0.,0.] </value>
      </BCType>
      <BCType id = "all" label = "Temperature"  var = "Neumann">
                           <value> 0.0 </value>
      </BCType>
      <BCType id = "all" label = "Density"      var = "Neumann">
                           <value> 0.0 </value>
      </BCType>
      <BCType id = "all" label = "SpecificVol"  var = "computeFromDensity">
                           <value> 0.0  </value>
      </BCType>
    </Face>
      .
      [other faces]
      .
  </BoundaryConditions>
</Grid>
```

There is also the field tag `id = "all".` In principal, one could set different boundary condition types for different materials. In practice, this is rarely used, so the usage illustrated here should be used. Note that pressure field id is always 0. Symmetric boundary conditions are set using:

```
<Face side = "y-">
    <BCType id = "all" label = "Symmetric" var = "symmetry"> </BCType>
</Face>
```

In addition to "Dirichlet", "Neumann", and "Symmetric" type boundary conditions ICE has several custom or experimental boundary conditions the user can access. The "Sine" boundary condition was designed to impose a pulsating pressure wave in the boundary cells by applying

$$p = p_{reference} + Asin(\omega t) \tag{8.17}$$

The input file parameters that control the frequency and magnitude of the wave are:

```
<SINE_BC>
  <omega>    1000 </omega>
  <A>        800 </A>
</SINE_BC>
```

and to specify them add

```
<BCType id = "0"   label = "Pressure"     var = "Sine">
                      <value> 0.0 </value>
</BCType>
<BCType id = "0"   label = "Temperature"  var = "Sine">
                      <value> 0.0 </value>
</BCType>
```

to the input file. For non-reflective boundary conditions the user should specify the "LODI" or locally one-dimensional invisid type [14]

```
<LODI>
  <press_infinity> 1.0132500000010138e+05  </press_infinity>
  <sigma>          0.27                    </sigma>
  <ice_material_index> 0                   </ice_material_index>
</LODI>
```

and

```
<Face side = "x+">
  <BCType id = "0"   label = "Pressure"     var = "LODI">
                        <value> 0. </value>
  </BCType>
  <BCType id = "0"   label = "Velocity"     var = "LODI">
                        <value> [0.,0.,0.] </value>
  </BCType>
  <BCType id = "0"   label = "Temperature"  var = "LODI">
                        <value> 0.0 </value>
  </BCType>
  <BCType id = "0"   label = "Density"      var = "LODI">
                        <value> 0.0 </value>
  </BCType>
  <BCType id = '0' label = "SpecificVol" var = "computeFromDensity">
                        <value> 0.0 </value>
  </BCType>
</Face>
```

This boundary condition is designed to suppress all the unwanted effects of an artifical boundary. **This BC is computationally expensive, not entirely effective and should be used with caution**. In flow fields where there are no passing through the outlet of the domain it reduces the reflected pressure waves significantly.

### 8.3.9 Variable Volume Fraction

An arbitrary number of materials may be layered on one another to generate mixtures in a cell. Each geometry object may have a specified fraction of the cell in the range (0,1]. A volume fraction may be specified as:

```
<geom\_object>
   ...
   <volumeFraction> 0.5 </volumeFraction>
</geom\_object>
```

If the volume fraction is spcified for one geometry object it MUST be specified for all geometry objects, even if they constitute a volume fraction equal to 1 in the given volume. Similarly, each cell must have a total of 1 for the summation of all volume fractions of materials in that cell. Failure of either of these criteria will result in a crashed simulation during the problem setup phase.

MPMICE also has the ability to use the variable volume fraction convention, however pure MPM does not.

### 8.3.10 Output Variable Names

There are numerous variables that can be saved during a simulation. The table below is a list of the most commonly saved variables. To see the entire list ICE specfic variables available to the user run

```
inputs/labelNames ice
```

Dimensions are given in mass (M), length (L), time (t) and tempertare (T). Bold face label names signify vectors quantities. The location of the variable on the grid is denoted by (CC) for the cell-centered or (FC) for face-centered. Conserved quantities that are summed over all cells, every timestep, and written to a "dat" file inside of the `uda` directory are denoted with (dat).

| LabelName | | Description |
|---|---|---|
| delP_Dilatate | $M/Lt^2$ | change in pressure during the, (CC). |
| delP_MassX | $M/Lt^2$ | change in pressure due to mass addition, (CC). |
| eng_adv | $ML^2/t^2$ | energy of a material after the advection task, (CC). |
| eng_exch_error | $ML^2/t^2$ | $\sum_{i=1}^{\text{AllCells}}$ Internal Energy After Exchange Process− $\sum_{i=1}^{\text{AllCells}}$ Internal Energy Before Exchange Process, (dat). |
| eng_L_ME_CC | $ML^2/t^2$ | Energy of a material after the exchange task and just before the advection task, (CC). |
| imp_delP | $M/Lt^2$ | (CC). |
| KineticEnergy | $ML^2/t^2$ | $\sum_{i=1}^{\text{AllCells}}(0.5m(\vec{v})^2)_i$, (dat). |
| mach | | Mach number, (CC). |
| mag_div_vel_CC | | Magnitude of the divergence of the velocity, (CC). |
| mag_grad_press_CC | | Magnitude of the gradient of the pressure, (CC). |
| mag_grad_rho_CC | | Magnitude of the gradient of the density, (CC). |
| mag_grad_temp_CC | | Magnitude of the gradient of the temperature, (CC). |
| mag_grad_vol_frac_CC | | Magnitude of the gradient of the volume fraction, (CC). |
| mass_adv | $M$ | Mass of a material after the advection task, (CC). |
| mass_L_CC | $M$ | Mass of a material just before the advection task, (CC). |
| modelEng_src | $ML^2/t^2$ | Energy source term, computed from a reaction model, (CC). |
| modelMass_src | $M$ | Mass source term, computed from a reaction model, (CC). |
| **modelMom_src** | $ML/t$ | Momentum source term, computed from a reaction model, (CC). |
| modelVol_src | | Volume source term, computed from a reaction model, (CC). |
| **mom_exch_error** | $ML/t$ | $\sum_{i=1}^{\text{AllCells}}$ Momemtum After Exchange Process− $\sum_{i=1}^{\text{AllCells}}$ Momentum Before Exchange Process, (dat). |
| **mom_L_CC** | $ML/t$ | Momemtum before momentum exchange task, (CC). |
| **mom_L_ME_CC** | $ML/t$ | Momentum after momentum exchange task, (CC). |
| **mom_source_CC** | $ML/t$ | All sources of momentum,(CC). |
| press_CC | $M/Lt^2$ | Pressure $P = P_{\text{equilibration}} + \Delta P$, (CC). |
| press_equil_CC | $M/Lt^2$ | Pressure after the compute equilibration task, (CC). |
| pressX_FC | $M/Lt^2$ | Pressure on the $x^{-,+}$ cell faces, (FC). |
| pressY_FC | $M/Lt^2$ | Pressure on the $y^{-,+}$ cell faces, (FC). |
| pressZ_FC | $M/Lt^2$ | Pressure on the $z^{-,+}$ cell faces, (FC). |
| rho_CC | $M/L^3$ | Density of each material, (CC). |
| specific_heat | $L^2/t^2T$ | Constant Specific Heat, (CC). |
| speedSound_CC | $L/t$ | Speed of sound of each material, (CC). |
| sp_vol_adv | | |
| sp_vol_CC | $L3/M$ | Specific volume of each material, (CC). |
| temp_CC | $T$ | Temperature of each material, (CC). |
| TempX_FC | $T$ | temperature on the $x^{-,+}$ cell faces, (FC). |
| TempY_FC | $T$ | temperature on the $y^{-,+}$ cell faces, (FC). |
| TempZ_FC | $T$ | temperature on the $z^{-,+}$ cell faces, (FC). |
| thermalCond | $ML/t^3T$ | Thermal conductivity, (CC). |
| TotalIntEng | $ML^2/t^2$ | $\sum_{i=1}^{\text{AllCells}}(mc_vT)_i$, (dat). |
| TotalMass | $M$ | $\sum_{i=1}^{\text{AllCells}} m_i$, (dat). |
| TotalMomentum | $ML/t$ | $\sum_{i=1}^{\text{AllCells}}(m\vec{v})_i$, (dat). |
| uvel_FC | $L/t$ | x-component of velocity, before momentum exchange, (FC). |
| uvel_FCME | $L/t$ | x-component of velocity, after momentum exchange task, (FC). |
| **vel_CC** | $L/t$ | Velocity at the end of a timestep, (CC). |
| viscosity | $M/Lt$ | Dynamic viscosity, (CC). |
| vol_frac_CC | | Volume fraction of each material, (CC). |
| vol_fracX_FC | | Volume fraction on the $x^{-,+}$ cell faces, (FC). |
| vol_fracY_FC | | Volume fraction on the $y^{-,+}$ cell faces, (FC). |
| vol_fracZ_FC | | Volume fraction on the $z^{-,+}$ cell faces, (FC). |
| vvel_FC | $L/t$ | y-component of velocity, before momentum exchange task, (FC). |
| vvel_FCME | $L/t$ | y-component of velocity, after momentum exchange task, (FC). |
| wvel_FC | $L/t$ | z-component of velocity, before momentum exchange, (FC). |
| wvel_FCME | $L/t$ | z-component of velocity, after momentum exchange task, (FC). |

The variables, `mag_div_vel_CC, mag_grad_press_CC, mag_grad_rho_CC, mag_grad_temp_CC, mag_grad_vol_` are the magnitude of the gradient or divergence of the respective primative variable. If the user visual To are large and based on this information the adaptive mesh cell refinement criteria can be set.

Below is a list of the XML tags pertaining specifically to ICE problems.

### 8.3.11 XML tag description

| XML tag | Type | Dimensions | Description |
|---|---|---|---|
| cfl | double | | Courant Number. |
| applyHydroStaticPressure | bool | | on/off switch for applying hydrostatic pressure correction. Default is true. |
| gravity | Vector | $[L/t^2]$ | gravitational acceleration, $\vec{g}$. |
| global material properties | | | |
| dynamic_viscosity | double | $[M/Lt]$ | viscosity, $\mu$. |
| thermal_conditucivity | double | $[ML/t^3T]$ | thermal conductivity, $k$ |
| specific_heat | double | $[L^2/t^2T]$ | $c_p$ |
| gamma | double | | ratio of specific heats, $\gamma$. |
| geometry object related | | | |
| res | vector | | resolution used for defining geometry objects. |
| velocity | vector | $[L/t]$ | initial velocity, $\vec{u}$. |
| density | double | $[M/L^3]$ | initial density, $\rho$. |
| temperature | double | $[T]$ | initial temperature, $T$. |
| pressure | double | | Not used. |
| AMR Parameters | | | |
| orderOfInterpolation | integer | | Order of interpolation at the coarse/fine interfaces. |
| do_Refluxing | boolean | | on/off switch for correcting the flux of mass, momentum, and energy at the course/fine interfaces. |

## 8.4 Examples

Below are several example problems that illustrate the wide range of problems that can be solved using the ICE algorithm. Where possible simulation results are compared to exact solutions or high fidelity numerical results. Note in order to run the post processing scripts the user should have a recent version of Octave installed. To visualize the results the visualization package VisIT should be used. VisIT session files are included.

### Poiseuille Flow

#### Problem Description

The Poiseuille flow problem is classical viscous flow problem in which flow is driven through two parallel plates from fixed pressure gradient. The pressure gradient is balanced by the diffusion $x$ momentum in the $y$ direction.

#### Simulation Specifics

**Component used:** ICE

**Input file name:** `CouettePoiseuille.ups`

Edit this file and set the boundary condition for the velocity on the y+ = 0.0. Change:

```
<BCType id = "0"   label = "Velocity"     var = "Dirichlet">
                       <value> [1.25,0.,0.] </value>
[to]
<BCType id = "0"   label = "Velocity"     var = "Dirichlet">
                      <value> [0,0.,0.] </value>
```

**Command used to run input file:**

```
mpirun -np 1 sus -solver hypre inputs/UintahRelease/ICE/CouettePoiseuille.ups
```

**Postprocessing command:**

`inputs/UintahRelease/ICE/compare_CouettePoiseuille.m -uda Couette-Poiseuille.uda`

You must edit `compare_CouettePoiseuille.m` and set `wallVel = 0`. This will generate a postscript file CouettePoiseuille.ps

**Simulation Domain:** 1 x .01 x .01 m

**Cell Spacing:**

10 x 5 x 10 mm (Level 0)

**Example Runtimes:**

8ish minutes (1 processor, 2.66 GHz Xeon)

**Physical time simulated:** 15 sec.

#### Results

Figure 8.1 shows a comparison of the exact and simulated u velocity at time $t = 15sec$, 5 cells from the end of the domain. The lower plot shows the difference of the velocity $\|u - u_{exact}\|$.

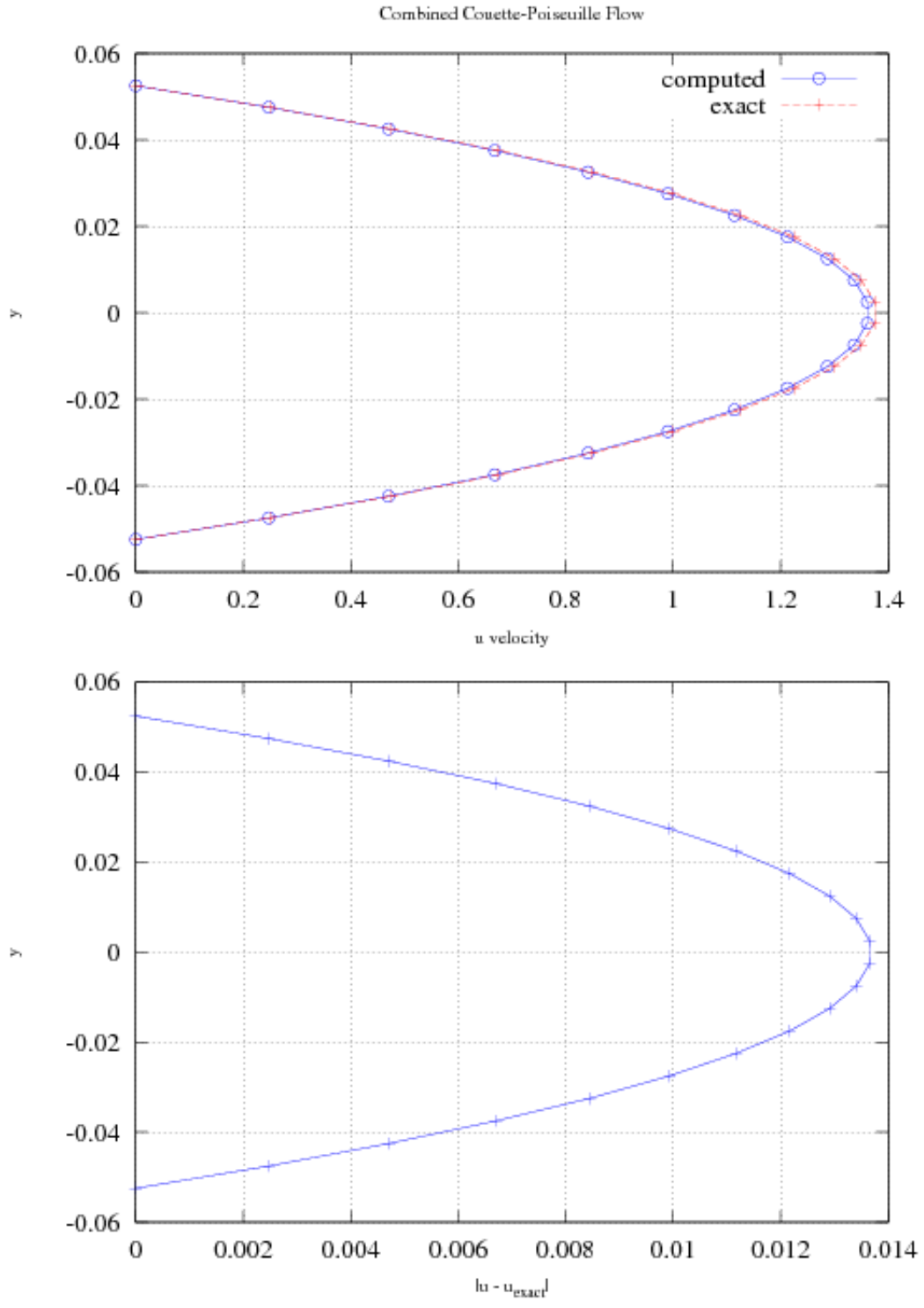



Figure 8.1: Comparison of $u$ velocity $t = 15sec$

# Combined Couette-Poiseuille Flow

## Problem Description

The combined Couette-Poiseuille flow problem is another classical viscous flow problem in which flow is driven through a channel by a pressure gradient and a wall moving. The reduced x momentum equation differential is

$$\mu \frac{d^2u}{dy^2} = \frac{dp}{dx} = \text{constant} \tag{8.18}$$

subject to the no slip boundary condition $u(\pm h)$ = wall velocity, where $h$ is half the height of the channel [19].

## Simulation Specifics

**Component used:** ICE

**Input file name:** `CouettePoiseuille.ups`

**Command used to run input file:**

```
mpirun -np 1 sus -solver hypre inputs/UintahRelease/ICE/CouettePoiseuille.ups
```

**Postprocessing command:**

`inputs/UintahRelease/ICE/compare_CouettePoiseuille.m -uda Couette-Poiseuille.uda`
This Octave script will generate a postscript file CouettePoiseuille.ps

**Simulation Domain:** 1 x .01 x .01 m

**Cell Spacing:**
10 x 5 x 10 mm (Level 0)

**Example Runtimes:**
8ish minutes (1 processor, 2.66 GHz Xeon)

**Physical time simulated:** 15 sec.

## Results

Figure 8.2 shows a comparison of the exact and simulated u velocity at time $t = 15sec$, 5 cells from the end of the domain. The lower plot shows the difference of the velocity $\|u - u_{exact}\|$.

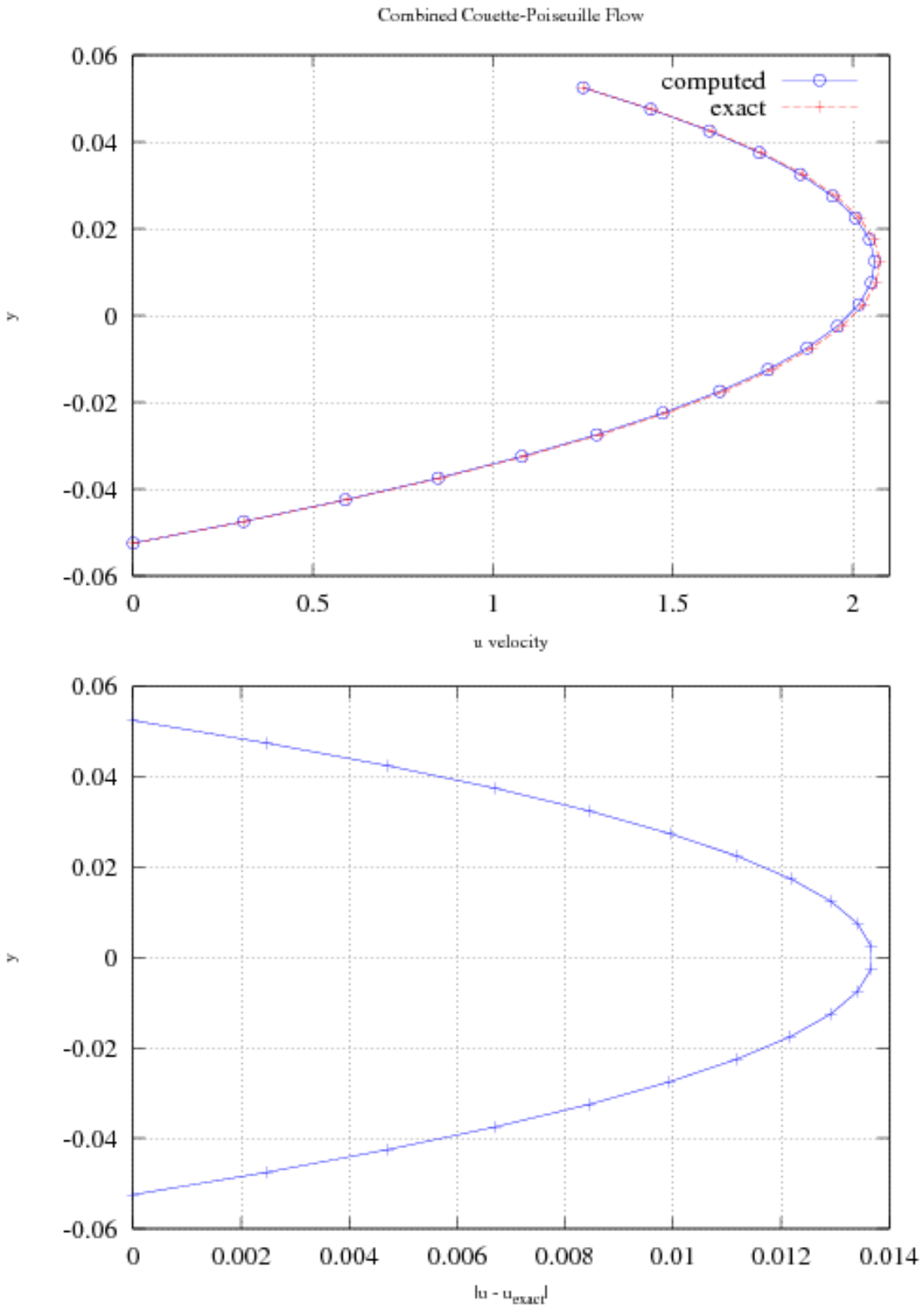



Figure 8.2: Comparison of $u$ velocity $t = 15sec$

# Shock Tube

## Problem Description

The shock tube problem is a standard 1D compressible flow problem that has been used by many as a validation test case [9, 13, 17]. At time $t = 0$ the computational domain is divided into two separate regions of space by a diaphram, with each region at a different density and pressure. The separated regions are at rest with a uniform temperature $= 300K$. The initial pressure ratio is $\frac{P_R}{P_L} = 10$ and density ratio is $\frac{\rho_R}{\rho_L} = 0.1$ The diaphram is instantly removed and a traveling shockwave, discontinutity and expansion fan form. The expansion fan moves towards the left while the shockwave and contact discontinutity move to the right. This problem tests the algorithm's ability to capture steep gradients and solve Eulers equations.

## Simulation Specifics

**Component used:** ICE

**Input file name:** `rieman_sm.ups`

**Command used to run input file:** `sus inputs/UintahRelease/ICE/shockTube.ups`

**Postprocessing command:**
`scripts/ICE/plot_shockTube_1L shockTube.uda y`
This Octave script will generate a postscript file shockTube.ps

**Simulation Domain:** 1 x .001 x .001 m

**Cell Spacing:**
1 x 1 x 1 mm (Level 0)

**Example Runtimes:**
1 minute (1 processor, 2.66 GHz Xeon)

**Physical time simulated:** 0.005 sec.

## Results

Figure 8.3 shows a comparison of the exact versus simulated results at time $t = 5msec$.

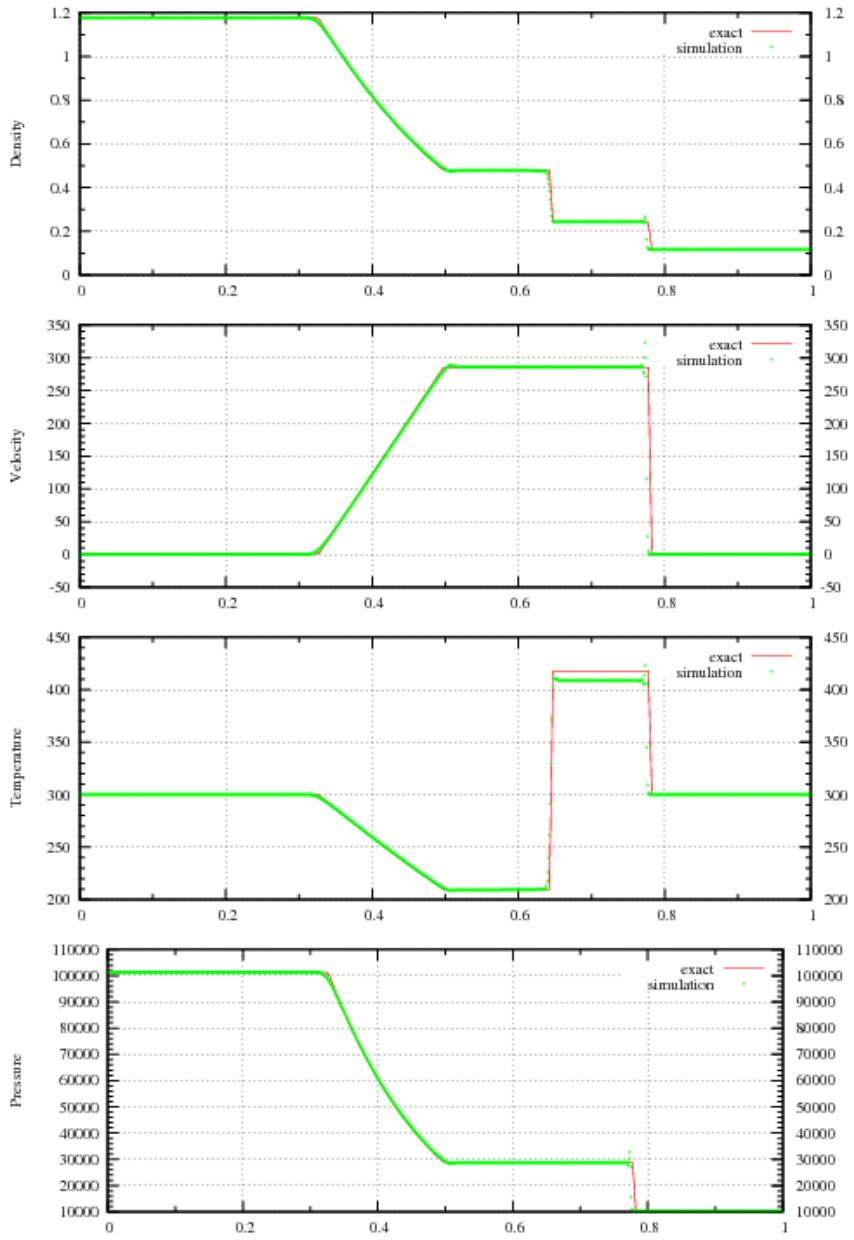



Figure 8.3: Shock tube results at time $t = 5msec$

# Shock Tube with Adaptive Mesh Refinement

## Simulation Specifics

**Component used:** ICE

**Input file name:** `shockTube_AMR.ups`

**Command used to run input file:**

`sus inputs/UintahRelease/ICE/shockTube_AMR.ups`

**Postprocessing command:**

`../../src/scripts/ICE/plot_shockTube_AMR shockTube_AMR.uda y`

This Octave script will generate a postscript file shockTube_AMR.ps

**Simulation Domain:** 1 x .001 x .001 m

**Cell Spacing:**

10 x 1 x 1 mm (Level 0)
2.5 x 1 x1 mm (Level 1)
0.625 x1 x1 mm (Level 2)

**Example Runtimes:**

2ish minutes (1 processor, 2.66 GHz Xeon)

**Physical time simulated:** 0.0005 sec.

## Results

Figure 8.4 shows a comparison of the exact versus simulated results at time $t = 5msec$.

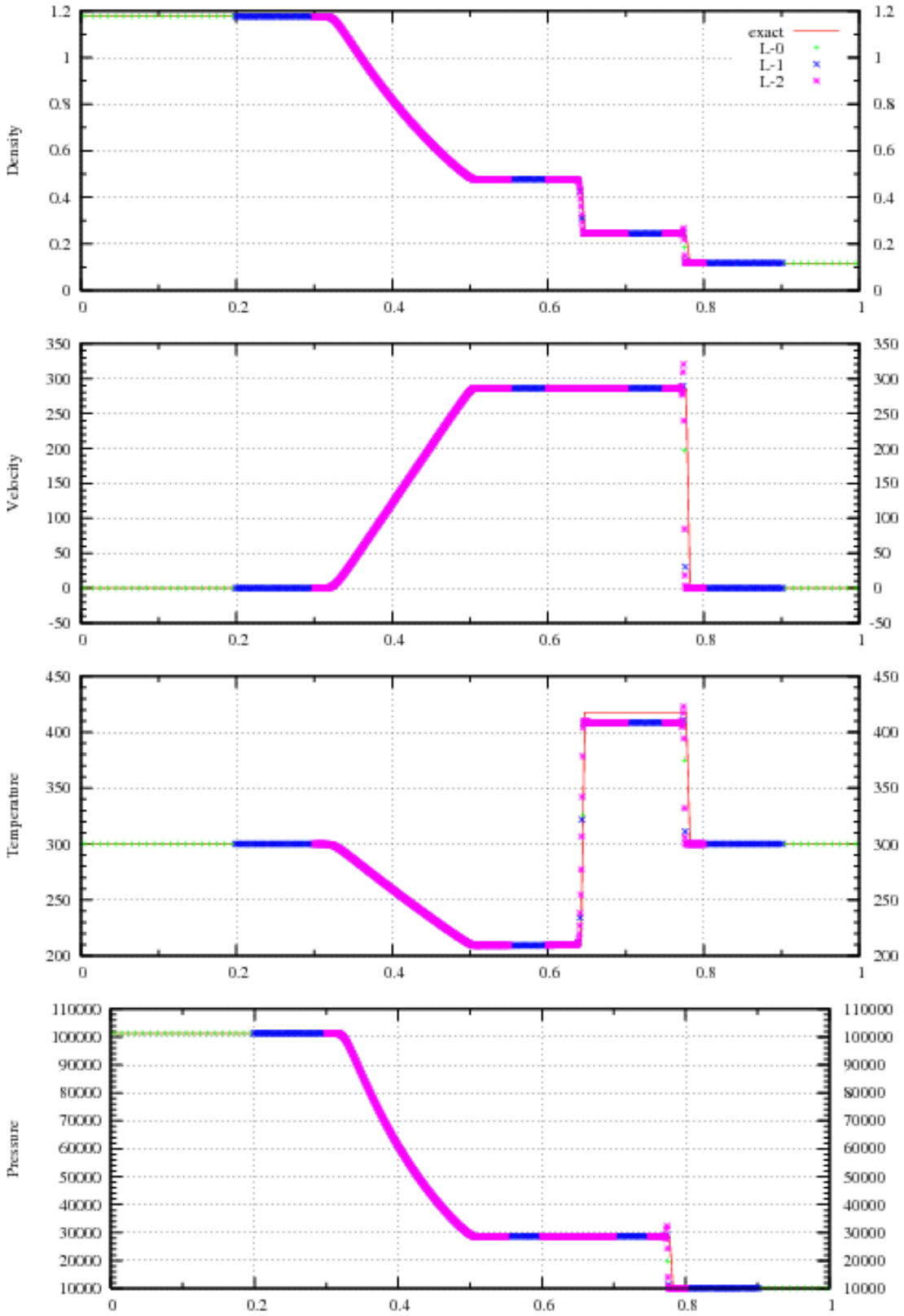



Figure 8.4: Shock tube results at time $t = 5msec$

# 2D Riemann Problem with Adaptive Mesh Refinement

## Problem Description

In two-dimensional Riemann problems there 15 different solutions that combine rarefaction waves, shock waves and a slip line or contact discontinuities [2, 10]. Here we simulate 4 slip lines that form a symmetrical single vortex turning counter clockwise. At time $t = 0$ the computational domain is divided into four quadrants by the lines $x = 1/2, y = 1/2$ The initial condition for $V = (p, \rho, u, v)$ in the four quadrants are $V_{ll} = (1, 1, -0.75, 0.5), V_{lr} = (1, 3, -0.75, -0.5), V_{ul} = (1, 2, 0.75, 0.5), V_{ur} = (1, 1, 0.75, -0.5)$ where, $p$ is pressure, $\rho$ is the density of the polytropic gas, $u$ and $v$ are the $x$ and $y$ component of velocity.

## Simulation Specifics

**Component used:** ICE

**Input file name:** `riemann2D_AMR.ups`

**Command used to run input file:**

```
mpirun -np 5 sus inputs/UintahRelease/ICE/riemann2D_AMR.ups
```

**VisIT session file:** `inputs/UintahRelease/ICE/riemann2D.session`

**Simulation Domain:** 0.96 x 0.96m x 0.1 m

**Cell Spacing:**

40 x 40 x 1 mm (Level 0)
10 x 10 x 1 mm (Level 1)
2.5 x 2.5 x 1 mm (Level 2)

**Example Runtimes:**

5ish minutes (5 processors, 2.66 GHz Xeon)

**Physical time simulated:** 0.3 sec.

## Results

Figure 8.5 shows a flood and line contour plot(s) of the density of the gas at 0.03sec.

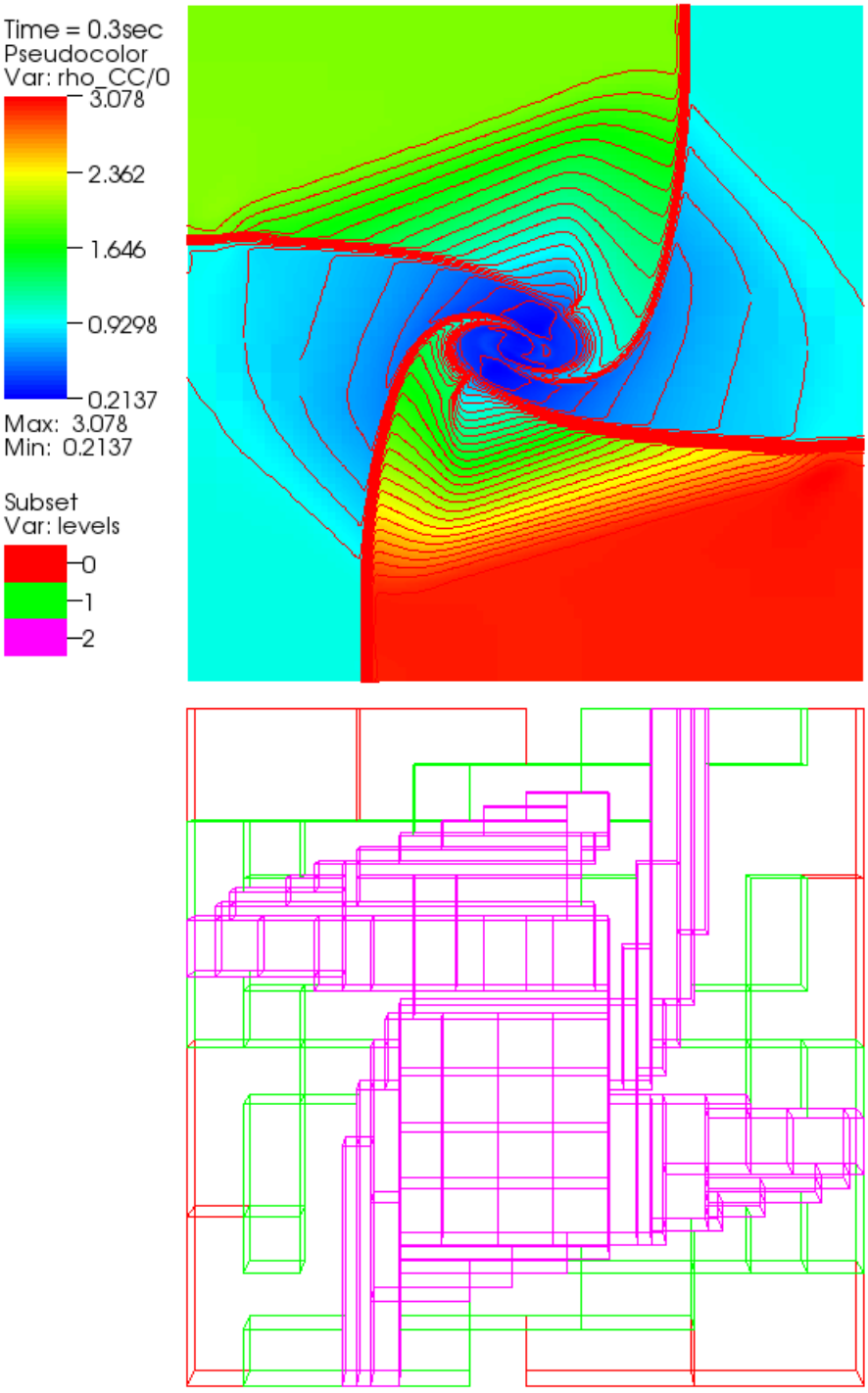



Figure 8.5: Contour plot of density for the 2D Riemann problem at time $t = 0.3sec$. Bottom plot shows the outline of the patches on the 3 levels.

# Explosion 2D

## Problem Description

For the multidimensional blast wave orexplosion test is a standard compressible flow problem that has been used by many as a validation test case. At time $t = 0$ there is a circular region of gas at the center of the domain at a relatively high pressure and density. The expansion of high pressure gas forms a circular shock wave and contact surface that expands into surrounding atmosphere. At the same time a circular rarefaction travels towards the origin. As the shock wave and contact surface move outwards they become weaker and at some point the contact reverses direction and travels inward. The rarefraction reflects from the center and forms an overexpanded region, creating a shock that travels inward [17]. At time $t = 0$ the computational domain is divided into two region, circular high pressure region with a radius $R = 0.4$ and the surrounding box 2x2x0.1. The initial condition inside of the circular region were $(p = 1, \rho = 1, u = 0, v = 0)$ and outside $(p = 0.1, \rho = 0.125, u = 0, v = 0)$. The fluid was an ideal, inviscid, polytropic gas.

## Simulation Specifics

**Component used:** ICE

**Input file name:** `explosion.ups`

**Command used to run input file:**

```
mpirun -np 4 sus inputs/UintahRelease/ICE/explosion.ups
```

**Visualization net file:** `inputs/UintahRelease/ICE/Explosion.session`

**Postprocessing command:**

```
scripts/ICE/plot_explosion_AMR Explosion_AMR.uda y
```

This Octave script will generate a postscript file explosion_AMR.ps

**Simulation Domain:** 2 x 2 x .1

**Cell Spacing:**

62.5 x 62.5 x 10 (Level 0)
15.625 x 15.625 x 10 (Level 1)
3.9 x 3.9 x 10 (Level 2)

**Example Runtimes:**

20 minutes (4 processor, 2.66 GHz Xeon)

**Physical time simulated:** 0.25 (non-dimensional).

## Results

Figures 8.6 and 8.7 shows surface plots of the pressure and density at $t = 0.25$. Since this test is symetrical we can use results from the equivalent 1 dimensional problem to compare against

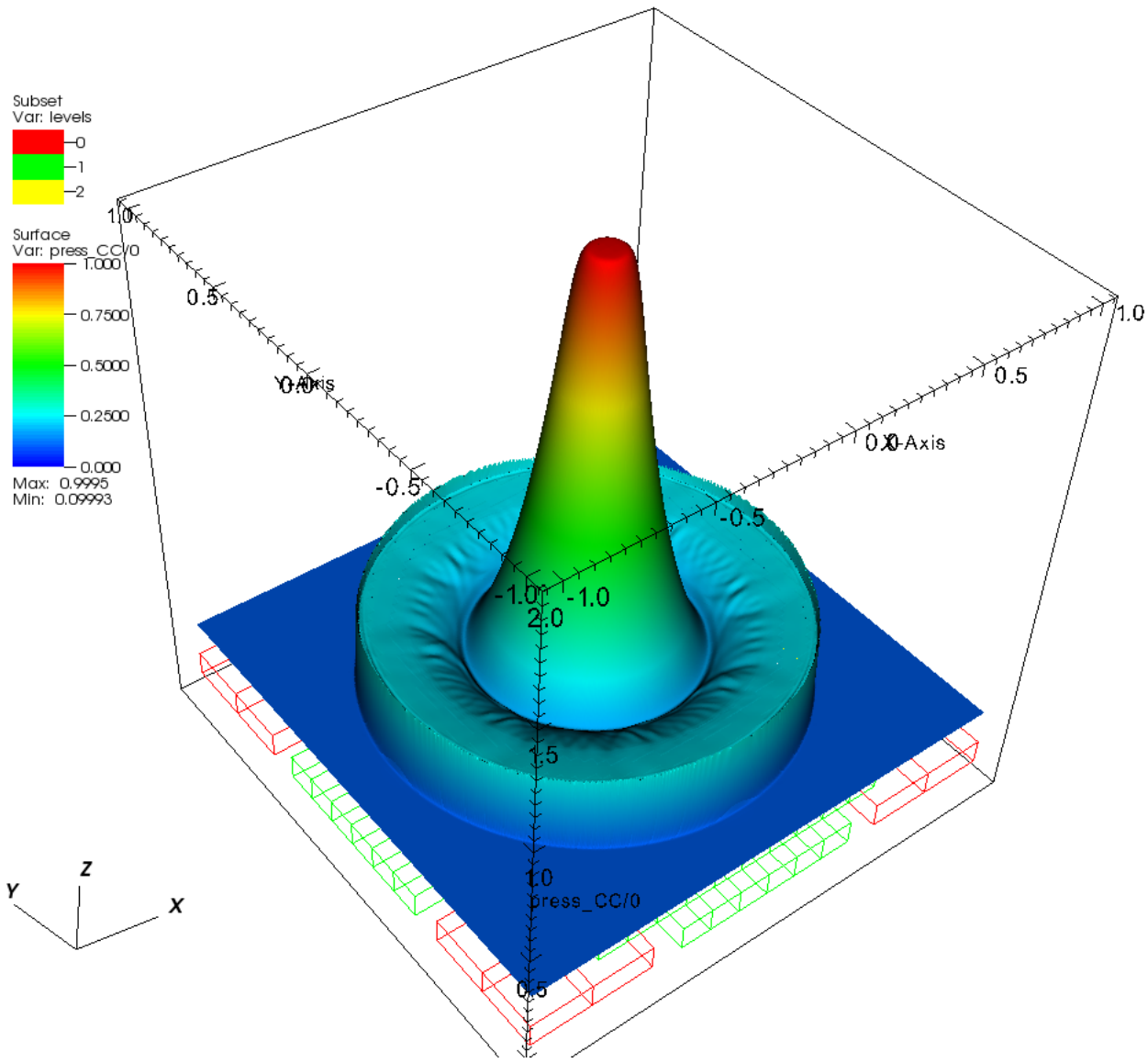



Figure 8.6: Pressure field at $t = 0.25$

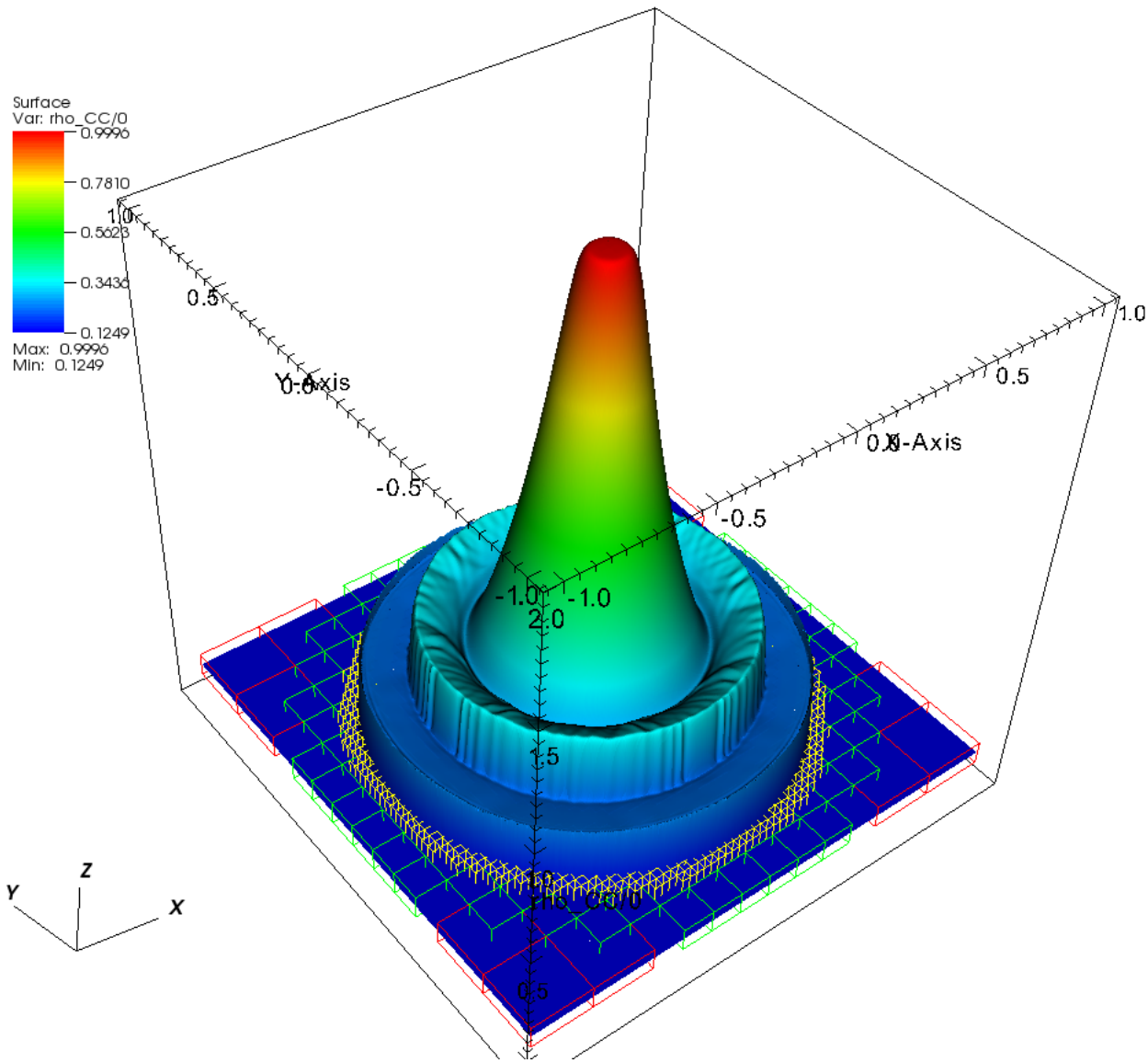



Figure 8.7: Density field at time $t = 0.25$

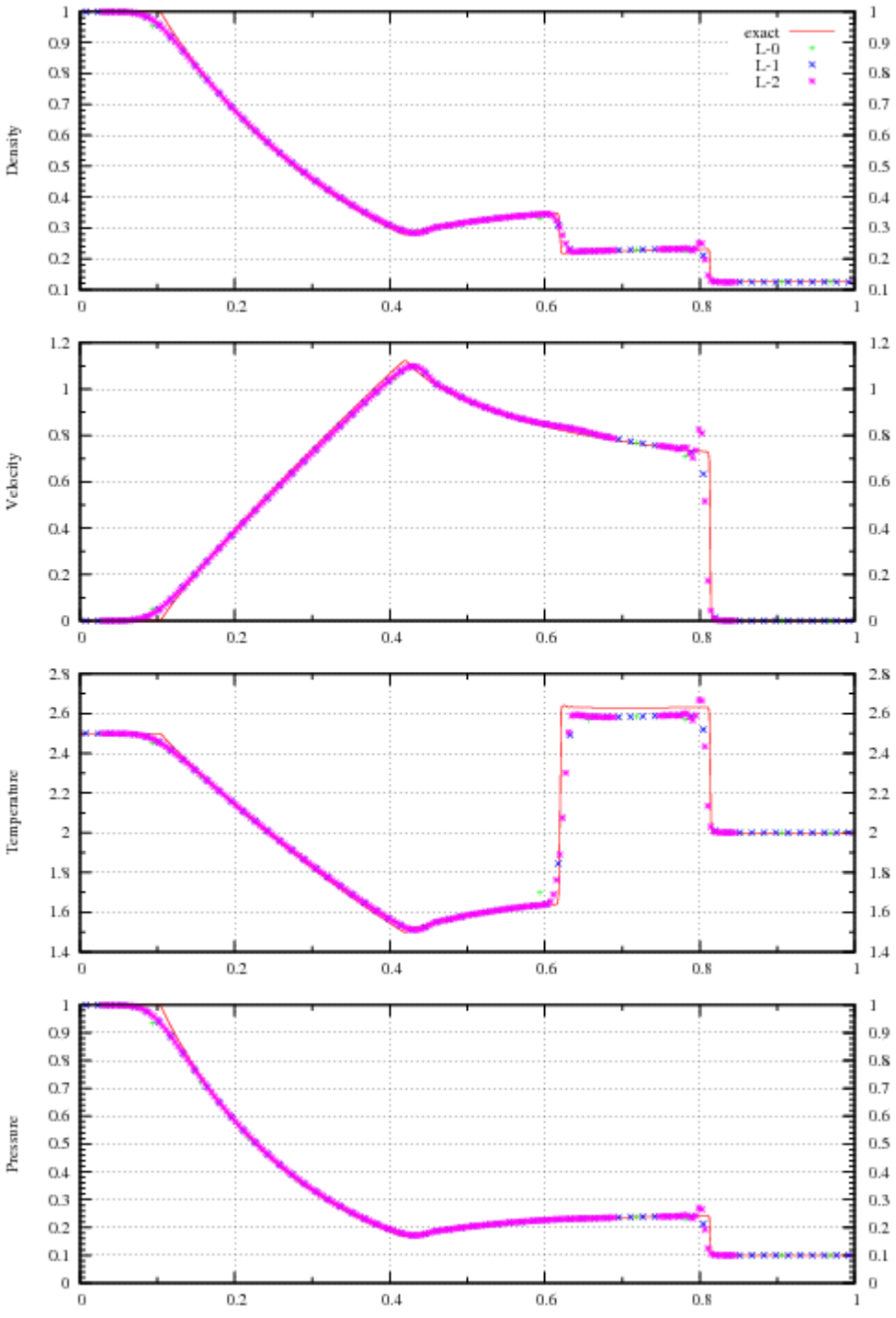



Figure 8.8: $t = 0.25$

# ANFO Rate Stick

## Problem Description

A cylindrical stick ($r = 8mm$) of Ammonium Nitrate Fuel Oil (ANFO) given an initial velocity of $90m/s$. As it strikes the domain boundary, pressure is generated sufficient to reach the initial pressure required to activate the JWL++ [11] detonation model. This empirically based model results in a steady state detonation that traverses the stick, consuming the solid explosive and generating high pressure gas. The experimentally observed curvature is generated at the detonation front, a feature that will not develop in programmed burn models. By running this simulation at a variety of cylinder radii, one can observe the "size effect", namely that cylinders of larger radii will reach a higher steady state detonation velocity, due to the increased effective confinement. An infinite radius case can be simulated by shrinking the computational domain to one cell in each of the transverse directions.

## Simulation Specifics

**Component used:** ICE

**Input file name:** `JWLpp8mmRS.ups`

**Command used to run input file:**

```
mpirun -np 4 sus inputs/UintahRelease/ICE/JWLpp8mmRS.ups
```

**Visualization net file:** `inputs/UintahRelease/ICE/RateStick.session`

**Simulation Domain:** 0.1 m x 0.015 m x 0.015 m

**Cell Spacing:**
0.0005 x 0.0005 x 0.0005 (Level 0)

**Example Runtimes:**
1.5 hours (4 processor, 3.16 GHz Xeon)

**Physical time simulated:** 20.0 $\mu$seconds

## Results

Figure 8.9 shows a volume rendering of the density of the reactant. Note the curvature of the reaction zone.

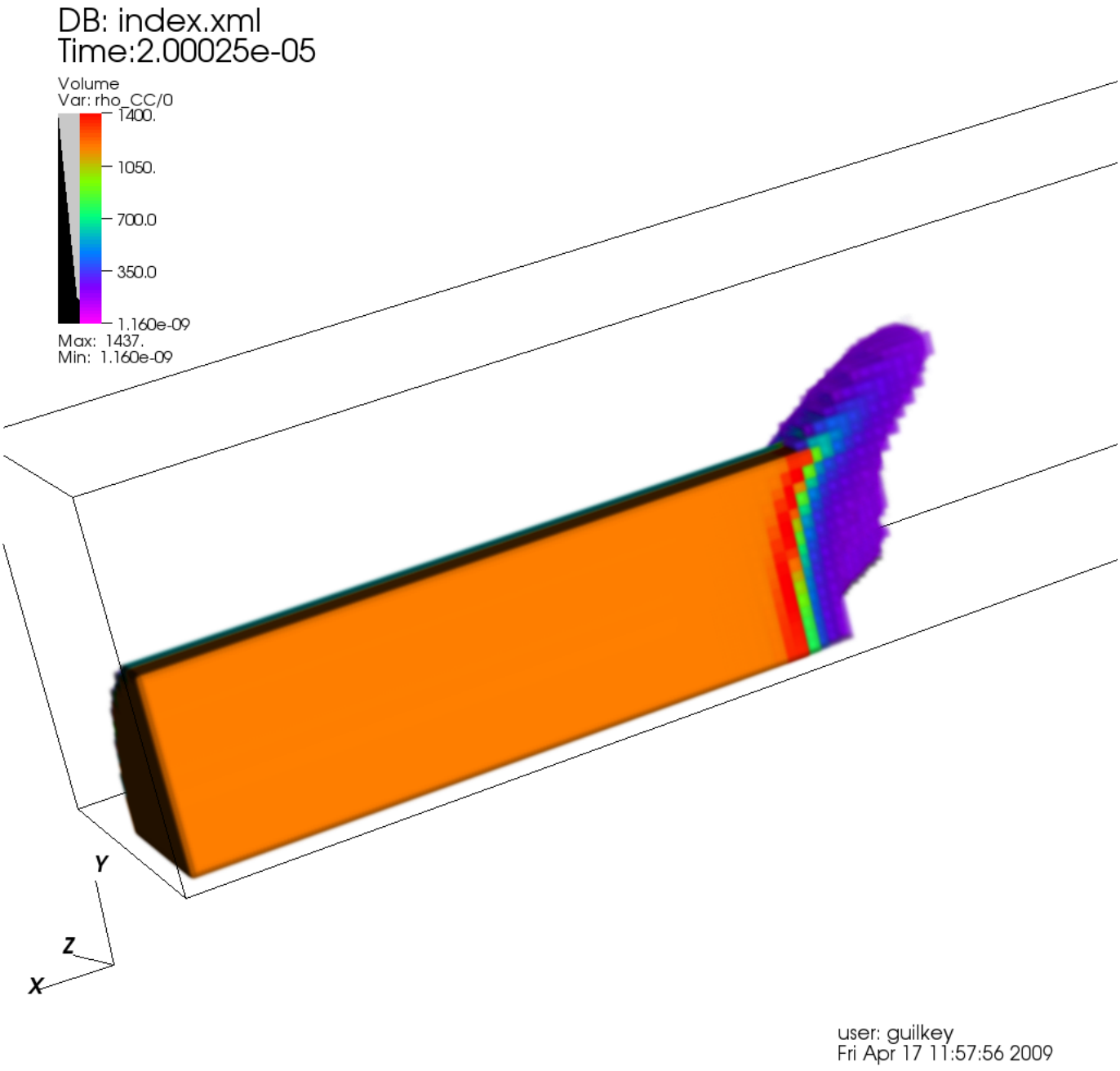



Figure 8.9: Density of reactant material.

## 8.5 Verification and Validation

This section contains the verification and validation tests that have been performed on ICE. The below problems are run at a variety of resolutions on multiple planes, and compared to either exact solutions, or experimental data. Along with comparison plots the L2 norm is calculated to gauge the order of accuracy for many of these problems, allowing us a concise way to analyze these results. Many of these problems run with the order of accuracy framework described in 5.

### Rayleigh Problem

#### Problem Description

The Rayleigh Problem is one of the few simple test cases to have an exact solution, which proves extremely usefull for verification. In this 2-D problem, a flat plate initial at rest is surrounded by a fluid. At t=0 the plate is instantaneously moved to a constant speed and the resulting velocity profile is observed.

#### Simulation Specifics

**Component used:** ICE

**Order of Accuracy Framework:** Yes

**Tests to run:**

| Name | Description | Varied Parameter |
|---|---|---|
| rayleigh_dx.tst | Rayleigh problem on the x-z plane | Resolution - 25 50 100 200 400 |
| rayleigh_dy.tst | Rayleigh problem on the x-y plane | Resolution - 25 50 100 200 400 |
| rayleigh_dz.tst | Rayleigh problem on the y-z plane | Resolution - 25 50 100 200 400 |

**Associated input files:**

rayleigh_dx.ups
rayleigh_dy.ups
rayleigh_dz.ups

**Postprocessing script:**

`compare_Rayleigh.m`

**Simulation Domain:** .01 x .01 x .01 m

**Example Runtimes:**

<5 minutes (1 processor, 2.66 GHz Xeon)

**Exact solution or data compared with:**

Exact unsteady solution to the rayleigh problem

Incompressible Flow, by Panton pg 177 [12]

$U = U_0 * (1 - erf(\frac{y}{2*\sqrt{\nu*t}}))$

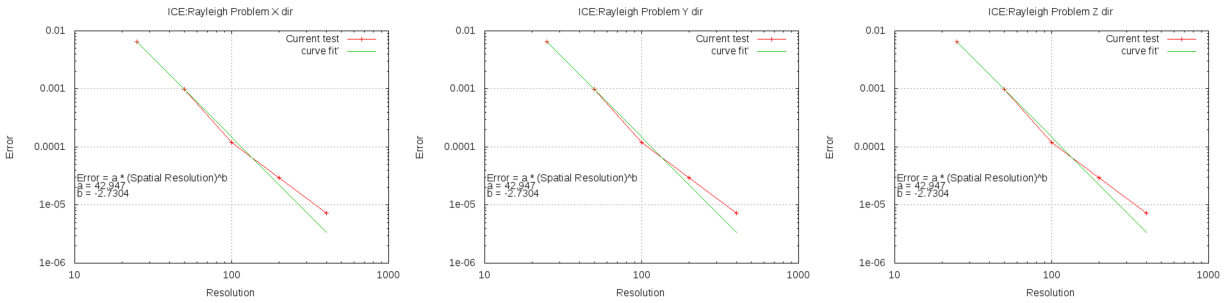



Figure 8.10: L2 norm as a function of resolution

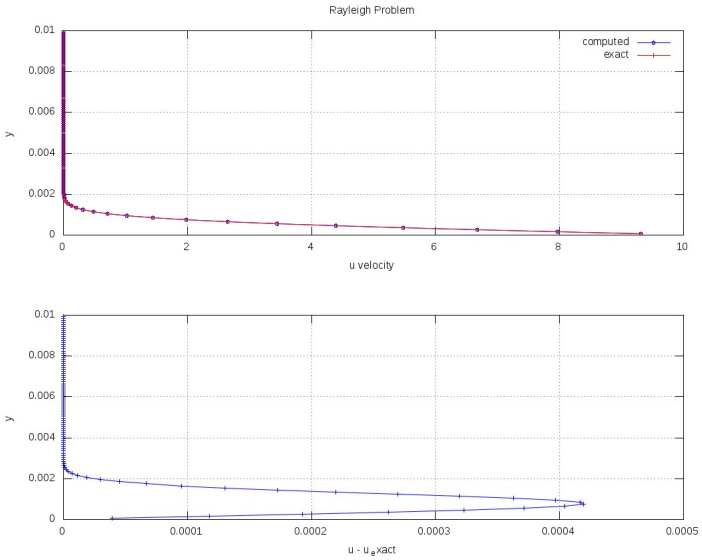



Figure 8.11: $u$ component of velocity versus height $y$

## Results

Figure 8.10 shows the order of accuracy of the Rayleigh Problem. This figure shows the L2 norm plotted against the resolution in the principal direction in all the planes.
Figure 8.11 shows the horizontal velocity $u$ plotted against height $y$ for both the exact solution and the computed solution.

# 2-D Lid-Driven Cavity Flow

## Problem Description

2-D lid driven cavity flow is another one of the simple test cases available. In this problem a square box containing a fluid is driven from a lid at an initial velocity. This lid creates vorticies, and a velocity profile we can compare against. We can also compare the distinguishing features of the vorticies, for a less analytical analysis.

## Simulation Specifics

**Component used:** ICE

**Order of Accuracy Framework:** Yes

**Tests to run:**

| Name | Description | Varied Parameter |
|---|---|---|
| cavityFlow_dx.tst | Cavity Flow with a resolution of 128x128 on the x-z plane | Reynolds Number - 100 400 1000 3200 5000 7500 10000 |
| cavityFlow_dy.tst | Cavity Flow with a resolution of 128x128 on the x-y plane | Reynolds Number - 100 400 1000 3200 5000 7500 10000 |
| cavityFlow_dz.tst | Cavity Flow with a resolution of 128x128 the y-z plane | Reynolds Number - 100 400 1000 3200 5000 7500 10000 |
| cavityFlow_res.tst | Cavity Flow with a Reynolds Number of 5000 on the x-y plane | Resolution - 16 32 64 128 256 |

**Associated input files:**

cavityFlow_dx.ups
cavityFlow_dy.ups
cavityFlow_dz.ups

**Postprocessing script:**

`compare_cavityFlow.m`

**Simulation Domain:** .1 x .1 x .01 m

**Example Runtimes:**

3 hours per test (6 processors, 2.66 GHz Xeon)

**Exact solution or data compared with:**

This test case has neither an exact solution, nor good experimental data. Ghia et al. -1982[18] has become the golden standard when looking to compare results for this specific case. The data located in the compare script comes from the tables located in this paper.

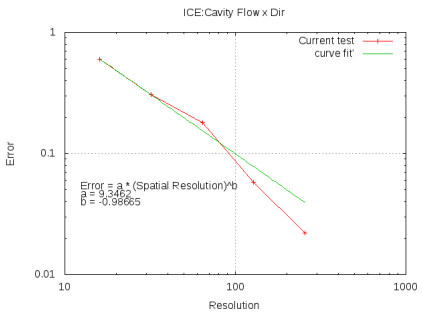



Figure 8.12: Order of Accuracy for a Reynolds Number of 5000

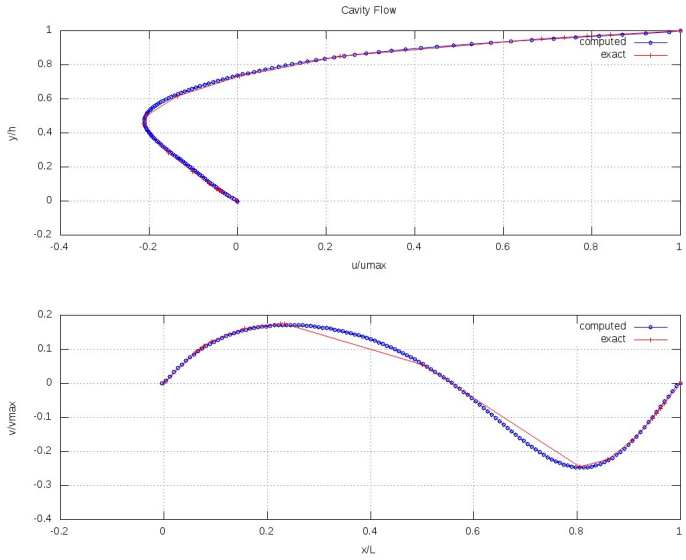



Figure 8.13: Nondimensional velocities versus nondimensional lengths in the x-z plane

## Results

Figure 8.12 shows the order of accuracy for the cavity flow problem at a reynolds number of 5000 in the x-y plane. Figures 8.13,8.14, and 8.13 show the nondimensional $u$ and $v$ components of velocity versus the nondimensional length $L$ in the perpendicular direction for all three planes. Each of these measurements are taken at the centerline in the corresponding direction of the velocity.

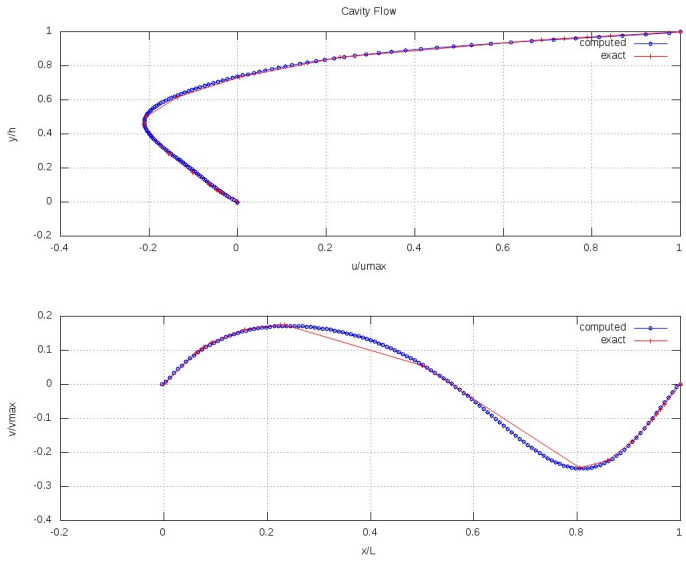



Figure 8.14: Nondimensional velocities versus nondimensional lengths in the x-y plane

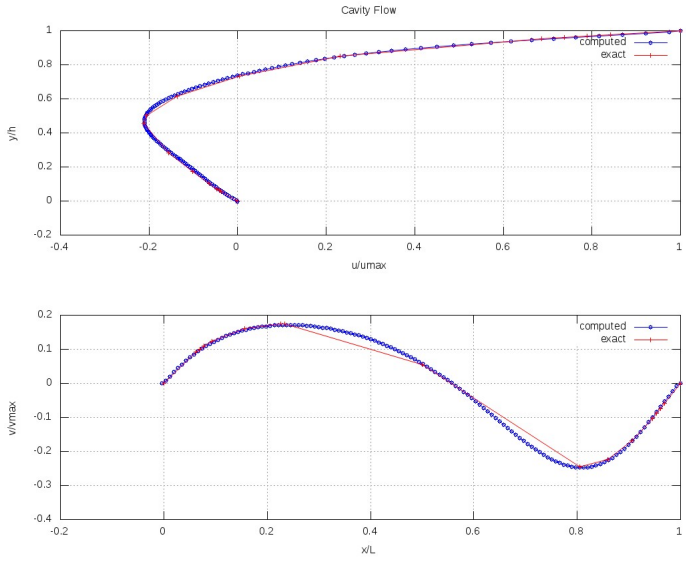



Figure 8.15: Nondimensional velocities versus nondimensional lengths in the y-z plane

# Bibliography


[1] A. Bejan. *Advanced Engineering Thermodynamics*. John Wiley and Sons, USA, 1988.

[2] J. P. Collins C. W. Schulz-Rinne and H. M. Glaz. Numerical solution of the riemann problem for two-dimensional gas dynamics. *J. Comput. Phys*, 14:1394–1414, 1993.

[3] R.D. Falgout, J.E. Jones, and U.M. Yang. The design and implementation of hypre, a library of parallel high performance preconditioners. In A.M. Bruaset and A. Tveito, editors, *Numerical Solution of Partial Differential Equations on Parallel Computers*, Lecture Notes in Computational Science and Engineering. Springer-Verlag, 2006.

[4] G. R. Gathers. *Selected Topics in Shock Wave Physics and Equation of State Modeling*. World Scientific, River Edge, NJ, 1994.

[5] F.H. Harlow and A.A. Amsden. Numerical calculation of almost incompressible flow. *J. Comp. Phys.*, 3:80–93, 1968.

[6] B.A. Kashiwa. A multifield model and method for fluid-structure interaction dynamics. Technical Report LA-UR-01-1136, Los Alamos National Laboratory, Los Alamos, 2001.

[7] B.A. Kashiwa and R.M. Rauenzahn. A cell-centered ICE method for multiphase flow simulations. Technical Report LA-UR-93-3922, Los Alamos National Laboratory, Los Alamos, 1994.

[8] B.A. Kashiwa and R.M. Rauenzahn. A multimaterial formalism. Technical Report LA-UR-94-771, Los Alamos National Laboratory, Los Alamos, 1994.

[9] C. B. Laney. *Computational Gasdynamics*. Cambridge University Press, Cambridge, 1998.

[10] R. Liska and B. Wendroff. Comparison of several difference schemes on 1d and 2d test problems for the euler equations. *J. Comput. Phys*, 25:995–1017, 2003.

[11] P. C. Souers S. Anderson J. Mercer E. McGuire and P. Vitello. Jwl++: A simple reactive flow code package for detonation. *Propellants, Explosives, Pyrotechnics*, 25:54–58, 2000.

[12] Ronald L. Panton. *Incompressible Flow*. Wily-Interscience, 1984.

[13] G. A. Sod. A survey of several finite difference methods for systems of nonlinear hyperbolic conservation laws. *J. Comput. Phys*, 27:1–31, 1978.

[14] J. C. Sutherland and Kenndey C.A. Improved boundary conditions for viscous, reacting compressible flows. *Journal of Computational Physics*, 191:502–524, 2003.

[15] P. A. Thompson. *Compressible-Fluid Dynamics*. McGraw-Hill, New York, 1988.

[16] J. S. Thomsen and T. J. Hartka. Strange carnot cycles; thermodynamics fo a system with a density extremum. *Am. J. Phys.*, 30:26–33, 1962.

[17] E. F. Toro. *Riemann Solvers and Numerical Methods for Fluid Dynamics*. Springer, Berlin Heidelberg, 1997.

[18] K. N. Ghia U. Ghia and C. T. Shin. High-re solutions for incompressible flow using the navier-stokes equations and a multigrid method. *Journal of Computational Physics 48*, pages 387–411, 1982.

[19] F. M. White. *Viscous Fluid Flow, 2nd Edition*. McGraw-Hill, 1991.

# Chapter 9

# MPM

## 9.1 Introduction

The material point method (MPM) was described by Sulsky et al. [55, 57] as an extension to the FLIP (Fluid-Implicit Particle) method of Brackbill [11], which itself is an extension of the particle-in-cell (PIC) method of Harlow [28]. Interestingly, the name "material point method" first appeared in the literature two years later in a description of an axisymmetric form of the method [56]. In both FLIP and MPM, the basic idea is the same: objects are discretized into particles, or material points, each of which contains all state data for the small region of material that it represents. This includes the position, mass, volume, velocity, stress and state of deformation of that material. MPM differs from other so called "mesh-free" particle methods in that, while each object is primarily represented by a collection of particles, a computational mesh is also an important part of the calculation. Particles do not interact with each other directly, rather the particle information is accumulated to the grid, where the equations of motion are integrated forward in time. This time advanced solution is then used to update the particle state.

The method usually uses a regular structured grid as a computational mesh. While this grid, in principle, deforms as the material that it is representing deforms, at the end of each timestep, it is reset to its original undeformed position, in effect providing a new computational grid for each timestep. The use of a regular structured grid for each time step has a number of computational advantages. Computation of spatial gradients is simplified. Mesh entanglement, which can plague fully Lagrangian techniques, such as the Finite Element Method (FEM), is avoided. MPM has also been successful in solving problems involving contact between colliding objects, having an advantage over FEM in that the use of the regular grid eliminates the need for doing costly searches for contact surfaces[6].

In addition to the advantages that MPM brings, as with any numerical technique, it has its own set of shortcomings. It is computationally more expensive than a comparable FEM code. Accuracy for MPM is typically lower than FEM, and errors associated with particles moving around the computational grid can introduce non-physical oscillations into the solution. Finally, numerical difficulties can still arise in simulations involving large deformation that will prematurely terminate the simulation. The severity of all of these issues (except for the expense) has been significantly reduced with the introduction of the Generalized Interpolation Material Point Method, or GIMP[8]. The basic concepts associated with GIMP will be described below. Throughout this document, MPM (which ends up being a special case of GIMP) will frequently be referred to interchangably with GIMP.

In addition, MPM can be incorporated with a multi-material CFD algorithm as the structural

component in a fluid-structure interaction formulation. This capability was first demonstrated in the CFDLIB codes from Los Alamos by Bryan Kashiwa and co-workers[34]. There, as in the Uintah-MPMICE component, MPM serves as the Lagrangian description of the solid material in a multimaterial CFD code. Certain elements of the solution procedure are based in the Eulerian CFD algorithm, including intermaterial heat and momentum transfer as well as satisfaction of a multimaterial equation of state. The use of a Lagrangian method such as MPM to advance the solution of the solid material eliminates the diffusion typically associated with Eulerian methods. The Uintah-MPM component will be described in later chapter of this manual.

Subsequent sections of this chapter will first give a relatively brief description of the MPM and GIMP algorithms. This will, of course, be focused mainly on describing the capabilities of the Uintah-MPM component. This is followed by a section that attempts to relate the information in Section 9.2 to the implementation in Uintah. Following that is a description of the information that goes into an input file. Finally, a number of examples are provided, along with representative results.

## 9.2 Algorithm Description

Time and space prohibit an exhaustive description of the theoretical underpinnings of the Material Point Method. Here we will concentrate on the discrete equations that result from applying a weak form analysis to the governing equations. The interested reader should consult [55, 57] for the development of these discrete equations in MPM, and [8] for the development of the equations for the GIMP method. These end up being very similar, the differences in how the two developments affect implementation will be described in Section 9.3.

In solving a structural mechanics problem with MPM, one begins by discretizing the object of interest into a suitable number of particles, or "material points". (**Aside:** What constitutes a suitable number is something of an open question, but it is typically advisable to use at least two particles in each computational cell in each direction, i.e. 4 particles per cell (PPC) in 2-D, 8 PPC in 3-D. In choosing the resolution of the computational grid, similar considerations apply as for any computational method (trade-off between time to solution and accuracy, use of resolution studies to ensure convergence in results, etc.).) Each of these particles will carry, minimally, the following variables:

- position - $\mathbf{x}_p$
- mass - $m_p$
- volume - $v_p$
- velocity - $\mathbf{v}_p$
- stress - $\boldsymbol{\sigma}_p$
- deformation gradient - $\mathbf{F}_p$

The description that follows is a recipe for advancing each of these variables from the current (discrete) time $n$ to the subsequent time $n + 1$. Note that particle mass, $m_p$, typically remains constant throughout a simulation unless solid phase reaction models are utilized, a feature that is not present in Uintah-MPM. (Such models are available in MPMICE, see Section 10.) It is also important to point out that the algorithm for advancing the timestep is based on the so-called

Update Stress Last (USL) algorithm. The superiority of this approach over the Update Stress First (USF) approach was clearly demonstrated by Wallstedt and Guilkey [61]. USF was the formulation used in Uintah until mid-2008.

The discrete momentum equation that results from the weak form is given as:

$$\mathbf{ma} \quad = \quad \mathbf{F}^{\text{ext}} - \mathbf{F}^{\text{int}} \tag{9.1}$$

where $\mathbf{m}$ is the mass matrix, $\mathbf{a}$ is the acceleration vector, $\mathbf{F}^{\text{ext}}$ is the external force vector (sum of the body forces and tractions), and $\mathbf{F}^{\text{int}}$ is the internal force vector resulting from the divergence of the material stresses. The construction of each of these quantities, which are based at the nodes of the computational grid, will be described below.

The solution begins by accumulating the particle state on the nodes of the computational grid, to form the mass matrix $\mathbf{m}$ and to find the nodal external forces $\mathbf{F}^{\text{ext}}$, and velocities, $\mathbf{v}$. In practice, a lumped mass matrix is used to avoid the need to invert a system of equations to solve Eq. 9.1 for acceleration. These quantities are calculated at individual nodes by the following equations, where the $\sum_p$ represents a summation over all particles:

$$m_i = \sum_p S_{ip} m_p, \qquad \mathbf{v}_i = \frac{\sum_p S_{ip} m_p \mathbf{v}_p}{m_i}, \qquad \mathbf{F}_i^{\text{ext}} \quad = \quad \sum_p S_{ip} \mathbf{F}_p^{\text{ext}} \tag{9.2}$$

and $i$ refers to individual nodes of the grid. $m_p$ is the particle mass, $\mathbf{v}_p$ is the particle velocity, and $\mathbf{F}_p^{\text{ext}}$ is the external force on the particle. The external forces that start on the particles typically the result of tractions, the application of which will be discussed in Section 9.5.9. $S_{ip}$ is the shape function of the $ith$ node evaluated at $\mathbf{x}_p$. The functional form of the shape functions differs between MPM and GIMP. This difference is discussed in Section 9.3.

Following the operations in Eq. 9.2, $\mathbf{F}^{\text{int}}$ is still required in order to solve for acceleration at the nodes. This is computed at the nodes as a volume integral of the divergence of the stress on the particles, specifically:

$$\mathbf{F}_i^{\text{int}} \quad = \quad \sum_p \mathbf{G}_{ip} \boldsymbol{\sigma}_p v_p, \tag{9.3}$$

where $\mathbf{G}_{ip}$ is the gradient of the shape function of the $ith$ node evaluated at $\mathbf{x}_p$, and $\boldsymbol{\sigma}_p$ and $v_p$ are the time $n$ values of particle stress and volume respectively.

Equation 9.1 can then be solved for $\mathbf{a}$.

$$\mathbf{a}_i \quad = \quad \frac{\mathbf{F}_i^{\text{ext}} - \mathbf{F}_i^{\text{int}}}{m_i} \tag{9.4}$$

An explicit forward Euler method is used for the time integration:

$$\mathbf{v}_i^L = \mathbf{v}_i + \mathbf{a}_i \Delta t \tag{9.5}$$

The time advanced grid velocity, $\mathbf{v}^L$ is used to compute a velocity gradient at each particle according to:

$$\nabla \mathbf{v}_p = \sum_i \mathbf{G}_{ip} \mathbf{v}_i^L \tag{9.6}$$

This velocity gradient is used to update the particle's deformation gradient, volume and stress. First, an incremental deformation gradient is computed using the velocity gradient:

$$\mathbf{dF}_p^{n+1} = (\mathbf{I} + \nabla \mathbf{v}_p \Delta t) \tag{9.7}$$

Particle volume and deformation gradient are updated by:

$$v_p^{n+1} = \mathrm{Det}(\mathbf{dF}_\mathrm{p}^{\mathrm{n+1}})\mathrm{v}_\mathrm{p}^\mathrm{n}, \qquad \mathbf{F}_\mathrm{p}^{\mathrm{n+1}} = \mathbf{dF}_\mathrm{p}^{\mathrm{n+1}}\mathbf{F}_\mathrm{p}^\mathrm{n} \tag{9.8}$$

Finally, the velocity gradient, and/or the deformation gradient are provided to a constitutive model, which outputs a time advanced stress at the particles. Specifics of this operation will be further discussed in Section 9.5.5

At this point in the timestep, the particle position and velocity are explicitly updated by:

$$\mathbf{v}_p(t + \Delta t) = \mathbf{v}_p(t) + \sum_i S_{ip}\mathbf{a}_i \Delta t \tag{9.9}$$

$$\mathbf{x}_p(t + \Delta t) = \mathbf{x}_p(t) + \sum_i S_{ip}\mathbf{v}_i^L \Delta t \tag{9.10}$$

This completes one timestep, in that the update of all six of the variables enumerated above (with the exception of mass, which is assumed to remain constant) has been accomplished. Conceptually, one can imagine that, since an acceleration and velocity were computed at the grid, and an interval of time has passed, the grid nodes also experienced a displacement. This displacement also moved the particles in an isoparametric fashion. In practice, particle motion is accomplished by Equation 9.10, and the grid never deforms. So, while the MPM literature will often refer to resetting the grid to its original configuration, in fact, this isn't necessary as the grid nodes never leave that configuration. Regardless, at this point, one is ready to advance to the next timestep.

The algorithm described above is the core of the Uintah-MPM implementation. However, it neglects a number of important considerations. The first is kinematic boundary conditions on the grid for velocity and acceleration. The manner in which these are handled will be described in Section 9.4. Next, is the use of advanced contact algorithms. By default, MPM enforces no-slip, no-interpenetration contact. This feature is extremely useful, but it also means that two bodies initially in "contact" (meaning that they both contain particles whose data are accumulated to common nodes) behave as if they are a single body. To enable multi-field simulations with frictional contact, or to impose displacement based boundary conditions, e.g. a rigid piston, additional steps must be taken. These steps implement contact formulations such as that described by Bardenhagen, et al.[7]. The *use* of the contact algorithms is described in Section 9.5.7, but the reader will be referred to the relevant literature for their development. Lastly, heat conduction is also available in the explicit MPM code, although it may be neglected via a run time option in the input file. Explicit MPM is typically used for high rate simulations in which heat conduction is negligible.

## 9.3 Shape functions for MPM and GIMP

In both MPM and GIMP, the basic idea is the same: objects are discretized into particles, or material points, each of which contains all state data for the small region of material that it represents. In MPM, these particles are spatially Dirac delta functions, meaning that the material that each

represents is assumed to exist at a single point in space, namely the position of the particle. Interactions between the particles and the grid take place using weighting functions, also known as shape functions or interpolation functions. These are typically, but not necessarily, linear, bilinear or trilinear in one, two and three dimensions, respectively.

More recently, Bardenhagen and Kober [8] generalized the development that gives rise to MPM, and suggested that MPM may be thought of as a subset of their "Generalized Interpolation Material Point" (GIMP) method. In the family of GIMP methods one chooses a characteristic function $\chi_p$ to represent the particles and a shape function $S_i$ as a basis of support on the computational nodes. An effective shape function $\bar{S}_{ip}$ is found by the convolution of the $\chi_p$ and $S_i$ which is written as:

$$\bar{S}_{ip}(\mathbf{x}_p) = \frac{1}{V_p} \int_{\Omega_p \cap \Omega} \chi_p(\mathbf{x} - \mathbf{x}_p) S_i(\mathbf{x}) \, \mathrm{d}\mathbf{x}. \tag{9.11}$$

While the user has significant latitude in choosing these two functions, in practice, the choice of $S_i$ is usually given (in one-dimension) as,

$$S_i(x) = \begin{cases} 1 + (x - x_i)/h & -h < x - x_i \leq 0 \\ 1 - (x - x_i)/h & 0 < x - x_i \leq h \\ 0 & \text{otherwise,} \end{cases} \tag{9.12}$$

where $x_i$ is the vertex location, and $h$ is the cell width, assumed to be constant in this formulation, although this is not a general restriction on the method. Multi-dimensional versions are constructed by forming tensor products of the one-dimensional version in the orthogonal directions.

When the choice of characteristic function is the Dirac delta,

$$\chi_p(\mathbf{x}) = \delta(\mathbf{x} - \mathbf{x}_p) V_p, \tag{9.13}$$

where $\mathbf{x}_p$ is the particle position, and $V_p$ is the particle volume, then traditional MPM is recovered. In that case, the effective shape function is still that given by Equation 9.12. Its gradient is given by:

$$G_i(x) = \begin{cases} 1/h & -h < x - x_i \leq 0 \\ -1/h & 0 < x - x_i \leq h \\ 0 & \text{otherwise,} \end{cases} \tag{9.14}$$

Plots of Equations 9.12 and 9.14 are shown below. The discontinuity in the gradient gives rise to poor accuracy and stability properties.

Typically, when an analyst indicates that they are "using GIMP" this implies use of the linear grid basis function given in Eq. 9.12 and a "top-hat" characteristic function, given by (in one-dimension),

$$\chi_p(x) = H(x - (x_p - l_p)) - H(x - (x_p + l_p)), \tag{9.15}$$

where $H(x)$ is the Heaviside function ($H(x) = 0$ if $x < 0$ and $H(x) = 1$ if $x \geq 0$) and $l_p$ is the half-length of the particle. When the convolution indicated in Eq. 9.11 is carried out using the

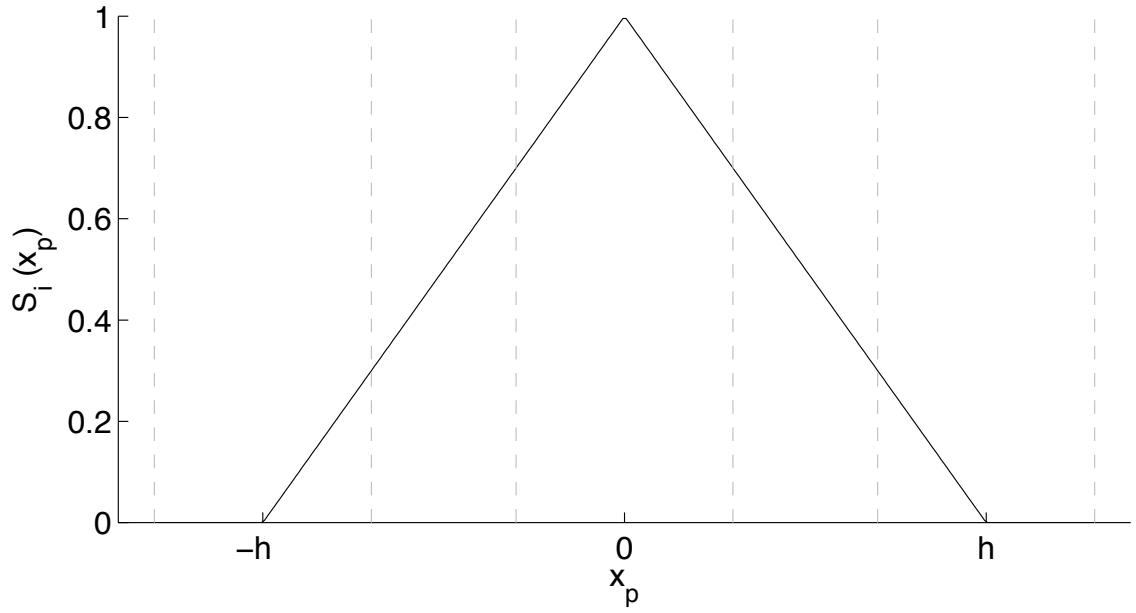



Figure 9.1: Effective shape function when using traditional MPM.

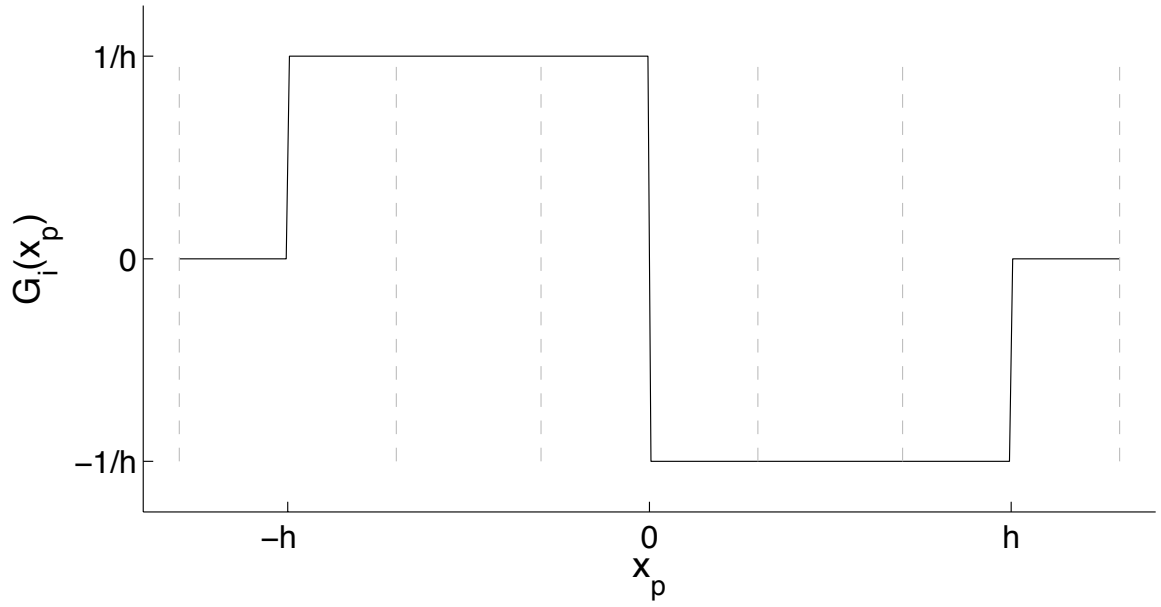



Figure 9.2: Gradient of the effective shape function when using traditional MPM.

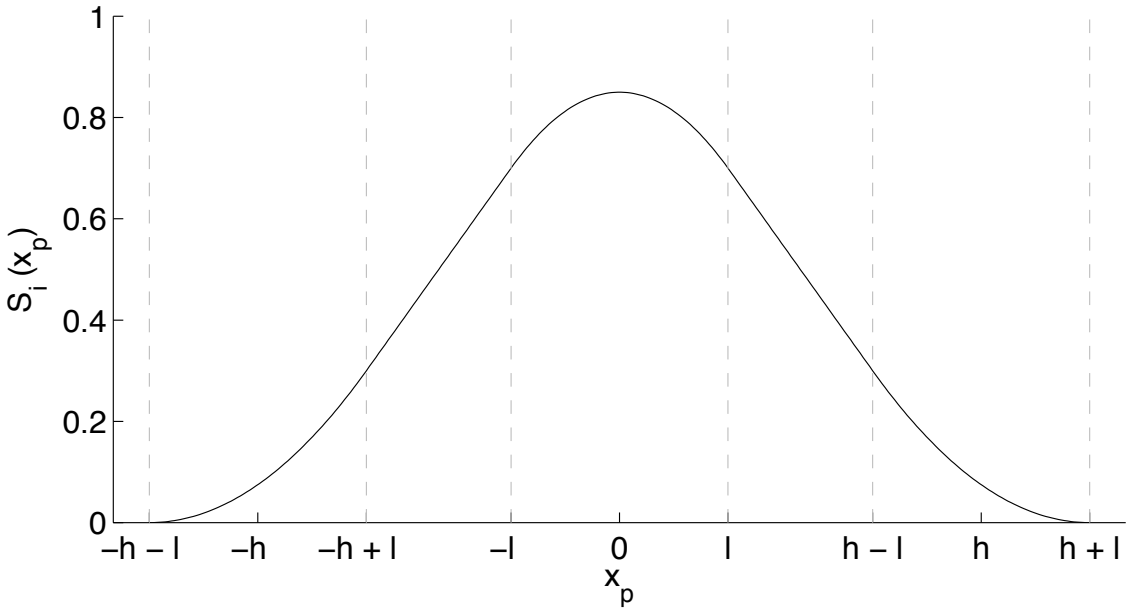



Figure 9.3: Effective shape function when using GIMP.

expressions in Eqns. 9.12 and 9.15, a closed form for the effective shape function can be written as:

$$S_i\left(x_p\right) = \begin{cases} \frac{\left(h+l_p+\left(x_p-x_i\right)\right)^2}{4hl_p} & -h-l_p < x_p - x_i \leq -h+l_p \\ 1+\frac{\left(x_p-x_i\right)}{h} & -h+l_p < x_p - x_i \leq -l_p \\ 1-\frac{\left(x_p-x_i\right)^2+l_p^2}{2hl_p} & -l_p < x_p - x_i \leq l_p \\ 1-\frac{\left(x_p-x_i\right)}{h} & l_p < x_p - x_i \leq h-l_p \\ \frac{\left(h+l_p-\left(x_p-x_i\right)\right)^2}{4hl_p} & h-l_p < x_p - x_i \leq h+l_p \\ 0 & \text{otherwise,} \end{cases} \tag{9.16}$$

The gradient of which is:

$$G_i(x_p) = \begin{cases} \frac{h+l_p+\left(x_p-x_i\right)}{2hl_p} & -h-l_p < x_p - x_i \leq -h+l_p \\ \frac{1}{h} & -h+l_p < x_p - x_i \leq -l_p \\ -\frac{\left(x_p-x_i\right)}{hl_p} & -l_p < x_p - x_i \leq l_p \\ -\frac{1}{h} & l_p < x_p - x_i \leq h-l_p \\ -\frac{h+l_p-\left(x_p-x_i\right)}{2hl_p} & h-l_p < x_p - x_i \leq h+l_p \\ 0 & \text{otherwise,} \end{cases} \tag{9.17}$$

Plots of Equations 9.16 and 9.17 are shown below. The continuous nature of the gradients are largely responsible for the improved robustness and accuracy of GIMP over MPM.

There is one further consideration in defining the effective shape function, and that is whether or not the size (length in 1-D) of the particle is kept fixed (denoted as "UGIMP" here) or is allowed to evolve due to material deformations ("Finite GIMP" or "Contiguous GIMP" in (1) and "cpGIMP" here). In one-dimensional simulations, evolution of the particle (half-)length is straightforward,

$$l_p^n = F_p^n l_p^0, \tag{9.18}$$

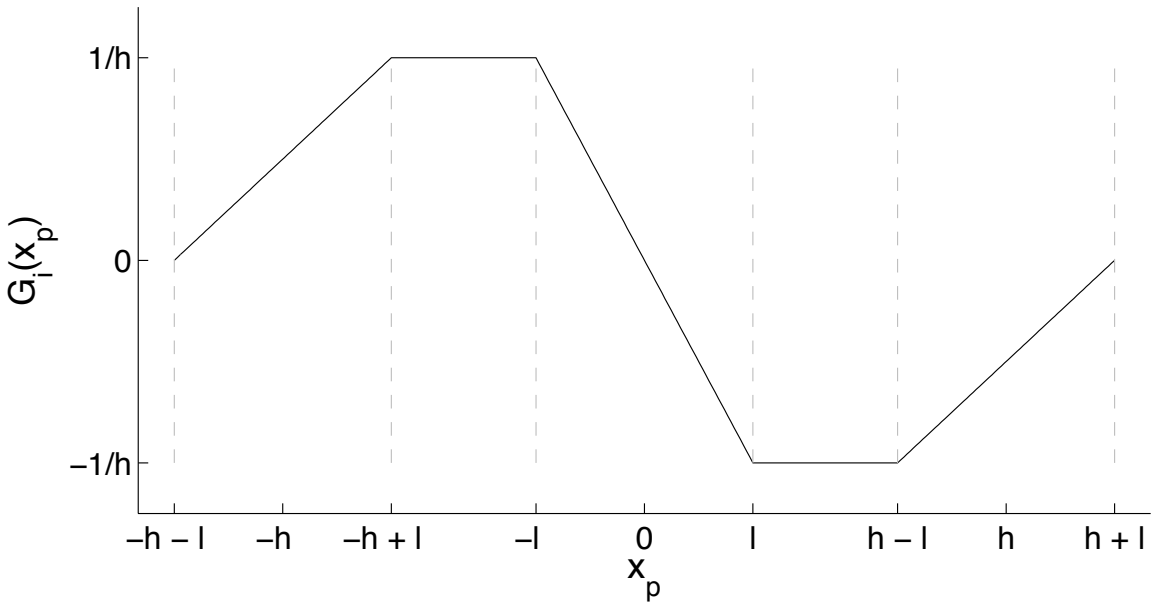



Figure 9.4: Gradient of the effective shape function when using GIMP.

where $F_p^n$ is the deformation gradient at time $n$. In multi-dimensional simulations, a similar approach can be used, assuming an initially rectangular or cuboid particle, to find the current particle shape. The difficulty arises in evaluating Eq. 9.11 for these general shapes. One approach, apparently effective, has been to create a cuboid that circumscribes the deformed particle shape [37]. Alternatively, one can assume that the particle size remains constant (insofar as it applies to the effective shape function evaluations only). This is the approach currently implemented in Uintah.

## 9.4 Uintah Implementation

Users of Uintah-MPM needn't necessarily bother themselves with the implementation in code of the algorithm described above. This section is intended to serve as a reference for users who find themselves needing to modify the source code, or those who are simply interested. Anyone just wishing to run MPM simulations may skip ahead to Sections 9.5 and 9.6. The goal of this section is to provide a mapping from the the algorithm described above to the software that carries it out. This won't be exhaustive, but will be a good starting point for the motivated reader.

The source code for the Uintah-MPM implementation can be found in

```
src/CCA/Components/MPM
```

Within that directory are a number of files and subdirectories, these will be discussed as needed. For the moment, consider the various files that end in "MPM.cc":

```
AMRMPM.cc  FractureMPM.cc  ImpMPM.cc  RigidMPM.cc  SerialMPM.cc  ShellMPM.cc
```

`AMRMPM.cc` is the nascent beginnings of an AMR implementation of MPM. It is far from complete and should be ignored. `FractureMPM.cc` is an implementation of the work of Guo and Nairn [24], and while it is viable, it is undocumented and unsupported. `ShellMPM.cc` is a treatment of MPM particles as shell and membrane elements, developed by Biswajit Bannerjee. It is also viable, but also undocumented and unsupported. `ImpMPM.cc` is an implicit time integration form of MPM

based on the work of Guilkey and Weiss [23]. It is also viable, and future releases of Uintah will include documentation of its capabilities and uses. For now, interested readers should contact Jim Guilkey directly for more information. `RigidMPM.cc` contains a very reduced level of functionality, and is used solely in conjunction with the MPMArches component.

This leaves `SerialMPM.cc`. This contains, despite its name, the parallel implementation of the algorithm described above in Section 9.2. For now, we will skip over the initialization procedures such as:

```
SerialMPM::problemSetup
SerialMPM::scheduleInitialize
SerialMPM::actuallyInitialize
```

and focus mainly on the timestepping algorithm described above. Reference will be made back to these functions as needed in Section 9.5.

Each of the Uintah components contains a function called `scheduleTimeAdvance`. The algorithms implemented in these components are broken into a number of steps. The implementation of these steps in Uintah take place in "tasks". Each task is responsible for performing the calculations needed to accomplish that step in the algorithm. Thus, each task requires some data upon which to operate, and it also creates some data, either as a final result, or as input to a subsequent task. Before individual tasks are executed, each is first "scheduled". The scheduling of tasks describes the dataflow and data dependencies for a given algorithm. By describing the data dependencies, both temporally and spatially, each task can be executed in the proper order, and communication tasks can automatically be generated by the Uintah infrastructure to achieve parallelism. Thus, scheduleTimeAdvance calls a series of functions, each of which schedules the individual tasks. Let's begin by looking at the `scheduleTimeAdvance` for `SerialMPM`, pasted below.

```
void
SerialMPM::scheduleTimeAdvance(const LevelP & level,
                               SchedulerP   & sched)
{
  MALLOC_TRACE_TAG_SCOPE("SerialMPM::scheduleTimeAdvance()");
  if (!flags->doMPMOnLevel(level->getIndex(), level->getGrid()->numLevels()))
    return;

  const PatchSet* patches = level->eachPatch();
  const MaterialSet* matls = d_sharedState->allMPMMaterials();

  scheduleApplyExternalLoads(             sched, patches, matls);
  scheduleInterpolateParticlesToGrid(     sched, patches, matls);
  scheduleExMomInterpolated(              sched, patches, matls);
  scheduleComputeContactArea(             sched, patches, matls);
  scheduleComputeInternalForce(           sched, patches, matls);

  scheduleComputeAndIntegrateAcceleration(sched, patches, matls);
  scheduleExMomIntegrated(                sched, patches, matls);
  scheduleSetGridBoundaryConditions(      sched, patches, matls);
  scheduleSetPrescribedMotion(            sched, patches, matls);
  scheduleComputeStressTensor(            sched, patches, matls);
  if(flags->d_doExplicitHeatConduction){
    scheduleComputeHeatExchange(          sched, patches, matls);
    scheduleComputeInternalHeatRate(      sched, patches, matls);
    scheduleComputeNodalHeatFlux(         sched, patches, matls);
    scheduleSolveHeatEquations(           sched, patches, matls);
    scheduleIntegrateTemperatureRate(     sched, patches, matls);
```

```
  }
  scheduleAddNewParticles(                sched, patches, matls);
  scheduleConvertLocalizedParticles(      sched, patches, matls);
  scheduleInterpolateToParticlesAndUpdate(sched, patches, matls);

  if(flags->d_canAddMPMMaterial){
    //  This checks to see if the model on THIS patch says that it's
    //  time to add a new material
    scheduleCheckNeedAddMPMMaterial(         sched, patches, matls);

    //  This one checks to see if the model on ANY patch says that it's
    //  time to add a new material
    scheduleSetNeedAddMaterialFlag(         sched, level,   matls);
  }

  sched->scheduleParticleRelocation(level, lb->pXLabel_preReloc,
                                    d_sharedState->d_particleState_preReloc,
                                    lb->pXLabel,
                                    d_sharedState->d_particleState,
                                    lb->pParticleIDLabel, matls);
  if(d_analysisModule){
    d_analysisModule->scheduleDoAnalysis( sched, level);
  }
}
```

The preceding includes scheduling for a number of rarely used features. For now, let's condense the preceding to the essential tasks:

```
void
SerialMPM::scheduleTimeAdvance(const LevelP & level,
                               SchedulerP   & sched)
{
  if (!flags->doMPMOnLevel(level->getIndex(), level->getGrid()->numLevels()))
    return;

  const PatchSet* patches = level->eachPatch();
  const MaterialSet* matls = d_sharedState->allMPMMaterials();

  scheduleApplyExternalLoads(             sched, patches, matls);
  scheduleInterpolateParticlesToGrid(     sched, patches, matls);
  scheduleExMomInterpolated(              sched, patches, matls);
  scheduleComputeInternalForce(           sched, patches, matls);

  scheduleComputeAndIntegrateAcceleration(sched, patches, matls);
  scheduleExMomIntegrated(                sched, patches, matls);
  scheduleSetGridBoundaryConditions(      sched, patches, matls);
  scheduleComputeStressTensor(            sched, patches, matls);
  scheduleInterpolateToParticlesAndUpdate(sched, patches, matls);

  sched->scheduleParticleRelocation(level, lb->pXLabel_preReloc,
                                    d_sharedState->d_particleState_preReloc,
                                    lb->pXLabel,
                                    d_sharedState->d_particleState,
                                    lb->pParticleIDLabel, matls);
}
```

As described above, each of the "schedule" functions describes dataflow, and it also calls the function that actually executes the task. The naming convention is illustrated by an example,

scheduleComputeAndIntegrateAcceleration calls computeAndIntegrateAcceleration. Let's examine this particular task, which executes Equations 9.4 and 9.5, more carefully. First, the scheduling of the task:

```
void SerialMPM::scheduleComputeAndIntegrateAcceleration(SchedulerP& sched,
                                                        const PatchSet* patches,
                                                        const MaterialSet* matls)
{
  if (!flags->doMPMOnLevel(getLevel(patches)->getIndex(),
                           getLevel(patches)->getGrid()->numLevels()))
    return;

  printSchedule(patches,cout_doing,"MPM::scheduleComputeAndIntegrateAcceleration\t\t\t\t");

  Task* t = scinew Task("MPM::computeAndIntegrateAcceleration",
                        this, &SerialMPM::computeAndIntegrateAcceleration);

  t->requires(Task::OldDW, d_sharedState->get_delt_label() );

  t->requires(Task::NewDW, lb->gMassLabel,          Ghost::None);
  t->requires(Task::NewDW, lb->gInternalForceLabel, Ghost::None);
  t->requires(Task::NewDW, lb->gExternalForceLabel, Ghost::None);
  t->requires(Task::NewDW, lb->gVelocityLabel,      Ghost::None);

  t->computes(lb->gVelocityStarLabel);
  t->computes(lb->gAccelerationLabel);

  sched->addTask(t, patches, matls);
}
```

The if statement basically directs the schedule to only do this task on the finest level (MPM can be used in AMR simulations, but only at the finest level.) The printSchedule command is in place for debugging purposes, this type of print statement can be turned on by setting an environmental variable. The real business of this task begins with the declaration of the Task. In the task declaration, the function associated with that task is identified. Subsequent to that is a description of the data dependencies. Namely, this task requires the mass, internal and external forces as well as velocity on the grid. No ghost data are required as this task is a node by node calculation. It also requires the timestep size. Note also that most of the required data are needed from the NewDW where DW refers to DataWarehouse. This simply means that these data were calculated by an earlier task in the current timestep. The timestep size for this step was computed in the previous timestep, and thus is required from the OldDW. Finally, this task computes the acceleration and time advanced velocity at each node.

The code to execute this task is as follows:

```
void SerialMPM::computeAndIntegrateAcceleration(const ProcessorGroup*,
                                                const PatchSubset* patches,
                                                const MaterialSubset*,
                                                DataWarehouse* old_dw,
                                                DataWarehouse* new_dw)
{
  for(int p=0;p<patches->size();p++){
    const Patch* patch = patches->get(p);
    printTask(patches, patch,cout_doing,"Doing computeAndIntegrateAcceleration\t\t\t\t");
```

```
    Ghost::GhostType  gnone = Ghost::None;
    Vector gravity = d_sharedState->getGravity();
    for(int m = 0; m < d_sharedState->getNumMPMMatls(); m++){
      MPMMaterial* mpm_matl = d_sharedState->getMPMMaterial( m );
      int dwi = mpm_matl->getDWIndex();

      // Get required variables for this patch
      constNCVariable<Vector> internalforce, externalforce, velocity;
      constNCVariable<double> mass;

      delt_vartype delT;
      old_dw->get(delT, d_sharedState->get_delt_label(), getLevel(patches) );

      new_dw->get(internalforce,lb->gInternalForceLabel, dwi, patch, gnone, 0);
      new_dw->get(externalforce,lb->gExternalForceLabel, dwi, patch, gnone, 0);
      new_dw->get(mass,         lb->gMassLabel,          dwi, patch, gnone, 0);
      new_dw->get(velocity,     lb->gVelocityLabel,      dwi, patch, gnone, 0);

      // Create variables for the results
      NCVariable<Vector> velocity_star,acceleration;
      new_dw->allocateAndPut(velocity_star, lb->gVelocityStarLabel, dwi, patch);
      new_dw->allocateAndPut(acceleration,  lb->gAccelerationLabel, dwi, patch);

      acceleration.initialize(Vector(0.,0.,0.));
      double damp_coef = flags->d_artificialDampCoeff;

      for(NodeIterator iter=patch->getExtraNodeIterator__New();
                       !iter.done();iter++){
        IntVector c = *iter;
        Vector acc(0.,0.,0.);
        if (mass[c] > flags->d_min_mass_for_acceleration){
          acc  = (internalforce[c] + externalforce[c])/mass[c];
          acc -= damp_coef*velocity[c];
        }
        acceleration[c] = acc +  gravity;
        velocity_star[c] = velocity[c] + acceleration[c] * delT;
      }
    }    // matls
  }
}
```

This task contains three nested for loops. First, is a loop over all of the "patches" that the processor executing this task is responsible for. Next is a loop over all materials (imagine a simulation involving the interaction between, say, tungsten and copper). Within this loop, the required data are retrieved from the `new_dw` (New DataWarehouse) and space for the data to be created is allocated. The final loop is over all of the nodes on the current patch, and the calculations described by Equations 9.4 and 9.5 are carried out. (This also includes a linear damping term not described above.)

Let's consider each task in turn. The remaining tasks will be described in much less detail, but the preceding dissection of a fairly simple task, along with a description of what the remaining tasks are intended to accomplish, should allow interested individuals to follow the remainder of the Uintah-MPM implementation.

1. `scheduleApplyExternalLoads` This task is mainly responsible for applying traction boundary conditions described in the input file. This is done by assigning external force vectors to

the particles. If the user wishes to apply a load that is not possible to acheive via the input file options, it is straightforward to modify the code here to do "one-off" tests.

2. `scheduleInterpolateParticlesToGrid` The name of this task was poorly chosen, but has persisted. This task carries out the operations given in Equation 9.2. It also sets boundary conditions on some of the variables, such as the grid temperature, and grid velocity (in the case of symmetry BCs).

3. `scheduleExMomInterpolated` This task actually exists in one of the contact models which can be found in the `Contact` directory. Each of those models has two main tasks. This is the the first of those. It is responsible for modifying the grid velocity computed by interpolateParticlesToGrid according to the rules for the particular contact model chosen in the input file. These models are briefly described in Section 9.5.7.

4. `scheduleComputeInternalForce` This task computes the volume integral of the divergence of stress. Specifically, it carries out the operation given in Equation 9.3. It also computes some diagnostic data, if requested in the input file, such as the reaction forces (tractions) on the boundaries of the computational domain.

5. `scheduleComputeAndIntegrateAcceleration` As described previously, this task carries out the operations described in Equations 9.4 and 9.5.

6. `scheduleExMomIntegrated` This is the second of the contact tasks (see above). It is responsible for modifiying the time advanced grid velocity computed in `computeAndIntegrateAcceleration.`

7. `scheduleSetGridBoundaryConditions` This task sets boundary conditions on the time advanced grid velocity. It also sets an acceleration boundary condition as well. However, rather than just setting the acceleration to a given value, it is computed by solving Equation 9.5 for acceleration, and then recomputing the acceleration (on all nodes) as:

$$\mathbf{a}_i = \frac{\mathbf{v}_i^L - \mathbf{v}_i}{\Delta t} \tag{9.19}$$

   Doing this operation on all nodes has several advantages. For most interior nodes, the value for acceleration will be unchanged, but for nodes on the where the velocity has been altered by enforcing boundary conditions, and for nodes at which the contact models have altered the velocity, the acceleration will be modified to reflect that alteration.

8. `scheduleComputeStressTensor` The task, `computeStressTensor,` exists in each of the models in the `ConstitutiveModel` directory. Each model is responsible for carrying out the operations given in Equation 9.8, and of course, as the name implies, it also computes the material stress. This task has one additional important function, and that is computing the timestep size for the subsequent step. The CFL condition dictates that the timestep size be limited according to:

$$\Delta t < \frac{\Delta x}{c + |u|} \tag{9.20}$$

   where $\Delta x$ is the cell spacing, $c$ is the wavespeed in the material, and $|u|$ is the magnitude of the local velocity. Because the wavespeed may depend on the state of stress that a material is in, this task provides a convenient time at which to make this calculation. A timestep size is

computed for all particles, and the minimum for the particle set on a given patch is put into a "reduction variable". The Uintah infrastructure then does a global reduction to select the smallest timestep from across the domain.

9. `scheduleInterpolateToParticlesAndUpdate` This task carries out the operations in Equations 9.9 and 9.10, namely updating the particle state based on the grid solution.

10. `scheduleParticleRelocation` This task is not actually located in the MPM code, but in the Uintah infrastructure. The idea is that as particles move, some will cross patch boundaries, and their data will need to be sent to other processors. This task is responsible for identifying particles that have left the patch that they were on, finding where they went, and sending their data to the correct processor.

## 9.5 Uintah Specification

Uintah input files are constructed in XML format. Each begins with:

```
<?xml version='1.0' encoding='ISO-8859-1' ?>
```

while the remainder of the file is enclosed within the following tags:

```
<Uintah_specification>
</Uintah_specification>
```

The following subsections describe the remaining inputs needed to construct an input file for an MPM simulation. The order of the various sections of the input file is not important. **The MPM, ICE and MPMICE components are dimensionless calculations. It is the responsibility of the analyst to provide the following inputs using a consistent set of units.**

### 9.5.1 Common Inputs

Each Uintah component is invoked using a single executable called *sus* , which chooses the type of simulation to execute based on the *SimulationComponent* tag in the input file. For the case of MPM simulations, this looks like:

```
<SimulationComponent type="mpm" />
```

There are a number of fields that are required for any component. The first is that describing the timestepping parameters, these are largely common to all components, and are described in Section 2.5. The only one that bears commenting on at this point is:

```
<timestep_multiplier>    0.5     </timestep_multiplier>
```

This is effectively the CFL number for MPM simulations, that is the number multiplied by the timestep size that is automatically calculated by the MPM code. Experience indicates that one should generally keep this value below 0.5, and should expect to use smaller values for high-rate, large-deformation simulations.

The next field common to the input files for all components is:

```
<DataArchiver>
</DataArchiver>
```

This is described in Section 2.6. To see a list of variables available for saving in MPM simulations, execute the following command from the `StandAlone` directory:

```
inputs/labelNames mpm
```

Note that for visualizing particle data, one must save `p.x,` and at least one other variable by which to color the particles.

The other principle common field is that which describes the computational grid. For MPM, this is typically broken up into two parts, the `<Level>` section specifies the physical extents and spatial resolution of the grid. For more information, consult Section 2.10. The other part specifies kinematic boundary conditions on the grid boundaries. These are discussed below in Section 9.5.8.

### 9.5.2 Physical Constants

The only physical constant required (or optional for that matter) for MPM simulations is gravity, this is specified as:

```
<PhysicalConstants>
   <gravity>            [0,0,0]   </gravity>
</PhysicalConstants>
```

### 9.5.3 MPM Flags

There are many options available when running MPM simulations. These are generally specified in the `<MPM>` section of the input file. Below is a list of these options taken from `inputs/UPS_SPEC/mpm_spec.xml` This file also gives possible values, or at least expected datatype, for these flags. A description of their functionality is forthcoming, in the meantime, consult the code and input files. A default value is set for many, see `MPM/MPMFlags.cc` for more.

```
<MPM>
 <!-- These are commonly used options -->
  <artificial_damping_coeff           spec="OPTIONAL DOUBLE 'positive'"/>
  <artificial_viscosity               spec="OPTIONAL BOOLEAN" />
  <artificial_viscosity_coeff1        spec="OPTIONAL DOUBLE" />
  <artificial_viscosity_coeff2        spec="OPTIONAL DOUBLE" />
  <axisymmetric                       spec="OPTIONAL BOOLEAN" />
  <boundary_traction_faces            spec="OPTIONAL STRING" />
  <DoExplicitHeatConduction           spec="OPTIONAL BOOLEAN" />
  <DoPressureStabilization            spec="OPTIONAL BOOLEAN" />
  <erosion                            spec="OPTIONAL NO_DATA"
        attribute1="algorithm REQUIRED STRING 'none, KeepStress, ZeroStress, RemoveMass'" />
  <interpolator                       spec="OPTIONAL STRING 'linear, gimp, 3rdorderBS, 4thorderBS'" />
  <minimum_particle_mass              spec="OPTIONAL DOUBLE 'positive'"/>
  <minimum_mass_for_acc               spec="OPTIONAL DOUBLE 'positive'"/>
  <maximum_particle_velocity          spec="OPTIONAL DOUBLE 'positive'"/>
  <testForNegTemps_mpm                spec="OPTIONAL BOOLEAN" />
  <time_integrator                    spec="OPTIONAL STRING 'explicit, fracture, implicit'" />
  <use_load_curves                    spec="OPTIONAL BOOLEAN" />
  <UsePrescribedDeformation           spec="OPTIONAL BOOLEAN" />
  <withColor                          spec="OPTIONAL BOOLEAN" />

 <!-- These are not commonly used options -->
  <CanAddMPMMaterial                  spec="OPTIONAL BOOLEAN" />
```

```
  <create_new_particles               spec="OPTIONAL BOOLEAN" />
  <do_contact_friction_heating        spec="OPTIONAL BOOLEAN" />
  <do_grid_reset                      spec="OPTIONAL BOOLEAN" />
  <DoThermalExpansion                 spec="OPTIONAL BOOLEAN" />
  <ForceBC_force_increment_factor     spec="OPTIONAL DOUBLE" />
  <manual_new_material                spec="OPTIONAL BOOLEAN" />
  <interpolateParticleTempToGridEveryStep spec="OPTIONAL BOOLEAN" />
  <temperature_solve                  spec="OPTIONAL BOOLEAN" />

 <!-- THE FOLLOWING APPLY ONLY TO THE IMPLICIT MPM CODE -->
  <dynamic                            spec="OPTIONAL BOOLEAN" />
  <solver                             spec="OPTIONAL STRING 'petsc, simple'" />
  <convergence_criteria_disp          spec="OPTIONAL DOUBLE 'positive'"/>
  <convergence_criteria_energy        spec="OPTIONAL DOUBLE 'positive'"/>
  <num_iters_to_decrease_delT         spec="OPTIONAL INTEGER" />
  <num_iters_to_increase_delT         spec="OPTIONAL INTEGER" />
  <iters_before_timestep_restart      spec="OPTIONAL INTEGER" />
  <DoTransientImplicitHeatConduction  spec="OPTIONAL BOOLEAN" />
  <delT_decrease_factor               spec="OPTIONAL DOUBLE" />
  <delT_increase_factor               spec="OPTIONAL DOUBLE" />
  <DoImplicitHeatConduction           spec="OPTIONAL BOOLEAN" />
  <DoMechanics                        spec="OPTIONAL BOOLEAN" />

  <!-- THE FOLLOWING APPLY ONLY TO THE FRACTURE MPM CODE -->
  <dadx                               spec="OPTIONAL DOUBLE" />
  <smooth_crack_front                 spec="OPTIONAL BOOLEAN" />
  <calculate_fracture_parameters      spec="OPTIONAL BOOLEAN" />
  <do_crack_propagation               spec="OPTIONAL BOOLEAN" />
  <use_volume_integral                spec="OPTIONAL BOOLEAN" />
</MPM>
```

### 9.5.4 Material Properties

The `Material Properties` section of the input file actually contains not only those, but also the geometry and initial condition data as well. Below is a simple example, copied from `inputs/MPM/disks.ups`. The `name` field is optional. The first field is the material `<density>`. The `<constitutive_model>` field refers to the model used to generate a stress tensor on each material point. The use of these models is described in detail in Section 9.5.5. Next are the thermal transport properties, `<thermal_conductivity>` and `<specific_heat>`. Note that these are required even if heat conduction is not being computed. These are the required material properties. There are additional optional parameters that are used in other auxiliary calculations, for a list of these see the `inputs/UPS_SPEC/mpm_spec.xml` .

Next is the specification of the geometry, and, along with it, the initial state of the material contained in that geometry. For more information on how initial geometry can be specified, see Section 2.8. Within the `<geom_object>` is the `<res>` field. This indicates how many particles per cell are to be used in each of the coordinate directions. Following that are initial values for velocity and temperature. Finally, the `<color>` designation has a number of uses, for example when one wishes to identify initially distinct regions of the same material. In Section 9.5.10 is a description of how this field is used to identify particles for on the fly data extraction.

An arbitray number of `<material>` fields can be specified. As the calculation proceeds, each of these materials has their own field variables, and, as such, each material behaves independently of the others. Interactions between materials occur as a result of "contact" models. Their use is

described in detail in Section 9.5.7.

```
<MaterialProperties>
   <MPM>
       <material name="disks">
          <density>1000.0</density>
          <constitutive_model type="comp_mooney_rivlin">
             <he_constant_1>100000.0</he_constant_1>
             <he_constant_2>20000.0</he_constant_2>
             <he_PR>.49</he_PR>
          </constitutive_model>
          <thermal_conductivity>1.0</thermal_conductivity>
          <specific_heat>5</specific_heat>
          <geom_object>
              <cylinder label = "gp1">
                 <bottom>[.25,.25,.05]</bottom>
                 <top>[.25,.25,.1]</top>
                 <radius> .2 </radius>
              </cylinder>
              <res>[2,2,2]</res>
              <velocity>[2.0,2.0,0]</velocity>
              <temperature>300</temperature>
              <color>             0               </color>
           </geom_object>
       </material>

       <contact>
         <type>null</type>
         <materials>[0]</materials>
       </contact>
   </MPM>
</MaterialProperties>
```

### 9.5.5 Constitutive Models

The MPM code contains a large number of constitutive models that provide a Cauchy stress on each particle based on the velocity gradient computed at that particle. The following is a list and very brief description of the most commonly used models. The reader may wish to consult the `inputs/MPM` and `inputs/MPMICE` directories to find explicit examples of the use of these models, and others not described below.

1. **Compressible Neo-Hookean Model** There are implementations of several hyperelastic-plastic model described by Simo and Hughes[51] (pp. 307 – 321). The model is dubbed "Unified Compressible Neo-Nookean Model" or UCNH for short. Models can still be specified with old input file specifications, (i.e. comp_neo_hook, comp_neo_hook_plastic, cnh_damage, cnhp_damage) however these are merely wrappers for the underyling UCNH model. Plastic flow and failure can be modelled in addition to elasticity by specifying several additional options with input flags. This models is very robust, and relatively straightforward because hyperelastic models don't require rotation back and forth between laboratory and material frames of reference.

   NOTE: Support for Implicit CNH and CNH with specified solver is still lacking (for a short time).

   The basic input section for UCNH:

```
<constitutive_model type="UCNH">
  <!-- Necessary flags for all CNH models -->
   <bulk_modulus>   8.9e9  </bulk_modulus>
   <shear_modulus>  3.52e9 </shear_modulus>
   <useModifiedEOS> true   </useModifiedEOS>

  <!-- Plasticity Parameters -->
   <usePlasticity>     true  </usePlasticity>
   <yield_stress>      100.0 </yield_stress>
   <hardening_modulus> 500.0 </hardening_modulus>
   <alpha>             1.0   </alpha>

</constitutive_model>
```

A fairly sophisticated means of seeding explicit material heterogeneity is also provided for. To use these features the following four steps are required:

1. To allow for failure (by material point erosion) the following must be set:

In the `<MPM>` block, the erosion algorithm must be set to one of the following:

```
<erosion algorithm="AllowNoTension"/>
<erosion algorithm="AllowNoShear"/>
<erosion algorithm="ZeroStress"/>
```

In the `<constitutive_model>` block:

```
<useDamage>true</useDamage>
```

2. The failure surface type must be specified. This is also in the `<constitutive_model>` block. One of the following must be specified:

```
<failure_criteria>  MohrCoulomb </failure_criteria>
<failure_criteria>  MaximumPrincipalStress </failure_criteria>
<failure_criteria>  MaximumPrincipalStrain </failure_criteria>
```

The cohesion, $c$, is assigned using a distribution, as described below. For the maximum principal stress and strain failure criteria, the cohesion is the maximum value of principal stress or strain that may be obtained (must be positive). The MohrCoulomb failure criteria is given by

$$\frac{\sigma_3 - \sigma_1}{2} = c\cos(\phi) - \frac{\sigma_3 + \sigma_1}{2}\sin(\phi) \tag{9.21}$$

where $\sigma_i$ are the ordered principal stresses, positive in tension ($\sigma_3 > \sigma_2 > \sigma_1$). Note, the MohrCoulomb failure surface also requires a friction angle, $\phi$, (in degrees):

```
<friction_angle> friction angle </friction_angle>
```

A tensile cutoff failure surface may be added for MohrCoulomb. The tensile cutoff is taken to be a fraction of the cohesion. This parameter is specified using:

```
<tensile_cutoff_fraction> 0.1 </tensile_cutoff_fraction>
```

Setting this to a large number effectively removes this failure surface, leaving just Mohr-Coulomb.

3. Material heterogeneity type must be specified. For MohrCoulomb the cohesion is distributed spatially (an independent assignment for each material point). For MaximumPrincipalStress and MaximumPrincipalStrain, the threshold stress or strain for failure, respectively, is distributed spatially (an independent assignment for each material point). Material heterogeneity is distributed spatially by assigning values consistent with a distribution function. Three different distributions may be used. All parameters are in the `<constitutive_model>` block:

```
<failure_distrib> gauss </failure_distrib>
<failure_distrib> weibull </failure_distrib>
<failure_distrib> constant </failure_distrib>
```

A Gaussian (gauss) distribution requires the following parameters:

```
<failure_mean> Gaussian mean value of cohesion </failure_mean>
<failure_std> Gaussian standard deviation of cohesion </failure_std>
<failure_seed> random number generator seed </failure_seed>
```

A Weibull (weibull) distribution requires the following parameters:

```
<failure_mean> Weibull mean value of cohesion </failure_mean>
<failure_std> Weibull modulus </failure_std>
<failure_seed> random number generator seed </failure_seed>
```

A homogeneous (constant) assignment requires the following parameters:

```
<failure_mean> value (all particles assigned one value) </failure_mean>
```

4. Distribution scaling with numerical resolution may optionally be specified. This is only available for Gaussian and Weibull distributions. All parameters are in the `<constitutive_model>` block:

```
<scaling> kayenta </scaling>
<scaling> none (default) </scaling>
```

For kayenta scaling, the mean value of the distribution is scaled by the factor

$$\left(\frac{\bar{V}}{V}\right)^{1/n} \tag{9.22}$$

where $V$ is the particle volume, a function of numerical resolution. The reference volume, $\bar{V}$ and exponent, $n$, both must be specified

```
<reference_volume> $\bar V$ </reference_volume>
<exponent> n </exponent>
```

The exponent defaults to the Weibull modulus if the Weibull distribution is used. This physically motivated scaling provides for an increase in mean cohesion with decreasing particle size, generallly consistent with the observation that smaller quantities of material contain fewer critical flaws.

`<comp_neo_hook>` is a basic elastic model, which calls the underlying `<UCNH>` . The specifications for CNH are:

```
<constitutive_model type="comp_neo_hook">
     <bulk_modulus> 8.9e9  </bulk_modulus>
     <shear_modulus>3.52e9  </shear_modulus>
     <useModifiedEOS> true  </useModifiedEOS>
</constitutive_model>
```

<comp_neo_hook_plastic> as the constitutive model type, tells Uintah to use the basic elastic model extended to include plasticity with isotropic linear hardening, which is equivalent to <usePlasticity> in UCNH. The specifications for CNHP are:

```
<constitutive_model type="comp_neo_hook_plastic">
     <bulk_modulus> 8.9e9     </bulk_modulus>
     <shear_modulus>3.52e9    </shear_modulus>
     <useModifiedEOS> true    </useModifiedEOS>
     <yield_stress>100.0      </yield_stress>
     <hardening_modulus>500.0 </hardening_modulus>
     <alpha>     1.0          </alpha>
</constitutive_model>
```

<cnh_damage> as the constiutive model or <useDamage> tells Uintah to use a basic elastic model, with an extension to failure based on a stress or strain as given below, thus yielding an elastic-brittle failure model. This model also allows a distribution of failure strain (or stress) based on normal or Weibull distributions. Note that the post-failure behaviour of simulations is not always robust.

The specification for CNHD are:

```
<constitutive_model type="cnh_damage">
   <bulk_modulus>   8.9e9  </bulk_modulus>
   <shear_modulus>  3.52e9 </shear_modulus>
   <useModifiedEOS> true   </useModifiedEOS>

</constitutive_model>
```

When specifying <cnh_damage> , the material heterogeneity and damage specification described for the general model (UCNH) may also be specified.

<cnhp_damage> as the constitutive model (or both damage and plasticity flags discussed above) uses an extension to failure based on a stress or strain as given below, thus yielding an elastic-plastic model with failure. Note that the post-failure behaviour of simulations is not always robust. The input section for damage and plasticity is similar to that for UCNH without <useDamage> and <usePlasticity>.

When a particle has failed, the value of the particle variable p.localized will be larger than one (0 means the particle has not failed) and can be output in the DataArchiver section of the input file. In addition, the total number of failed particles as a function of time TotalLocalizedParticle can be output.

Another damage model that can be used with <cnh_damage> and <cnhp_damage> is a subset of the brittle damage model of LS-DYNA's Concrete Model 159 (FHWA-HRT-057-062, 2007). The model is invoked by the following MPMFlag

```
<erosion algorithm="BrittleDamage"/>
```

in the <MPM> section of the input file. Two key features of the model are the use of progressive (as opposed to sudden) damage due to softening to improve numerical stability, and the reduction of mesh size sensitivity via the specification of fracture energy.

Brittle damage occurs when the mean stress $\sigma_{kk}/3$ is tensile and the energy $\tau_b$, related to the maximum principal strain $\epsilon_{max}$, has exceeded a threshold value $r_0^b$

$$\sigma_{kk} > 0, \qquad \tau_b = \sqrt{E\epsilon_{max}^2} \geq r_0^b \tag{9.23}$$

where $E$ is the Young's modulus. If at the next time step the mean stress is less than zero (compressive), the damage mechanism can be optionally inactivated such that the current stress is set temporarily to a fraction of the undamaged stress to model stiffness recovery due to crack closing. When the mean stress becomes tensile again, the value of the previous maximum damage $d$ can be restored; recovery is a user option in Uintah but should be used with caution since stiffening is more prone to instability. The softening function for brittle damage is assumed to be

$$d(\tau_b) = \frac{0.999}{D}\left(\frac{1+D}{1+D\exp^{-C(\tau_b - r_0^b)}}\right) \tag{9.24}$$

where $C$ and $D$ are constants that define the shape of the softening stress-strain curve.

To regulate mesh size sensitivity, the fracture energy ($G_f$), defined as the area under the stress-displacement curve for displacement larger than $x_0$ (the displacement at peak strength), is to be maintained constant. The user needs to input $G_f$ and $D$; $C$ is calculated internally.

The maximum increment of damage that can accumulate over a single time step is a user-defined input to avoid excessive damage accumulation over a single time step to reduce numerical instability.

The brittle damage model is applicable to <cnh_damage> and <cnhp_damage>. For cnh_damage, the parameters for brittle damage can be specified as

```
                <constitutive_model type="cnh_damage">
               <shear_modulus>3.52e9</shear_modulus>
                 <bulk_modulus>8.9e9</bulk_modulus>
   <brittle_damage_initial_threshold>57.0 </brittle_damage_initial_threshold>
  <brittle_damage_fracture_energy>11.2</brittle_damage_fracture_energy>
  <brittle_damage_constant_D>0.1</brittle_damage_constant_D>
 <brittle_damage_max_damage_increment>0.1</brittle_damage_max_damage_increment>
<brittle_damage_allowRecovery> false </brittle_damage_allowRecovery>
<brittle_damage_recoveryCoeff> 1.0 </brittle_damage_recoveryCoeff>
   <brittle_damage_printDamage> false </brittle_damage_printDamage>
                </constitutive_model>
```

The tags in the input file for brittle damage are shown in the following table.

When a particle is damaged, the value of the particle variable p.damage can be output in the DataArchiver section of the input file.

2. **Compressible Mooney-Rivlin Model** This model is generally parameterized for rubber type materials. Usage is as follows:

| Tag | Symbol | Description |
|---|---|---|
| brittle_damage_initial_threshold | $r_0^b$ | material property |
| brittle_damage_fracture_energy | $G_f$ | material property |
| brittle_damage_constant_D | $D$ | material property |
| brittle_damage_max_damage_increment | | optional, default=0.1 |
| brittle_damage_allowRecovery | | allow crack closing (stiffening) optional, default=false |
| brittle_damage_recoveryCoeff | | fraction of undamaged stress to recover (between 0 and 1), optional default=1.0 (full recovery) used only when brittle_damage_allowRecovery is set to true |
| brittle_damage_printDamage | | print the state of damage of damaged particles, default=false (to reduce large amounts of output) |

```
<constitutive_model type="comp_mooney_rivlin">
   <he_constant_1>100000.0</he_constant_1>
   <he_constant_2>20000.0</he_constant_2>
   <he_PR>.49</he_PR>
 </constitutive_model>
```

where `<he_constant_(1,2)>` are usually referred to as $C1$ and $C2$ in the literature.

3. **Kayenta** This is the model formerly known as the Sandia Geomodel. Use is limited to licensees, see Rebecca Brannon for details. It also requires an obscene number of input parameters which are best covered in the users guide for this model. For a simple list, see the source code in `Kayenta.cc`.

4. **Simplified Geomodel** This is a simplified model which has the basic features needed for geomaterials. The yield function of this model is a two-surface plasticity model which includes a linear Drucker-Prager part and a cap yield function. The cap part reflects the fact that plastic deformations can occur even under purely hydrostatic compression as a consequence of void collapse. It means that the simplified geomodel considers the presence of both microscale flaws such as porosity and networks of microcracks. This mdel uses a multi-stage return algorithm proposed in [12]. Usage is as follows:

```
<constitutive_model type="simplified_geo_model">
    <B0>10000</B0>
    <G0>3750</G0>
    <hardening_modulus>0.0</hardening_modulus>
    <FSLOPE> 0.057735026919 </FSLOPE>
    <FSLOPE_p> 0.057735026919 </FSLOPE_p>
    <PEAKI1> 612.3724356953976 </PEAKI1>
    <CR> 6.0 </CR>
    <p0_crush_curve> -1837.0724 </p0_crush_curve>
    <p1_crush_curve> 6.666666666666666e-4 </p1_crush_curve>
    <p3_crush_curve> 0.5 </p3_crush_curve>
    <p4_fluid_effect> 0.2 </p4_fluid_effect>
    <fluid_B0> 0.0 </fluid_B0>
    <fluid_pressur_initial> 0.0 </fluid_pressur_initial>
```

```
    <kinematic_hardening_constant> 0.0 </kinematic_hardening_constant>
</constitutive_model>
```

where <B0> and <G0> are the bulk and shear moduli of the material, <FSLOPE> is the tangent of the friction angle of the Drucker-Prager part, <FSLOPE_p> is the tangent of the dilation angle of the Drucker-Prager part, <hardening_modulus> is the ensemble hardening modulus, and <PEAKI1> is the initial tensile limit of the first stress invariant, $I_1$. The Drucker-Prager yield criterion is given as

$$\sqrt{J_2} + FSLOPE \times (I_1 - PEAKI1) = 0, \tag{9.25}$$

<CR> is a shape parameter that allows porosity to affect shear strength which equals the eccentricity (width divided by height) of the elliptical cap function, <p0_crush_curve> , <p1_crush_curve> , and <p3_crush_curve> are the constants in the fitted post yielding part of the crush curve

$$p_3 - \bar{\epsilon}_v^p = p_3 \exp -3p_1(\bar{p} - p_0) \tag{9.26}$$

in which $\bar{p}$ is the pressure.

In pure kinematic hardening the center of the yield surface changes with its size and shape remaining unchanged. Generally, kinematic hardening is modeled by introducing the back stress tensor, and defining an appropriate evolution rule for it. In the simplified geomodel, linear Zieglers rule is used:

$$\dot{\alpha}_{ij} = \dot{\mu}(\boldsymbol{\sigma}_{ij} - \alpha_{ij}) \tag{9.27}$$

in which $\alpha_{ij}$ is the back stress tensor, $\dot{\alpha}_{ij}$ is time derivative of the back stress tensor, and

$$\dot{\mu} = c\dot{\xi}^p \tag{9.28}$$

where $\dot{\xi}^p$ is the deviatoric invariant of the rate of plastic strain and $c$ is a constant defined yb the user as <kinematic_hardening_constant> .

Based on the research work done by M. Homel at the University of Utah, the following equations are used in the simplified geomodel to consider the fluid-filled porous effects:

$$\begin{aligned}\frac{\partial X}{\partial \varepsilon_v^p} = \quad & \frac{1}{p_1 p_3} \exp\left(-p_1 X - p_0\right) - \frac{3K_f\left(\exp(p_3+p_4)-1\right)\exp(p_3+p_4+\varepsilon_v^p)}{\left(\exp(p_3+p_4+\varepsilon_v^p)-1\right)^2} \\ & + \frac{3K_f\left(\exp(p_3+p_4)-1\right)\exp(p_3+\varepsilon_v^p)}{\left(\exp(p_3+\varepsilon_v^p)-1\right)^2}\end{aligned} \tag{9.29}$$

in which $X$ is the value of the first stress invariant at the intersection of the cap yield surface and the mean pressure axis, $K_f$ is the fluid bulk modulus, which is defined by the user as <fluid_B0>, $\varepsilon_v^p$ is the volumetric part of the plastic strain, and $p_4$ is a constant defined by the user as <p4_fluid_effect>. The isotropic part of the back stress tensor is updated using the following equation

$$\alpha_{n+1}^{\text{iso}} = \alpha_n^{\text{iso}} + \frac{3K_f \exp(p_3)\left(\exp(p_4) - \exp(\varepsilon_v^p)\right)}{\left(\exp(p_3+\varepsilon_v^p)-1\right)} \dot{\varepsilon}_v^p \Delta t \ \boldsymbol{1} \tag{9.30}$$

in which $\boldsymbol{1}$ is the second-order identity tensor. Also, the effective bulk modulus is calculated as

$$K_e = B0 + \frac{K_f\left(\exp(p_3+p_4)-1\right)\exp(p_3+p_4+\varepsilon_v^e+\varepsilon_v^p)}{\left(\exp(p_3+p_4+\varepsilon_v^e+\varepsilon_v^p)-1\right)^2} \tag{9.31}$$

in which $\varepsilon_v^e$ is the volumetric part of the elastic strain.

5. **Water** This is a model for water, reported in [18]. The P-V relationship is given by:

$$p = \kappa \left[ \left( \frac{\rho}{\rho_0} \right)^{\gamma} - 1 \right] \tag{9.32}$$

   Shear stress is simple Newtonian behavior. It has not been validated, but gives qualitatively reasonable behavior. Usage is given by:

```
<constitutive_model type="water">
   <bulk_modulus>15000.0</bulk_modulus>
   <viscosity>.5</viscosity>
   <gamma>7.0</gamma>
 </constitutive_model>
```

6. **Ideal Gas** This is simply an equation of state, with no shear stress. Usage is given by:

```
<constitutive_model type="ideal_gas">
    <specific_heat>  717.5        </specific_heat>
    <gamma>  1.4        </gamma>
</constitutive_model>
```

7. **Rigid Material** This model was designed for use with the `specified` contact model described in Section 9.5.7. It is designed to compute zero stress and deformation of the material, and is basically a fast place holder for materials that won't be developing a stress anyway.

8. **ElasticPlastic** The `<elastic_plastic>` model is a general purpose model that was primarily implemented for the purpose of modeling high strain rate metal plasticity. Dr. Biswajit Banerjee has written an extensive description of the theory, implementation and use of this model. Because of the amount of detail involved, and because these subtopics are interwoven, this model is given its own section below.

There is a large number remaining models but these are not frequently utilized. This includes models for viscoelasticity, soil models, and transverse isotropic materials (i.e., fiber reinforced composites). Examples of their use can be found in the `inputs` directory. Input files can also be constructed by checking the source code to see what parameters are required.

There are a few models whose use is explicitly not recommended. In particular, `HypoElasticPlastic, Membrane and SmallStrainPlastic.` Input files calling for the first of these should be switched to the `ElasticPlastic` model instead.

### 9.5.6 Hypo-Elastic Plasticity in Uintah

The hypoelastic-plastic stress update is a mix and match combination of isotropic models. The equation of state may be varied independently of the deviatoric response. For elastic deviatoric response the shear moduli may be taken to be functions of temperature and pressure. Plasticity is based on an additive decomposition of the rate of deformation tensor into elastic and plastic parts. Incompressibility is assumed for plastic deformations, i.e. the plastic strain rate is proportional to the deviatoric stress. Various yield conditions and flow stresses may be mixed and matched. There are also options for damage/melting modeling. *Note that there are few checks to prevent users from mixing and matching inappropriate models.*

Additional models can be added to this framework. Presently, the material models available are:

1. Adiabatic Heating and Specific Heat:
    - Taylor-Quinney coefficient.
    - Constant Specific Heat (**default**).
    - Cubic Specific Heat.
    - Specific Heat for Copper.
    - Specific Heat for Steel.
2. The equation of state (pressure/volume response):
    - Hypoelastic (**default**).
    - Neo-Hookean.
    - Mie-Gruneisen.
3. The deviatoric stress model:
    - Linear hypoelasticity (**default**).
    - Linear hypoviscoelasticity.
4. The melting model:
    - Constant melt temperature (**default**).
    - Linear melt temperature.
    - Steinberg-Cochran-Guinan (SCG) melt.
    - Burakovsky-Preston-Silbar (BPS) melt.
5. Temperature and pressure dependent shear moduli (only works with linear hypoelastic deviatoric stress model):
    - Constant shear modulus (**default**).
    - Mechanical Threshold Stress (MTS) model.
    - Steinberg-Cochran-Guinan (SCG) model.
    - Nadal-LePoac (NP) model.
    - Preston-Tonks-Wallace (PTW) model.
6. The yield condition:
    - von Mises.
    - Gurson-Tvergaard-Needleman (GTN).
7. The flow stress:
    - the Isotropic Hardening model
    - the Johnson-Cook (JC) model
    - the Steinberg-Cochran-Guinan-Lund (SCG) model.
    - the Zerilli-Armstrong (ZA) model.

- the Zerilli-Armstrong for polymers model.
- the Mechanical Threshold Stress (MTS) model.
- the Preston-Tonks-Wallace (PTW) model.

8. The plastic return algorithm:
   - Radial Return (**default**).
   - Modified Nemat-Nasser/Maudlin.
9. The damage model:
   - Johnson-Cook damage model.

This model is invoked using

```
<constitutive_model="elastic_plastic_hp">
  <shear_modulus>0.2845e4</shear_modulus>
  <bulk_modulus>1.41e4</bulk_modulus>
  <initial_material_temperature>298</initial_material_temperature>
  <plastic_convergence_algo>radialReturn</plastic_convergence_algo>
  <taylor_quinney_coeff> 0.9 </taylor_quinney_coeff>

  submodels

</constitutive_model>
```

where "submodels" indicates subsets of tags corresponding to the listed above (and detailed below). *Note that the specified bulk and shear moduli are used to calcuate a stable time step size for the first time step (hence it is important that they be consistent with EOS and deviatoric stress submodel material constants). However, if the default EOS and/or deviatoric stress models are used, then these material constants are sufficient for bulk and/or deviatoric stress response, and are automatically used in those models.* The bulk modulus is also used to determine artificial viscosity parameters (throughout the simulation).

**Adiabatic Heating and Specific Heat** A part of the plastic work done is converted into heat and used to update the temperature of a particle. The increase in temperature ($\Delta T$) due to an increment in plastic strain ($\Delta\epsilon_p$) is given by the equation

$$\Delta T = \frac{\chi\sigma_y}{\rho C_p}\Delta\epsilon_p \tag{9.33}$$

where $\chi$ is the Taylor-Quinney coefficient, and $C_p$ is the specific heat. The value of the Taylor-Quinney coefficient is taken to be 0.9 in all our simulations (see [48] for more details on the variation of $\chi$ with strain and strain rate).

The Taylor-Quinney coefficient is taken as input in the ElasticPlastic model using the tags

```
<taylor_quinney_coeff> 0.9 </taylor_quinney_coeff>
```

**Default specific heat model** The default model returns a constant specific heat and is invoked using

```
<specific_heat_model type="constant_Cp">
</specific_heat_model>
```

**Cubic specific heat model** The specific heat model is of the form [39]:

$$C_v = \frac{\tilde{T}^3}{c_3\tilde{T}^3 + c_2\tilde{T}^2 + c_1\tilde{T} + c_0} \tag{9.34}$$

where $\tilde{T}$ is the reduced temperature, and $c_0$-$c_3$ are fit parameters. The reduced temperature is calculated using $\tilde{T} = T/\theta(V)$ and the Debye temperature is:

$$\theta(V) = \theta_0 \left(\frac{V_0}{V}\right)^a e^{b(V_0-V)/V} \tag{9.35}$$

where $V$ is the specific volume and $\theta_0$ is the reference Debye temperature and a and b are fit parameters. The constant pressure specific heat is calculated via:

$$C_p = C_v + \beta^2 TVK_T \tag{9.36}$$

where $\beta$ and $K_T$ are the volumetric expansion coefficeint and isothermal bulk modulus respectively.

The model is invoked using:

```
<specific_heat_model type="cubic_Cp">
  <a> 1.0 </a>
  <b> 1.0 </b>
  <beta> 1.0 </beta>
  <c0> 1.0 </c0>
  <c1> 1.0 </c1>
  <c2> 1.0 </c2>
  <c3> 1.0 </c3>
</specific_heat_model>
```

where all parameters but $\beta$, $a$ and $b$ are required.

**Specific heat model for copper** The specific heat model for copper is of the form

$$C_p = \begin{cases} A_0\ T^3 - B_0\ T^2 + C_0\ T - D_0 & \text{if } T < T_0 \\ A_1\ T + B_1 & \text{if } T \geq T_0\ . \end{cases} \tag{9.37}$$

The model is invoked using

```
<specific_heat_model type = "copper_Cp"> </specific_heat_model>
```

**Specific heat model for steel** A relation for the dependence of $C_p$ upon temperature is used for the steel ([36]).

$$C_p = \begin{cases} A_1 + B_1\ t + C_1\ |t|^{-\alpha} & \text{if } T < T_c \\ A_2 + B_2\ t + C_2\ t^{-\alpha'} & \text{if } T > T_c \end{cases} \tag{9.38}$$

$$t = \frac{T}{T_c} - 1 \tag{9.39}$$

where $T_c$ is the critical temperature at which the phase transformation from the $\alpha$ to the $\gamma$ phase takes place, and $A_1, A_2, B_1, B_2, \alpha, \alpha'$ are constants.

The model is invoked using

```
<specific_heat_model type = "steel_Cp"> </specific_heat_model>
```

The heat generated at a material point is conducted away at the end of a time step using the transient heat equation. The effect of conduction on material point temperature is negligible (but non-zero) for the high strain-rate problems simulated using Uintah.

**Equation of State Models** The elastic-plastic stress update assumes that the volumetric part of the Cauchy stress can be calculated using an equation of state. There are three equations of state that are implemented in Uintah. These are

1. A default hypoelastic equation of state.
2. A neo-Hookean equation of state.
3. A Mie-Gruneisen type equation of state.

**Default hypoelastic equation of state** In this case we assume that the stress rate is given by

$$\dot{\boldsymbol{\sigma}} = \lambda \ \mathrm{tr}(\boldsymbol{d}^e) \ \boldsymbol{1} + 2 \ \mu \ \boldsymbol{d}^e \tag{9.40}$$

where $\boldsymbol{\sigma}$ is the Cauchy stress, $\boldsymbol{d}^e$ is the elastic part of the rate of deformation, and $\lambda, \mu$ are constants.

If $\boldsymbol{\eta}^e$ is the deviatoric part of $\boldsymbol{d}^e$ then we can write

$$\dot{\boldsymbol{\sigma}} = \left(\lambda + \frac{2}{3} \ \mu\right) \ \mathrm{tr}(\boldsymbol{d}^e) \ \boldsymbol{1} + 2 \ \mu \ \boldsymbol{\eta}^e = \kappa \ \mathrm{tr}(\boldsymbol{d}^e) \ \boldsymbol{1} + 2 \ \mu \ \boldsymbol{\eta}^e \ . \tag{9.41}$$

If we split $\boldsymbol{\sigma}$ into a volumetric and a deviatoric part, i.e., $\boldsymbol{\sigma} = p \ \boldsymbol{1} + \boldsymbol{s}$ and take the time derivative to get $\dot{\boldsymbol{\sigma}} = \dot{p} \ \boldsymbol{1} + \dot{\boldsymbol{s}}$ then

$$\dot{p} = \kappa \ \mathrm{tr}(\boldsymbol{d}^e) \ . \tag{9.42}$$

In addition we assume that $\boldsymbol{d} = \boldsymbol{d}^e + \boldsymbol{d}^p$. If we also assume that the plastic volume change is negligible, we can then write that

$$\dot{p} = \kappa \ \mathrm{tr}(\boldsymbol{d}) \ . \tag{9.43}$$

This is the equation that is used to calculate the pressure $p$ in the default hypoelastic equation of state, i.e.,

$$\boxed{p_{n+1} = p_n + \kappa \ \mathrm{tr}(\boldsymbol{d}_{n+1}) \ \Delta t \ .} \tag{9.44}$$

To get the derivative of $p$ with respect to $J$, where $J = \det(\boldsymbol{F})$, we note that

$$\dot{p} = \frac{\partial p}{\partial J} \ \dot{J} = \frac{\partial p}{\partial J} \ J \ \mathrm{tr}(\boldsymbol{d}) \ . \tag{9.45}$$

Therefore,

$$\boxed{\frac{\partial p}{\partial J} = \frac{\kappa}{J} \, .} \tag{9.46}$$

This model is invoked in Uintah using

```
<equation_of_state type="default_hypo">
</equation_of_state>
```

The code is in `.../MPM/ConstitutiveModel/PlasticityModels/DefaultHypoElasticEOS.cc` If an EOS is not specified then this model is the **default**.

**Default hyperelastic equation of state** In this model the pressure is computed using the relation

$$p = \frac{1}{2}\ \kappa\ \left(J^e - \frac{1}{J^e}\right) \tag{9.47}$$

where $\kappa$ is the bulk modulus and $J^e$ is determinant of the elastic part of the deformation gradient.

We can also compute

$$\frac{dp}{dJ} = \frac{1}{2}\ \kappa\ \left(1 + \frac{1}{(J^e)^2}\right)\ . \tag{9.48}$$

This model is invoked in Uintah using

```
<equation_of_state type="default_hyper">
</equation_of_state>
```

The code is in `.../MPM/ConstitutiveModel/PlasticityModels/HyperElasticEOS.cc`.

**Mie-Gruneisen equation of state** The pressure ($p$) is calculated using a Mie-Grüneisen equation of state

$$p = p_{ref} + \rho\Gamma(e - e_{ref}) \tag{9.49}$$

where $\rho$ is the mass density, $\Gamma$ the Grüneisen parameter (unitless) and $p_{ref}$ and $e_{ref}$ are known pressure and internal specific energy on a reference curve *and are a function of volume only.* As the form can be formally viewed as an expansion valid near the reference curve, ideally the reference curve prescribes states near those of interest. The reference curve could be the shock Hugoniot, the standard adiabat (through the initial state), the 0 K isotherm, the isobar $p = 0$, the curve $e = 0$, or some composite of these curves to cover the range of interest.

For shock calculations it makes sense to use the Hugoniot as a reference curve. We Assume the following relationship between shock wave velocity, $U_s$ and particle velocity, $U_p$,

$$U_s = C_0 + s_\alpha U_p + s_2\frac{U_p^2}{U_s} + s_3\frac{U_p^3}{U_s^2} \tag{9.50}$$

where $C_0$ is the bulk speed of sound, and the $s$'s are dimensionless coefficients. This form is due to Steinberg ([52]), and is a straight–forward extension to a nonlinear shock velocity, particle velocity relationship. It reduces to the linear relationship most frequently used, e.g. ([63, 68]), with $s_2 = s_3 = 0$. Using the steady shock jump conditions for conservation of mass, momentum and energy, the Hugoniot reference pressure, $p_H$ and specific energy, $e_H$ may be determined

$$p_H = \frac{\rho_0 C_0^2 \eta}{1 - s_\alpha\eta - s_2\eta^2 - s_3\eta^3} \tag{9.51}$$

$$e_H = \frac{p_H \eta}{2\rho_0} \tag{9.52}$$

where $\rho_0$ is the initial density (pre-shock) and

$$\eta = 1 - \frac{\rho_0}{\rho} \tag{9.53}$$

is a measure of volumetric deformation. Using these relationships and the additional assumption that $\rho\Gamma = \rho_0\Gamma_0$ the Mie-Grüneisen equation of state may be written

$$\boxed{p = \frac{\rho_0\ C_0^2\ \eta(1 - \frac{\Gamma_0\eta}{2})}{(1 - s_\alpha\eta - s_2\eta^2 - s_3\eta^3)^2} + \rho_0\Gamma_0\ e} \tag{9.54}$$

To extend the Mie Grüneisen EOS into tensile stress regimes, for $\eta < 0$ the pressure is evaluated as

$$\boxed{p = \rho_0 \; C_0^2 \; \eta + \rho_0 \Gamma_0 \; e} \tag{9.55}$$

This equation is integrated explicitly, using beginning timestep values for energy and the current value of density to update the pressure. For isochoric plasticity,

$$J^e = J = \det(\boldsymbol{F}) = \frac{\rho_0}{\rho} \, .$$

where $\boldsymbol{F}^e$ is the elastic part of the deformation gradient. The increment in specific internal energy is computed using

$$\rho^* \Delta e = (\sigma^*_{ij} D_{ij} - q D_{kk}) \Delta t \tag{9.56}$$

where $\sigma^*_{ij}$ and $\rho^*$ are the average stress and density over the time step, $D_{ij}$ is the rate of deformation tensor, and $q$ is the artificial viscosity. Note that the artificial velocity term must be included explicitly since it is not accumulated in the total stress.

The temperature, $T$, is calculated using the thermodynamic relationship

$$dT = -\rho \Gamma T dv + \frac{T dS}{C_v} \tag{9.57}$$

where $v$ is the specific volume, $S$ the entropy and $C_v$ the specific heat at constant volume. Entropy change is associated with irreversible, or dissipative, processes. Equating $TdS$ to the dissipated work terms, those components of temperature change are computed in the appropriate routines, such as plasticity or artificial viscosity. In fact, not all of the dissipated energy needs be converted to heat, as allowed for by using the Taylor–Quinney coefficient (see below). The first term in may be integrated to give the isentropic temperature change

$$\Delta T_{isentropic} = -T \Gamma_0 \frac{\rho_0}{\rho} D_{kk} \Delta t \tag{9.58}$$

where the same assumption, $\rho\Gamma = \rho_0 \Gamma_0$ is used. The isentropic temperature change is computed as part of the EOS response.

Should an implicit integration scheme be used the tangent moduli are needed, which in turn require calculation of

$$\boxed{\frac{\partial p}{\partial J^e} = \frac{\rho_0 \; C_0^2 \; [1 + (S_\alpha - \Gamma_0) \; (1 - J^e)]}{[1 - S_\alpha \; (1 - J^e)]^3} - \Gamma_0 \; \frac{\partial e}{\partial J^e}.} \tag{9.59}$$

We neglect the $\frac{\partial e}{\partial J^e}$ term in our calculations. *Note: this calculation hasn't been updated for the Steinberg nonlinear shock velocity, particle velocity relationship.*

This model is invoked in Uintah using

```
<equation_of_state type="mie_gruneisen">
  <C_0>5386</C_0>
  <Gamma_0>1.99</Gamma_0>
  <S_alpha>1.339</S_alpha>
  <S_2>1.339</S_2>
  <S_3>1.339</S_3>
</equation_of_state>
```

The code is in `.../MPM/ConstitutiveModel/PlasticityModels/MieGruneisenEOS.cc`.

It is worth noting that this approach to calculating energy and temperature is not necessarily consistent with existing implementations. In fact, it does not appear that there is a standard approach, many shock codes have unique implementations. In particular, it appears that the elastic stored energy term is often neglected, as well as the isentropic temperature change.

It is worth noting that the approach outlined above is consistent with that taken fairly recently by Wilkins ([63]). Wilkins expands the pressure and energy as polynomials in $\eta$ and uses Hugoniot data (and a linear $U_s$, $U_p$ relationship) to determine the coefficients. Using the additional thermodynamic relationship

$$de = -pdv + TdS \tag{9.60}$$

and substituting for $TdS$ from 9.57, assuming $C_v$ constant and $\rho\Gamma = \rho_0\Gamma_0$ (as in Wilkins), the following relationship may be derived

$$e = -\int_{v_0}^{v}(p - \rho_0\Gamma_0 C_v T)dv + C_v(T - T_0) \tag{9.61}$$

Since the integral is for $dS = 0$, it may be integrated to give an alternate form of 9.58,

$$T_{isentropic} = T_0 \exp(\Gamma_0(1 - \frac{\rho_0}{\rho})) \tag{9.62}$$

which may be substituted into 9.61 along with the expansion for $p(\eta)$ (here for $s_2 = s_3 = 0$)

$$p = \rho_0\Gamma_0 e + \rho_0 C_0^2(\eta + (2s_\alpha - \frac{\Gamma_0}{2})\eta^2 + s_\alpha(3s_\alpha - \Gamma_0)\eta^3) + O(\eta^4) \tag{9.63}$$

to give the equation

$$e + \int_{v_0}^{v}\rho_0\Gamma_0 e dv = C_v(T - T_0) + \rho_0\Gamma_0 C_v T_0\int_{v_0}^{v}\exp(\Gamma_0\eta)dv - \rho_0\Gamma_0\int_{v_0}^{v}(\eta + (2s_\alpha - \frac{\Gamma_0}{2})\eta^2 + s_\alpha(3s_\alpha - \Gamma_0)\eta^3)dv \tag{9.64}$$

Finally, using

$$e = e_0(\eta) + \int_{T_0}^{T} C_v dT \tag{9.65}$$

with a polynomial expansion for $e(\eta)$ in powers of $\eta$, 9.64 can be integrated to determine the coefficients in the expansion. Determining the coefficients this way gives exactly the same expansion for energy as derived in Wilkins, using a different approach. The advantages of the approach outlined above are it's relative simplicity, and generality – no assumption of constant specific heat is needed.

**Deviatoric Stress Models** The elastic-plastic stress update assumes that the deviatoric part of the Cauchy stress can be calculated independently of the equation of state. There are two deviatoric stress models that are implemented in Uintah. These are

1. A default hypoelastic deviatoric stress.
2. A linear hypoviscooelastic deviatoric stress.

**Default Hypoelastic Deviatoric Stress** In this case the stress rate is given by

$$\dot{\boldsymbol{s}} = 2\mu(\boldsymbol{\eta} - \boldsymbol{\eta}^p) \tag{9.66}$$

where $\mu$ is the shear modulus. This model is invoked using

```
<deviatoric_stress_model type="hypoElastic">
</deviatoric_stress_model>
```

If a deviatoric stress model is not specified then this model is the **default**.

**Linear Hypoviscoelastic Deviatoric Stress** This model is a three–dimensional version of a Generalized Maxwell model, as presented in [67]. It is specifically implemented to be combined with the ZA for Polymers Flow Stress Model described previously. Together these models combine into a hypoviscoplastic model. The stress update is given by

$$\dot{\boldsymbol{s}} = 2\mu(\boldsymbol{\eta} - \boldsymbol{\eta}^p) - \sum_{i=1}^{N} \frac{\boldsymbol{s}_i}{\tau_i} \tag{9.67}$$

where $\mu$ is the shear modulus and $\boldsymbol{s}_i$ are maxwell element stresses which are tracked internally. Also

$$\boldsymbol{s} = \sum_{i=1}^{N} \boldsymbol{s}_i \qquad \mu = \sum_{i=1}^{N} \mu_i \tag{9.68}$$

This model is invoked using

```
<deviatoric_stress_model type="hypoViscoElastic">
  <mu> [3.0, 5.0, 7.0] </mu>
  <tau> [1.67, 10.7, 107.0] </tau>
</deviatoric_stress_model>
```

where the number of elements in arrays mu and tau must be the same.

### Melting Temperature

**Default model** The default model is to use a constant melting temperature. This model is invoked using

```
<melting_temp_model type="constant_Tm">
</melting_temp_model>
```

**Linear melt model** A melting temperature model designed to be linear in pressure can be invoked using

```
<melting_temp_model type="linear_Tm">
  <T_m0> </T_m0>
  <a>   </a>
  <b> </b>
  <Gamma_0> </Gamma_0>
  <K_T> </K_T>
</melting_temp_model>
```

where T_m0 is required, and either a or Gamma_0 along with K_T, or b. T_m0 is the initial melting temperature in Kelvin, a is the Kraut-Kennedy coefficient, Gamma_0 is the Gruniesen Gamma, b is the pressure coefficient in Kelvin per Pascal, and K_T is the isothermal bulk modulus in Pascals. The pressure is calculated using either

$$T_m = T_{m0} + bP \tag{9.69}$$

or

$$T_m = T_{m0}\left(1 + a\frac{\rho_0}{\rho}\right) \tag{9.70}$$

The constants in these equations are linked together via the following equation

$$b = \frac{aT_{m0}}{K_T} \tag{9.71}$$

and a is related to the Gruneisen Gamma by the Lindemann Law [44]:

$$a = 2\left(\Gamma_0 - \frac{1}{3}\right) \tag{9.72}$$

**SCG melt model** We use a pressure dependent relation to determine the melting temperature ($T_m$). The Steinberg-Cochran-Guinan (SCG) melt model ([53]) has been used for our simulations of copper. This model is based on a modified Lindemann law and has the form

$$T_m(\rho) = T_{m0}\exp\left[2a\left(1 - \frac{1}{\eta}\right)\right]\eta^{2(\Gamma_0 - a - 1/3)}; \quad \eta = \frac{\rho}{\rho_0} \tag{9.73}$$

where $T_{m0}$ is the melt temperature at $\eta = 1$, $a$ is the coefficient of the first order volume correction to Grüneisen's gamma ($\Gamma_0$).

This model is invoked with

```
<melting_temp_model type="scg_Tm">
  <T_m0> 2310.0 </T_m0>
  <Gamma_0> 3.0 </Gamma_0>
  <a> 1.67 </a>
</melting_temp_model>
```

**BPS melt model** An alternative melting relation that is based on dislocation-mediated phase transitions - the Burakovsky-Preston-Silbar (BPS) model ([14]) can also be used. This model has been used to determine the melt temperature for 4340 steel. The BPS model has the form

$$T_m(p) = T_m(0)\left[\frac{1}{\eta} + \frac{1}{\eta^{4/3}}\,\frac{\mu_0^{'}}{\mu_0}\,p\right] \;; \quad \eta = \left(1 + \frac{K_0^{'}}{K_0}\,p\right)^{1/K_0^{'}} \tag{9.74}$$

$$T_m(0) = \frac{\kappa\lambda\mu_0\ v_{WS}}{8\pi\ln(z-1)\ k_b}\ln\left(\frac{\alpha^2}{4\ b^2\rho_c(T_m)}\right) \tag{9.75}$$

where $p$ is the pressure, $\eta = \rho/\rho_0$ is the compression, $\mu_0$ is the shear modulus at room temperature and zero pressure, $\mu_0^{'} = \partial\mu/\partial p$ is the derivative of the shear modulus at zero pressure, $K_0$ is the bulk modulus at room temperature and zero pressure, $K_0^{'} = \partial K/\partial p$ is the derivative of the bulk

modulus at zero pressure, $\kappa$ is a constant, $\lambda = b^3/v_{WS}$ where $b$ is the magnitude of the Burgers' vector, $v_{WS}$ is the Wigner-Seitz volume, $z$ is the coordination number, $\alpha$ is a constant, $\rho_c(T_m)$ is the critical density of dislocations, and $k_b$ is the Boltzmann constant.

This model is invoked with

```
<melting_temp_model type="bps_Tm">
  <B0> 137e9 </B0>
  <dB_dp0> 5.48 <dB_dp0>
  <G0> 47.7e9 <G0>
  <dG_dp0> 1.4 <dG_dp0>
  <kappa> 1.25 <kappa>
  <z> 12 <z>
  <b2rhoTm> 0.64 <b2rhoTm>
  <alpha> 2.9 <alpha>
  <lambda> 1.41 <lambda>
  <a> 3.6147e-9<a>
  <v_ws_a3_factor> 1/4 <v_ws_a3_factor>
  <Boltzmann_Constant> <Boltzmann_Constant>
</melting_temp_model>
```

**Shear Modulus** Three models for the shear modulus ($\mu$) have been tested in our simulations. The first has been associated with the Mechanical Threshold Stress (MTS) model and we call it the MTS shear model. The second is the model used by Steinberg-Cochran-Guinan and we call it the SCG shear model while the third is a model developed by Nadal and Le Poac that we call the NP shear model.

**Default model** The default model gives a constant shear modulus. The model is invoked using

```
<shear_modulus_model type="constant_shear">
</shear_modulus_model>
```

**MTS Shear Modulus Model** The simplest model is of the form suggested by [60] ([15])

$$\mu(T) = \mu_0 - \frac{D}{exp(T_0/T) - 1} \tag{9.76}$$

where $\mu_0$ is the shear modulus at 0K, and $D, T_0$ are material constants.

The model is invoked using

```
<shear_modulus_model type="mts_shear">
  <mu_0>28.0e9</mu_0>
  <D>4.50e9</D>
  <T_0>294</T_0>
</shear_modulus_model>
```

**SCG Shear Modulus Model** The Steinberg-Cochran-Guinan (SCG) shear modulus model ([53, 68]) is pressure dependent and has the form

$$\mu(p,T) = \mu_0 + \frac{\partial \mu}{\partial p}\frac{p}{\eta^{1/3}} + \frac{\partial \mu}{\partial T}(T - 300); \quad \eta = \rho/\rho_0 \tag{9.77}$$

where, $\mu_0$ is the shear modulus at the reference state($T = 300$ K, $p = 0$, $\eta = 1$), $p$ is the pressure, and $T$ is the temperature. When the temperature is above $T_m$, the shear modulus is instantaneously set to zero in this model.

The model is invoked using

```
<shear_modulus_model type="scg_shear">
  <mu_0> 81.8e9 </mu_0>
  <A> 20.6e-12 </A>
  <B> 0.16e-3 </B>
</shear_modulus_model>
```

**NP Shear Modulus Model** A modified version of the SCG model has been developed by [40] that attempts to capture the sudden drop in the shear modulus close to the melting temperature in a smooth manner. The Nadal-LePoac (NP) shear modulus model has the form

$$\mu(p,T) = \frac{1}{\mathcal{J}(\widehat{T})}\left[\left(\mu_0 + \frac{\partial\mu}{\partial p}\frac{p}{\eta^{1/3}}\right)(1-\widehat{T}) + \frac{\rho}{Cm}\, k_b\, T\right]; \quad C := \frac{(6\pi^2)^{2/3}}{3} f^2 \tag{9.78}$$

where

$$\mathcal{J}(\widehat{T}) := 1 + \exp\left[-\frac{1+1/\zeta}{1+\zeta/(1-\widehat{T})}\right] \quad \text{for} \quad \widehat{T} := \frac{T}{T_m} \in [0, 1+\zeta], \tag{9.79}$$

$\mu_0$ is the shear modulus at 0 K and ambient pressure, $\zeta$ is a material parameter, $k_b$ is the Boltzmann constant, $m$ is the atomic mass, and $f$ is the Lindemann constant.

The model is invoked using

```
<shear_modulus_model type="np_shear">
  <mu_0>26.5e9</mu_0>
  <zeta>0.04</zeta>
  <slope_mu_p_over_mu0>65.0e-12</slope_mu_p_over_mu0>
  <C> 0.047 </C>
  <m> 26.98 </m>
</shear_modulus_model>
```

**PTW Shear model** The PTW shear model is a simplified version of the SCG shear model. The inputs can be found in `.../MPM/ConstitutiveModel/PlasticityModel/PTWShear.h`.

**Yield conditions** When failure is to be simulated we can use the Gurson-Tvergaard-Needleman yield condition instead of the von Mises condition.

**The von Mises yield condition** The von Mises yield condition is the default. It specifies a yield condition of the form

$$\Phi = \sqrt{3J_2} - \sigma_y \tag{9.80}$$

where $J_2$ is the second invariant of the deviatoric stress tensor ($J_2 = \frac{1}{2}\boldsymbol{s} : \boldsymbol{s}$) and $\sigma_y$ is the flow stress. Currently the return algorithms are restricted to plastic flow in the direction of the deviatoric stress (Eqn. 9.116). See the discussion in the Radial Return algorithm description for details. The von Mises yield condition is invoked using the tags

```
<yield_condition type="vonMises">
</yield_condition>
```

**The Gurson-Tvergaard-Needleman (GTN) yield condition** The Gurson-Tvergaard-Needleman (GTN) yield condition [25, 58] depends on porosity. *This model is for experts only!!!* Here are some caveats: Formally, you can replace the flow stress in Gursons model with the flow stresses of Johnson-Cook, ZA, etc., but the internal variable updates would have to be modified extensively. For example, the JC yield stress depends on the equivalent plastic strain, but this needs to be the equivalent plastic strain of the matrix material, which is very different from the equivalent plastic strain of the porous composite. For example, under pure hydrostatic compression at the macroscale, the matrix material will suffer massive amounts of plastic SHEAR strains at the microscale (even though, for hydrostatic loading, it has zero plastic shear strain at the macroscale) and thus would need to harden. While the models should run, they are unlikely to give realistic results.

The GTN yield condition is a fairly good bound in compression but a TERRIBLE bound in tension (in fact using it in tension can produce non-physical predictions of negative plastic work tantamount to tension causing pore COLLAPSE; very few Gurson implementations catch this problem because very few of them include run-time checks of solution quality, including this one).

The GTN yield condition will not work with Radial Return. An error will be generated in this case. Presently it only runs with the modified Nemat-Nasser/Maudlin return algorithm. Plastic flow is assumed to be in the direction of deviatoric stress. Hence this is nonassociated flow for this pressure dependent yield condition.

The GTN yield condition can be written as

$$\Phi = \left(\frac{\sigma_{eq}}{\sigma_f}\right)^2 + 2q_1 f_* \cosh\left(q_2 \frac{Tr(\sigma)}{2\sigma_f}\right) - (1 + q_3 f_*^2) = 0 \tag{9.81}$$

where $q_1, q_2, q_3$ are material constants and $f_*$ is the porosity (damage) function given by

$$f* = \begin{cases} f & \text{for } \; f \leq f_c, \\ f_c + k(f - f_c) & \text{for } \; f > f_c \end{cases} \tag{9.82}$$

where $k$ is a constant and $f$ is the porosity (void volume fraction). The flow stress in the matrix material is computed using either of the two plasticity models discussed earlier. Note that the flow stress in the matrix material also remains on the undamaged matrix yield surface and uses an associated flow rule.

This yield condition is invoked using

```
<yield_condition type="gurson">
  <q1> 1.5 </q1>
  <q2> 1.0 </q2>
  <q3> 2.25 </q3>
  <k> 4.0 </k>
  <f_c> 0.05 </f_c>
</yield_condition>
```

**Porosity model** The evolution of porosity is calculated as the sum of the rate of growth and the rate of nucleation [47]. The rate of growth of porosity and the void nucleation rate are given

by the following equations [16]

$$\dot{f} = \dot{f}_{\text{nucl}} + \dot{f}_{\text{grow}} \tag{9.83}$$

$$\dot{f}_{\text{grow}} = (1-f)\text{Tr}(\boldsymbol{D}_p) \tag{9.84}$$

$$\dot{f}_{\text{nucl}} = \frac{f_n}{(s_n\sqrt{2\pi})}\exp\left[-\frac{1}{2}\frac{(\epsilon_p-\epsilon_n)^2}{s_n^2}\right]\dot{\epsilon}_p \tag{9.85}$$

where $\boldsymbol{D}_p$ is the rate of plastic deformation tensor, $f_n$ is the volume fraction of void nucleating particles , $\epsilon_n$ is the mean of the distribution of nucleation strains, and $s_n$ is the standard deviation of the distribution.

The inputs tags for porosity are of the form

```
<evolve_porosity> true </evolve_porosity>
<initial_mean_porosity>         0.005 </initial_mean_porosity>
<initial_std_porosity>          0.001 </initial_std_porosity>
<critical_porosity>             0.3   </critical_porosity>
<frac_nucleation>               0.1   </frac_nucleation>
<meanstrain_nucleation>         0.3   </meanstrain_nucleation>
<stddevstrain_nucleation>       0.1   </stddevstrain_nucleation>
<initial_porosity_distrib>      gauss </initial_porosity_distrib>
```

**Flow Stress** We have explored seven temperature and strain rate dependent models that can be used to compute the flow stress. Some of these are also pressure dependent (note that plastic flow does is non-associative for pressure dependent models):

1. the Isotropic Hardening model
2. the Johnson-Cook (JC) model
3. the Steinberg-Cochran-Guinan-Lund (SCG) model.
4. the Zerilli-Armstrong (ZA) model.
5. the Zerilli-Armstrong for polymers model.
6. the Mechanical Threshold Stress (MTS) model.
7. the Preston-Tonks-Wallace (PTW) model.

**Isotropic Hardening Flow Stress Model** The Isotropic Hardening model is a simple linear relationship for the flow stress

$$\sigma_y(\epsilon_p) = \sigma_y + K(\epsilon_p) \tag{9.86}$$

where $\epsilon_p$ is the equivalent plastic strain, $\sigma_y$ and $k$ are material constants.

The inputs for this model are

```
<flow_model type="isotropic_hardening">
  <sigma_Y>792.0e6</sigma_y>
  <K>510.0e6</K>
</flow_model>
```

**JC Flow Stress Model** The Johnson-Cook (JC) model ([31]) is purely empirical and gives the following relation for the flow stress ($\sigma_y$)

$$\sigma_y(\epsilon_p, \dot{\epsilon}_p, T) = [A + B(\epsilon_p)^n] \left[1 + C \ln(\dot{\epsilon}_p^*)\right] [1 - (T^*)^m] \tag{9.87}$$

where $\epsilon_p$ is the equivalent plastic strain, $\dot{\epsilon}_p$ is the plastic strain rate, A, B, C, n, m are material constants,

$$\dot{\epsilon}_p^* = \frac{\dot{\epsilon}_p}{\dot{\epsilon}_{p0}}; \quad T^* = \frac{(T - T_0)}{(T_m - T_0)}\,, \tag{9.88}$$

$\dot{\epsilon}_{p0}$ is a user defined plastic strain rate, $T_0$ is a reference temperature, and $T_m$ is the melt temperature. For conditions where $T^* < 0$, we assume that $m = 1$.

The inputs for this model are

```
<flow_model type="johnson_cook">
  <A>792.0e6</A>
  <B>510.0e6</B>
  <C>0.014</C>
  <n>0.26</n>
  <m>1.03</m>
  <T_r>298.0</T_r>
  <T_m>1793.0</T_m>
  <epdot_0>1.0</epdot_0>
</flow_model>
```

**SCG Flow Stress Model** The Steinberg-Cochran-Guinan-Lund (SCG) model is a semi-empirical model that was developed by [53] for high strain rate situations and extended to low strain rates and bcc materials by [54]. The flow stress in this model is given by

$$\sigma_y(\epsilon_p, \dot{\epsilon}_p, T) = [\sigma_a f(\epsilon_p) + \sigma_t(\dot{\epsilon}_p, T)] \frac{\mu(p, T)}{\mu_0} \tag{9.89}$$

where $\sigma_a$ is the athermal component of the flow stress, $f(\epsilon_p)$ is a function that represents strain hardening, $\sigma_t$ is the thermally activated component of the flow stress, $\mu(p, T)$ is the shear modulus, and $\mu_0$ is the shear modulus at standard temperature and pressure. The strain hardening function has the form

$$f(\epsilon_p) = [1 + \beta(\epsilon_p + \varepsilon_{pi})]^n; \quad \sigma_a f(\epsilon_p) \leq \sigma_{\max} \tag{9.90}$$

where $\beta, n$ are work hardening parameters, and $\varepsilon_{pi}$ is the initial equivalent plastic strain. The thermal component $\sigma_t$ is computed using a bisection algorithm from the following equation (based on the work of [30])

$$\dot{\epsilon}_p = \left[\frac{1}{C_1} \exp\left[\frac{2U_k}{k_b\, T}\left(1 - \frac{\sigma_t}{\sigma_p}\right)^2\right] + \frac{C_2}{\sigma_t}\right]^{-1}; \quad \sigma_t \leq \sigma_p \tag{9.91}$$

where $2U_k$ is the energy to form a kink-pair in a dislocation segment of length $L_d$, $k_b$ is the Boltzmann constant, $\sigma_p$ is the Peierls stress. The constants $C_1, C_2$ are given by the relations

$$C_1 := \frac{\rho_d L_d a b^2 \nu}{2w^2}; \quad C_2 := \frac{D}{\rho_d b^2} \tag{9.92}$$

where $\rho_d$ is the dislocation density, $L_d$ is the length of a dislocation segment, $a$ is the distance between Peierls valleys, $b$ is the magnitude of the Burgers' vector, $\nu$ is the Debye frequency, $w$ is the width of a kink loop, and $D$ is the drag coefficient.

The inputs for this model are of the form

```
<flow_model type="steinberg_cochran_guinan">
  <mu_0> 81.8e9 </mu_0>
  <sigma_0> 1.15e9 </sigma_0>
  <Y_max> 0.25e9 </Y_max>
  <beta> 2.0 </beta>
  <n> 0.50 </n>
  <A> 20.6e-12 </A>
  <B> 0.16e-3 </B>
  <T_m0> 2310.0 </T_m0>
  <Gamma_0> 3.0 </Gamma_0>
  <a> 1.67 </a>
  <epsilon_p0> 0.0 </epsilon_p0>
</flow_model>
```

**ZA Flow Stress Model** The Zerilli-Armstrong (ZA) model ([65, 66, 64]) is based on simplified dislocation mechanics. The general form of the equation for the flow stress is

$$\sigma_y(\epsilon_p, \dot{\epsilon}_p, T) = \sigma_a + B\exp(-\beta(\dot{\epsilon}_p)T) + B_0\sqrt{\epsilon_p}\exp(-\alpha(\dot{\epsilon}_p)T) \tag{9.93}$$

where $\sigma_a$ is the athermal component of the flow stress given by

$$\sigma_a := \sigma_g + \frac{k_h}{\sqrt{l}} + K\epsilon_p^n, \tag{9.94}$$

$\sigma_g$ is the contribution due to solutes and initial dislocation density, $k_h$ is the microstructural stress intensity, $l$ is the average grain diameter, $K$ is zero for fcc materials, $B, B_0$ are material constants. The functional forms of the exponents $\alpha$ and $\beta$ are

$$\alpha = \alpha_0 - \alpha_1\ln(\dot{\epsilon}_p); \quad \beta = \beta_0 - \beta_1\ln(\dot{\epsilon}_p); \tag{9.95}$$

where $\alpha_0, \alpha_1, \beta_0, \beta_1$ are material parameters that depend on the type of material (fcc, bcc, hcp, alloys). The Zerilli-Armstrong model has been modified by [1] for better performance at high temperatures. However, we have not used the modified equations in our computations.

The inputs for this model are of the form

```
<flow_model type="zerilli_armstrong">
   <sigma_g>     50.0e6   </sigma_g>
   <k_H>         5.0e6    </k_H>
   <sqrt_l_inv>  5.0      </sqrt_l_inv>
   <B>           25.0e6   </B>
   <beta_0>      0.0      </beta_0>
   <beta_1>      0.0      </beta_1>
   <B_0>         0.0      </B_0>
   <alpha_0>     0.0      </alpha_0>
   <alpha_1>     0.0      </alpha_1>
   <K>           5.0e9    </K>
   <n>           1.0      </n>
 </flow_model>
```

**ZA for Polymers Flow Stress Model** The Zerilli-Armstrong flow stress model for polymers([67]) is a modification to the ZA flow stress model for metals motivated by considering thermally activated processess appropriate to polymers, in place of dislocations. The ZA flow stress function for

polymers has three terms. The first term accounts for a saturation of the flow stress to finite stress at higher temperatures (Although such stress component is specified as "athermal" it should follow the generally weaker temperature dependence of the elastic shear modulus, hence the subscript "g"). The second gives the yield stress as a function of temperature and plastic strain rate. The third gives an increment due to strain hardening, influenced by the pressure. The general form of the equation for the flow stress implemented in Uintah is

$$\sigma_y(\epsilon_p, \dot{\epsilon}_p, p, T) = \sigma_g + B\exp(-\beta(T - T_0)) + B_0\sqrt{\omega\epsilon_p}\exp(-\alpha(T - T_0)) \tag{9.96}$$

where it should be noted that the equation is slightly modified from the original to include a reference temperature $T_0$ and an athermal stress, $\sigma_g$, which is a constant. The other terms are specified via

$$B = B_{pa}(1 + B_{pb}\sqrt{p})^{B_{pn}}; \quad B_0 = B_{0pa}(1 + B_{0pb}\sqrt{p})^{B_{0pn}}; \quad \omega = \omega_a + \omega_b \ln(\dot{\epsilon}_p) + \omega_p\sqrt{p} \tag{9.97}$$

where $B_{pa}$, $B_{pb}$, $B_{pn}$, $B_{0pa}$, $B_{0pb}$, $B_{0pn}$, $\omega_a$, $\omega_b$, and $\omega_p$ are material parameters. The functional forms of the exponents $\alpha$ and $\beta$ are (as in the original)

$$\alpha = \alpha_0 - \alpha_1 \ln(\dot{\epsilon}_p); \quad \beta = \beta_0 - \beta_1 \ln(\dot{\epsilon}_p); \tag{9.98}$$

where $\alpha_0, \alpha_1, \beta_0, \beta_1$ are material parameters. Note that the pressure is taken to be min(p,0), eliminating pressure dependence in tension.

The inputs for this model are of the form

```
<flow_model type="zerilli_armstrong_polymer">
  <sigma_g>     50.0e6      </sigma_g>
  <B_pa>        0.0      </B_pa>
  <B_pb>        0.0      </B_pb>
  <B_pn>        0.0      </B_pn>
  <beta_0>      0.0      </beta_0>
  <beta_1>      0.0      </beta_1>
  <T_0>         0.0      </T_0>
  <B_0pa>       500.0e6      </B_0pa>
  <B_0pb>       0.0      </B_0pb>
  <B_0pn>       0.0      </B_0pn>
  <omega_a>     1.0      </omega_a>
  <omega_b>     0.0      </omega_b>
  <omega_p>     0.0      </omega_p>
  <alpha_0>     0.0      </alpha_0>
  <alpha_1>     0.0      </alpha_1>
</flow_model>
```

**MTS Flow Stress Model** The Mechanical Threshold Stress (MTS) model ([21, 22, 35]) gives the following form for the flow stress

$$\sigma_y(\epsilon_p, \dot{\epsilon}_p, T) = \sigma_a + (S_i\sigma_i + S_e\sigma_e)\frac{\mu(p, T)}{\mu_0} \tag{9.99}$$

where $\sigma_a$ is the athermal component of mechanical threshold stress, $\mu_0$ is the shear modulus at 0 K and ambient pressure, $\sigma_i$ is the component of the flow stress due to intrinsic barriers to thermally activated dislocation motion and dislocation-dislocation interactions, $\sigma_e$ is the component of the flow stress due to microstructural evolution with increasing deformation (strain hardening), $(S_i, S_e)$

are temperature and strain rate dependent scaling factors. The scaling factors take the Arrhenius form

$$S_i = \left[1 - \left(\frac{k_b\ T}{g_{0i} b^3 \mu(p,T)} \ln \frac{\dot{\epsilon}_{p0i}}{\dot{\epsilon}_p}\right)^{1/q_i}\right]^{1/p_i} \tag{9.100}$$

$$S_e = \left[1 - \left(\frac{k_b\ T}{g_{0e} b^3 \mu(p,T)} \ln \frac{\dot{\epsilon}_{p0e}}{\dot{\epsilon}_p}\right)^{1/q_e}\right]^{1/p_e} \tag{9.101}$$

where $k_b$ is the Boltzmann constant, $b$ is the magnitude of the Burgers' vector, $(g_{0i}, g_{0e})$ are normalized activation energies, $(\dot{\epsilon}_{p0i}, \dot{\epsilon}_{p0e})$ are constant reference strain rates, and $(q_i, p_i, q_e, p_e)$ are constants. The strain hardening component of the mechanical threshold stress $(\sigma_e)$ is given by a modified Voce law

$$\frac{d\sigma_e}{d\epsilon_p} = \theta(\sigma_e) \tag{9.102}$$

where

$$\theta(\sigma_e) = \theta_0[1 - F(\sigma_e)] + \theta_{IV} F(\sigma_e) \tag{9.103}$$

$$\theta_0 = a_0 + a_1 \ln \dot{\epsilon}_p + a_2 \sqrt{\dot{\epsilon}_p} - a_3 T \tag{9.104}$$

$$F(\sigma_e) = \frac{\tanh\left(\alpha \dfrac{\sigma_e}{\sigma_{es}}\right)}{\tanh(\alpha)} \tag{9.105}$$

$$\ln\left(\frac{\sigma_{es}}{\sigma_{0es}}\right) = \left(\frac{kT}{g_{0es} b^3 \mu(p,T)}\right) \ln\left(\frac{\dot{\epsilon}_p}{\dot{\epsilon}_{p0es}}\right) \tag{9.106}$$

and $\theta_0$ is the hardening due to dislocation accumulation, $\theta_{IV}$ is the contribution due to stage-IV hardening, $(a_0, a_1, a_2, a_3, \alpha)$ are constants, $\sigma_{es}$ is the stress at zero strain hardening rate, $\sigma_{0es}$ is the saturation threshold stress for deformation at 0 K, $g_{0es}$ is a constant, and $\dot{\epsilon}_{p0es}$ is the maximum strain rate. Note that the maximum strain rate is usually limited to about $10^7$/s.

The inputs for this model are of the form

```
<flow_model type="mts_model">
  <sigma_a>363.7e6</sigma_a>
  <mu_0>28.0e9</mu_0>
  <D>4.50e9</D>
  <T_0>294</T_0>
  <koverbcubed>0.823e6</koverbcubed>
  <g_0i>0.0</g_0i>
  <g_0e>0.71</g_0e>
  <edot_0i>0.0</edot_0i>
  <edot_0e>2.79e9</edot_0e>
  <p_i>0.0</p_i>
  <q_i>0.0</q_i>
  <p_e>1.0</p_e>
  <q_e>2.0</q_e>
  <sigma_i>0.0</sigma_i>
  <a_0>211.8e6</a_0>
  <a_1>0.0</a_1>
  <a_2>0.0</a_2>
```

```
  <a_3>0.0</a_3>
  <theta_IV>0.0</theta_IV>
  <alpha>2</alpha>
  <edot_es0>3.42e8</edot_es0>
  <g_0es>0.15</g_0es>
  <sigma_es0>1679.3e6</sigma_es0>
</flow_model>
```

**PTW Flow Stress Model** The Preston-Tonks-Wallace (PTW) model ([45]) attempts to provide a model for the flow stress for extreme strain rates (up to $10^{11}$/s) and temperatures up to melt. The flow stress is given by

$$\sigma_y(\epsilon_p, \dot{\epsilon}_p, T) = \begin{cases} 2\left[\tau_s + \alpha \ln\left[1 - \varphi \exp\left(-\beta - \dfrac{\theta \epsilon_p}{\alpha\varphi}\right)\right]\right]\mu(p,T) & \text{thermal regime} \\ 2\tau_s \mu(p,T) & \text{shock regime} \end{cases} \tag{9.107}$$

with

$$\alpha := \frac{s_0 - \tau_y}{d}; \quad \beta := \frac{\tau_s - \tau_y}{\alpha}; \quad \varphi := \exp(\beta) - 1 \tag{9.108}$$

where $\tau_s$ is a normalized work-hardening saturation stress, $s_0$ is the value of $\tau_s$ at 0K, $\tau_y$ is a normalized yield stress, $\theta$ is the hardening constant in the Voce hardening law, and $d$ is a dimensionless material parameter that modifies the Voce hardening law. The saturation stress and the yield stress are given by

$$\tau_s = \max\left\{s_0 - (s_0 - s_\infty)\text{erf}\left[\kappa \widehat{T} \ln\left(\frac{\gamma\dot{\xi}}{\dot{\epsilon}_p}\right)\right], s_0\left(\frac{\dot{\epsilon}_p}{\gamma\dot{\xi}}\right)^{s_1}\right\} \tag{9.109}$$

$$\tau_y = \max\left\{y_0 - (y_0 - y_\infty)\text{erf}\left[\kappa \widehat{T} \ln\left(\frac{\gamma\dot{\xi}}{\dot{\epsilon}_p}\right)\right], \min\left\{y_1\left(\frac{\dot{\epsilon}_p}{\gamma\dot{\xi}}\right)^{y_2}, s_0\left(\frac{\dot{\epsilon}_p}{\gamma\dot{\xi}}\right)^{s_1}\right\}\right\} \tag{9.110}$$

where $s_\infty$ is the value of $\tau_s$ close to the melt temperature, $(y_0, y_\infty)$ are the values of $\tau_y$ at 0K and close to melt, respectively, $(\kappa, \gamma)$ are material constants, $\widehat{T} = T/T_m$, $(s_1, y_1, y_2)$ are material parameters for the high strain rate regime, and

$$\dot{\xi} = \frac{1}{2}\left(\frac{4\pi\rho}{3M}\right)^{1/3}\left(\frac{\mu(p,T)}{\rho}\right)^{1/2} \tag{9.111}$$

where $\rho$ is the density, and $M$ is the atomic mass.

The inputs for this model are of the form

```
<flow_model type="preston_tonks_wallace">
  <theta> 0.025 </theta>
  <p> 2.0 </p>
  <s0> 0.0085 </s0>
  <sinf> 0.00055 </sinf>
  <kappa> 0.11 </kappa>
  <gamma> 0.00001 </gamma>
  <y0> 0.0001 </y0>
  <yinf> 0.0001 </yinf>
  <y1> 0.094 </y1>
  <y2> 0.575 </y2>
```

```
    <beta> 0.25 </beta>
    <M> 63.54 </M>
    <G0> 518e8 </G0>
    <alpha> 0.20 </alpha>
    <alphap> 0.20 </alphap>
  </flow_model>
```

**Return Algorithms** Two return algorithms are presently available. Both assume the direction of plastic flow is proportional to the current deviatoric stress.

1. Radial Return (**default**).

2. Modified Nemat-Nasser/Maudlin Return Algorithm.

**Radial Return Algorithm** The plastic state is obtained using an iterative radial return procedure as described in [50], page 124, except that the Newton procedure has been generalized to allow flow stresses to be functions of both equivalent plastic strain and equivalent plastic strain rate.

The rotated spatial rate of deformation tensor ($\boldsymbol{d}$) is additively decomposed into an elastic part, $\boldsymbol{d}^e$, and a plastic part, $\boldsymbol{d}^p$,

$$\boldsymbol{d} = \boldsymbol{d}^e + \boldsymbol{d}^p \tag{9.112}$$

It is convenient to work with the deviatoric parts of $\boldsymbol{d}$, $\boldsymbol{d}^e$, and $\boldsymbol{d}^p$, denoted $\boldsymbol{\eta}$, $\boldsymbol{\eta}^e$, and $\boldsymbol{\eta}^p$ respectively. The same additive decomposition obtains

$$\boldsymbol{\eta} = \boldsymbol{\eta}^e + \boldsymbol{\eta}^p \tag{9.113}$$

Presently these models are limited to the case of plastic incompressibility ($\mathrm{tr}(\boldsymbol{d}^p) = 0$), so that $\boldsymbol{d}^p = \boldsymbol{\eta}^p$.

The radial return algorithm assumes a yield condition of the form

$$f(\boldsymbol{s}, \epsilon_p, \dot{\epsilon_p}) = \sqrt{3J_2} - \sigma_y(\epsilon_p, \dot{\epsilon_p}) \tag{9.114}$$

where $\sigma_y$ is the flow stress, $J_2 = \frac{1}{2}\boldsymbol{s} : \boldsymbol{s}$ is the second invariant of the deviatoric part of the Cauchy stress, $\boldsymbol{s}$, and the equivalent plastic strain is defined as

$$\epsilon_p = \int_0^t \sqrt{\frac{2}{3}\boldsymbol{d}^p : \boldsymbol{d}^p}dt = \int_0^t \sqrt{\frac{2}{3}\boldsymbol{\eta}^p : \boldsymbol{\eta}^p}dt \tag{9.115}$$

Assuming a state at the end of the previous time step, time $t^n$, satisfying $f(\boldsymbol{s}^n, \epsilon_p^n, \dot{\epsilon}_p^n) \leq 0$, a new state satisfying Eqn. 9.114 at time $t^{n+1} = t^n + \Delta t$, where $\Delta t$ is the time step size, is sought.

Attention is further restricted to plastic flow associated with the yield condition, Eqn. 9.114, i.e.

$$\boldsymbol{d}^p = \boldsymbol{\eta}^p \propto \frac{\partial f}{\partial \boldsymbol{\sigma}} = \dot{\lambda}\frac{\boldsymbol{s}}{\|\boldsymbol{s}\|} = \dot{\lambda}\boldsymbol{n} \tag{9.116}$$

where $\boldsymbol{\sigma}$ is the Cauchy stress , $\boldsymbol{n} = \boldsymbol{s}/\|\boldsymbol{s}\|$ and $\dot{\lambda} > 0$ is a proportionality constant to be determined. Attention is also restricted to isotropic materials, for which the deviatoric response may be separated from the volumetric response. The linear hypoelastic/plastic constitutive equation for deviatoric response is

$$\dot{\boldsymbol{s}} = 2\mu(\boldsymbol{\eta} - \boldsymbol{\eta}^p) \tag{9.117}$$

where $\mu$ is the shear modulus. The shear modulus is required to be constant over the time step. This permits evolution of the shear modulus based on the state at the beginning of the time step using the various shear modulus models described later, which are pressure and temperature dependent. It also allows for a visoelastic deviatoric stress response provided an instantaneous shear modulus may be defined, as described later in this section for linear hypoviscoelasticity.

A trial stress is calculated assuming no plastic deformation, i.e.

$$\boldsymbol{s}^{\text{trial}} = \boldsymbol{s}^n + 2\mu\boldsymbol{\eta}\Delta t \tag{9.118}$$

If $f(\boldsymbol{s}^{\text{trial}}, \epsilon_p^n, 0) \leq 0$, the deformation is purely elastic and the solution at time $t^{n+1}$ is $\boldsymbol{s}^{n+1} = \boldsymbol{s}^{\text{trial}}$, $\epsilon_p^{n+1} = \epsilon_p^n$. If $f(\boldsymbol{s}^{\text{trial}}, \epsilon_p^n, 0) > 0$, the deformation is at least partially plastic. In this case

$$\boldsymbol{s}^{n+1} = \boldsymbol{s}^n + \dot{\boldsymbol{s}}\Delta t \tag{9.119}$$

where $\dot{\boldsymbol{s}}$ is given by Eqn. 9.117. Eqn. 9.119 may be rewritten in terms of the trial stress

$$\boldsymbol{s}^{n+1} = \boldsymbol{s}^{\text{trial}} - 2\mu\boldsymbol{\eta}^p\Delta t = \boldsymbol{s}^{\text{trial}} - 2\mu\dot{\lambda}\Delta t\frac{\boldsymbol{s}^{n+1}}{\|\boldsymbol{s}^{n+1}\|} \tag{9.120}$$

using Eqn.s 9.118, 9.117 and 9.116. This equation may be rearranged to give

$$\boldsymbol{s}^{\text{trial}} = \boldsymbol{s}^{n+1}\left[1 + \frac{2\mu\dot{\lambda}\Delta t}{\|\boldsymbol{s}^{n+1}\|}\right] \tag{9.121}$$

which gives the key result that $\boldsymbol{s}^{\text{trial}} \propto \boldsymbol{s}^{n+1}$, i.e. the trial stress and the updated stress are in the same direction. Consequently the flow direction may be written

$$\boldsymbol{n} = \frac{\boldsymbol{s}^{n+1}}{\|\boldsymbol{s}^{n+1}\|} = \frac{\boldsymbol{s}^{\text{trial}}}{\|\boldsymbol{s}^{\text{trial}}\|} \tag{9.122}$$

and Eqn. 9.120 may be rewritten

$$\boldsymbol{s}^{n+1} = \boldsymbol{s}^{\text{trial}} - 2\mu\dot{\lambda}\Delta t\boldsymbol{n} \tag{9.123}$$

or, contracting both sides with $\boldsymbol{n}$, and using Eqn. 9.122,

$$\left\|\boldsymbol{s}^{n+1}\right\| = \left\|\boldsymbol{s}^{\text{trial}}\right\| - 2\mu\dot{\lambda}\Delta t \tag{9.124}$$

which is a scalar equation for the proportionality constant $\dot{\lambda}$. Using the yield condition $f(\boldsymbol{s}^{n+1}, \epsilon_p^{n+1}, \dot{\epsilon}_p^{n+1}) = 0$ (Eqn. 9.114), this equation may be written in terms of $\dot{\lambda}$

$$\sqrt{\frac{2}{3}}\sigma_y(\epsilon_p^{n+1}, \dot{\epsilon}_p^{n+1}) = \left\|\boldsymbol{s}^{\text{trial}}\right\| - 2\mu\dot{\lambda}\Delta t \tag{9.125}$$

where from Eqn.s 9.115 and 9.116, $\epsilon_p^{n+1} = \epsilon_p^n + \sqrt{\frac{2}{3}}\dot{\lambda}\Delta t$ and $\dot{\epsilon_p}^{n+1} = \sqrt{\frac{2}{3}}\dot{\lambda}$.

For the special case of linear isotropic hardening, $\sigma_y(\epsilon_p, \dot{\epsilon}_p) = \sigma_{y0} + k\epsilon_p$, where $\sigma_{y0}$ is the initial yield stress and $k$ is the hardening modulus, Eqn. 9.125 may be solved exactly. More generally the solution may be found using Newton iteration. Defining $\Delta\lambda = \dot{\lambda}\Delta t$, and letting $j$ denote the iteration, define

$$g(\Delta\lambda_j) = \left\|\boldsymbol{s}^{\text{trial}}\right\| - 2\mu\Delta\lambda_j - \sqrt{\frac{2}{3}}\sigma_y(\epsilon_{p,j}^{n+1}, \dot{\epsilon}_{p,j}^{n+1}) \tag{9.126}$$

and, using the chain rule, the derivative may be calculated

$$\frac{dg}{d\Delta\lambda}(\Delta\lambda_j) = -2\mu - \frac{2}{3}\left[\frac{\partial\sigma_y}{\partial\epsilon_p}(\epsilon_{p,j}^{n+1}, \dot{\epsilon}_{p,j}^{n+1}) + \frac{\partial\sigma_y}{\partial\dot{\epsilon}_{p,j}^{n+1}}(\epsilon_{p,j}^{n+1}, \dot{\epsilon}_{p,j}^{n+1})\frac{1}{\Delta t}\right] \tag{9.127}$$

Then until $|g(\Delta\lambda_{j+1})| < \text{TOL}$, calculate

$$\Delta\lambda_{j+1} = \Delta\lambda_j - \frac{g(\Delta\lambda_j)}{\frac{dg}{d\Delta\lambda}(\Delta\lambda_j)} \tag{9.128}$$

where $\Delta\lambda_0 = 0$, $\epsilon_{p,0}^{n+1} = \epsilon_p^n$, $\dot{\epsilon}_{p,0}^{n+1} = 0$, and, once $\Delta\lambda$ has been determined to a specified accuracy, the final values of plastic strain and strain rate are given by

$$\epsilon_p^{n+1} = \epsilon_p^n + \sqrt{\frac{2}{3}}\Delta\lambda_{j+1} \tag{9.129}$$

$$\dot{\epsilon_p}^{n+1} = \sqrt{\frac{2}{3}}\frac{\Delta\lambda_{j+1}}{\Delta t} \tag{9.130}$$

and the final value of $\boldsymbol{s}^{n+1}$ is calculated from Eqn. 9.123.

While several of the allowed flow stresses are of the form $\sigma_y(\epsilon_p, \dot{\epsilon_p})$ as given in Eqn. 9.114, others include temperature and/or pressure dependence. Similarly, several of the shear moduli models are functions of temperature and/or pressure. Using the radial return algorithm in these cases amounts to convergence to the yield surface neglecting temperature changes, and assuming a non-associated flow rule of the form in Eqn. 9.116 (which is non-associated because the pressure dependence of the flow stress has been neglected, i.e. Eqn. 9.116 no longer holds). This non associated flow rule results in zero plastic dilation (actually dilitation). The end result is convergence to the flow stress with temperature and pressure held constant, i.e. to $\sigma_y(\epsilon_p^{n+1}, \dot{\epsilon_p}^{n+1}, p^n, T^n)$ with $\mu(p^n, T^n)$ and no plastic dilitation. While holding pressure and temperature fixed over a time step is probably a good approximation for most explicit calculations, non-associated flow may not be.

Finally, it was found that this return algorithm worked equally well for linear hypoviscoelastic deviatoric response, i.e.

$$\dot{\boldsymbol{s}} = 2\mu(\boldsymbol{\eta} - \boldsymbol{\eta}^p) - \sum_{i=1}^{N}\frac{\boldsymbol{s}_i}{\tau_i} \tag{9.131}$$

rather than Eqn. 9.117. Eqn. 9.131 is the constitutive equation for $N$ linear Maxwell elements in parallel, each with shear modulus $\mu_i$, time constant $\tau_i$, and deviatoric stress $\boldsymbol{s}_i$, and

$$\boldsymbol{s} = \sum_{i=1}^{N}\boldsymbol{s}_i \qquad \mu = \sum_{i=1}^{N}\mu_i \tag{9.132}$$

This model is detailed in the Deviatoric Stress Models section. Combined with a yield condition, the combination results in a model for viscoplastic material response.

The radial return algorith is the **default**, and also may be explicitely invoked with the tag

```
<plastic_convergence_algo>radialReturn</plastic_convergence_algo>
```

**Modified Nemat-Nasser/Maudlin Return Algorithm** This stress update algorithm is a slightly modified version of the approach taken by Nemat-Nasser et al. (1991,1992) [41, 42], Wang (1994) [62], Maudlin (1996) [38], and Zocher et al. (2000) [68]. It is presently only documented in the code itself. It is also known to give erroneous results under uniaxial stress conditions.

The modified Nemat-Nasser/Maudlin return algorith is invoked with the tag

```
<plastic_convergence_algo>biswajit</plastic_convergence_algo>
```

*This is an experts only algorithm!!!*

**Damage Models and Failure** Only the Johnson-Cook damage evolution rule has been added to the DamageModelFactory so far. The damage model framework is designed to be similar to the plasticity model framework. New models can be added using the approach described later in this section.

A particle is tagged as "failed" when its temperature is greater than the melting point of the material at the applied pressure. An additional condition for failure is when the porosity of a particle increases beyond a critical limit and the strain exceeds the fracture strain of the material. Another condition for failure is when a material bifurcation condition such as the Drucker stability postulate is satisfied. Upon failure, a particle is either removed from the computation by setting the stress to zero or is converted into a material with a different velocity field which interacts with the remaining particles via contact. Either approach leads to the simulation of a newly created surface. More details of the approach can be found in [2, 3, 4].

**Damage model** After the stress state has been determined on the basis of the yield condition and the associated flow rule, a scalar damage state in each material point can be calculated using the Johnson-Cook model [32]. The Johnson-Cook model has an explicit dependence on temperature, plastic strain, ans strain rate.

The damage evolution rule for the Johnson-Cook damage model can be written as

$$\dot{D} = \frac{\dot{\epsilon_p}}{\epsilon_p^f} \; ; \quad \epsilon_p^f = \left[ D_1 + D_2 \exp\left( \frac{D_3}{3}\sigma^* \right) \right] \left[ 1 + D_4 \ln(\dot{\epsilon_p}^*) \right] \left[ 1 + D_5 T^* \right] \; ; \quad \sigma^* = \frac{\mathrm{Tr}(\boldsymbol{\sigma})}{\sigma_{eq}} \; ; \tag{9.133}$$

where $D$ is the damage variable which has a value of 0 for virgin material and a value of 1 at fracture, $\epsilon_p^f$ is the fracture strain, $D_1, D_2, D_3, D_4, D_5$ are constants, $\boldsymbol{\sigma}$ is the Cauchy stress, and $T^*$ is the scaled temperature as in the Johnson-Cook plasticity model.

The input tags for the damage model are :

```
<damage_model type="johnson_cook">
  <D1>0.05</D1>
  <D2>3.44</D2>
  <D3>-2.12</D3>
  <D4>0.002</D4>
  <D5>0.61</D5>
</damage_model>
```

An initial damage distribution can be created using the following tags

```
<evolve_damage>                 true  </evolve_damage>
<initial_mean_scalar_damage>    0.005  </initial_mean_scalar_damage>
<initial_std_scalar_damage>     0.001 </initial_std_scalar_damage>
<critical_scalar_damage>        1.0   </critical_scalar_damage>
<initial_scalar_damage_distrib> gauss </initial_scalar_damage_distrib>
```

**Erosion algorithm** Under normal conditions, the heat generated at a material point is conducted away at the end of a time step using the heat equation. If special adiabatic conditions apply (such as in impact problems), the heat is accumulated at a material point and is not conducted to the surrounding particles. This localized heating can be used to determine whether a material point has melted.

The determination of whether a particle has failed can be made on the basis of either or all of the following conditions:

- The particle temperature exceeds the melting temperature.
- The TEPLA-F fracture condition [33] is satisfied. This condition can be written as

$$(f/f_c)^2 + (\epsilon_p/\epsilon_p^f)^2 = 1 \tag{9.134}$$

  where $f$ is the current porosity, $f_c$ is the maximum allowable porosity, $\epsilon_p$ is the current plastic strain, and $\epsilon_p^f$ is the plastic strain at fracture.
- An alternative to ad-hoc damage criteria is to use the concept of bifurcation to determine whether a particle has failed or not. Two stability criteria have been explored in this paper - the Drucker stability postulate [20] and the loss of hyperbolicity criterion (using the determinant of the acoustic tensor) [49, 43].

The simplest criterion that can be used is the Drucker stability postulate [20] which states that time rate of change of the rate of work done by a material cannot be negative. Therefore, the material is assumed to become unstable (and a particle fails) when

$$\dot{\boldsymbol{\sigma}} : \boldsymbol{D}^p \leq 0 \tag{9.135}$$

Another stability criterion that is less restrictive is the acoustic tensor criterion which states that the material loses stability if the determinant of the acoustic tensor changes sign [49, 43]. Determination of the acoustic tensor requires a search for a normal vector around the material point and is therefore computationally expensive. A simplification of this criterion is a check which assumes that the direction of instability lies in the plane of the maximum and minimum principal stress [10]. In this approach, we assume that the strain is localized in a band with normal $\boldsymbol{n}$, and the magnitude of the velocity difference across the band is $\boldsymbol{g}$. Then the bifurcation condition leads to the relation

$$R_{ij}g_j = 0 \ ; \quad R_{ij} = M_{ikjl}n_k n_l + M_{ilkj}n_k n_l - \sigma_{ik}n_j n_k \tag{9.136}$$

where $M_{ijkl}$ are the components of the co-rotational tangent modulus tensor and $\sigma_{ij}$ are the components of the co-rotational stress tensor. If $\det(R_{ij}) \leq 0$, then $g_j$ can be arbitrary and there is a possibility of strain localization. If this condition for loss of hyperbolicity is met, then a particle deforms in an unstable manner and failure can be assumed to have occurred at that particle. We use a combination of these criteria to simulate failure.

Since the material in the container may unload locally after fracture, the hypoelastic-plastic stress update may not work accurately under certain circumstances. An improvement would be to use a hyperelastic-plastic stress update algorithm. Also, the plasticity models are temperature dependent. Hence there is the issue of severe mesh dependence due to change of the governing equations from hyperbolic to elliptic in the softening regime [29, 9, 59]. Viscoplastic stress update models or nonlocal/gradient plasticity models [46, 27] can be used to eliminate some of these effects and are currently under investigation.

The tags used to control the erosion algorithm are in two places. In the `<MPM> </MPM>` section the following flags can be set

```
<erosion algorithm = "ZeroStress"/>
<create_new_particles>           false      </create_new_particles>
<manual_new_material>            false      </manual_new_material>
```

If the erosion algorithm is "none" then no particle failure is done.

In the <constitutive_model type="elastic_plastic"> section, the following flags can be set

```
<evolve_porosity>               true  </evolve_porosity>
<evolve_damage>                 true  </evolve_damage>
<do_melting>                    true  </do_melting>
<useModifiedEOS>                true  </useModifiedEOS>
<check_TEPLA_failure_criterion> true  </check_TEPLA_failure_criterion>
<check_max_stress_failure>      false </check_max_stress_failure>
<critical_stress>              12.0e9 </critical_stress>
```

**Implementation** The elastic response is assumed to be isotropic. The material constants that are taken as input for the elastic response are the bulk and shear modulus. The flow rule is determined from the input and the appropriate plasticity model is created using the `PlasticityModelFactory` class. The damage evolution rule is determined from the input and a damage model is created using the `DamageModelFactory` class. The equation of state that is used to determine the pressure is also determined from the input. The equation of state model is created using the `MPMEquationOfStateFactory` class.

In addition, a damage evolution variable ($D$) is stored at each time step (this need not be the case and will be transfered to the damage models in the future). The left stretch and rotation are updated incrementally at each time step (instead of performing a polar decomposition) and the rotation tensor is used to rotate the Cauchy stress and rate of deformation to the material coordinates at each time step (instead of using a objective stress rate formulation).

Any evolution variables for the plasticity model, damage model or the equation of state are specified in the class that encapsulates the particular model.

The flow stress is calculated from the plasticity model using a function call of the form

```
double flowStress = d_plasticity->computeFlowStress(tensorEta, tensorS,
                                                    pTemperature[idx],
                                                    delT, d_tol, matl, idx);
```

A number of plasticity models can be evaluated using the inputs in the `computeFlowStress` call. The variable `d_plasticity` is polymorphic and can represent any of the plasticity models that can be created by the plasticity model factory. The plastic evolution variables are updated using a polymorphic function along the lines of `computeFlowStress`.

The equation of state is used to calculate the hydrostatic stress using a function call of the form

```
Matrix3 tensorHy = d_eos->computePressure(matl, bulk, shear,
                                          tensorF_new, tensorD,
                                          tensorP, pTemperature[idx],
                                          rho_cur, delT);
```

Similarly, the damage model is called using a function of the type

```
double damage = d_damage->computeScalarDamage(tensorEta, tensorS,
                                              pTemperature[idx],
                                              delT, matl, d_tol,
                                              pDamage[idx]);
```

Therefore, the plasticity, damage and equation of state models are easily be inserted into any other type of stress update algorithm without any change being needed in them as can be seen in the hyperelastic-plastic stress update algorithm discussed below.

**Example input file for the elastic-plastic model** An example of the portion of an input file that specifies a copper body with a hypoelastic stress update, Johnson-Cook plasticity model, Johnson-Cook Damage Model and Mie-Gruneisen Equation of State is shown below.

```
<material>

  <include href="inputs/MPM/MaterialData/MaterialConstAnnCopper.xml"/>
  <constitutive_model type="elastic_plastic">
    <tolerance>5.0e-10</tolerance>
    <include href="inputs/MPM/MaterialData/IsotropicElasticAnnCopper.xml"/>
    <include href="inputs/MPM/MaterialData/JohnsonCookPlasticAnnCopper.xml"/>
    <include href="inputs/MPM/MaterialData/JohnsonCookDamageAnnCopper.xml"/>
    <include href="inputs/MPM/MaterialData/MieGruneisenEOSAnnCopper.xml"/>
  </constitutive_model>

  <geom_object>
    <cylinder label = "Cylinder">
      <bottom>[0.0,0.0,0.0]</bottom>
      <top>[0.0,2.54e-2,0.0]</top>
      <radius>0.762e-2</radius>
    </cylinder>
    <res>[3,3,3]</res>
    <velocity>[0.0,-208.0,0.0]</velocity>
    <temperature>294</temperature>
  </geom_object>

</material>
```

The general material constants for copper are in the file `MaterialConstAnnCopper.xml`. The contents are shown below

```
<?xml version='1.0' encoding='ISO-8859-1' ?>
<Uintah_Include>
  <density>8930.0</density>
  <toughness>10.e6</toughness>
  <thermal_conductivity>1.0</thermal_conductivity>
  <specific_heat>383</specific_heat>
  <room_temp>294.0</room_temp>
  <melt_temp>1356.0</melt_temp>
</Uintah_Include>
```

The elastic properties are in the file `IsotropicElasticAnnCopper.xml`. The contents of this file are shown below.

```
<?xml version='1.0' encoding='ISO-8859-1' ?>
<Uintah_Include>
  <shear_modulus>45.45e9</shear_modulus>
  <bulk_modulus>136.35e9</bulk_modulus>
</Uintah_Include>
```

The constants for the Johnson-Cook plasticity model are in the file `JohnsonCookPlasticAnnCopper.xml`. The contents of this file are shown below.

```
<?xml version='1.0' encoding='ISO-8859-1' ?>
<Uintah_Include>
  <flow_model type="johnson_cook">
    <A>89.6e6</A>
```

```
    <B>292.0e6</B>
    <C>0.025</C>
    <n>0.31</n>
    <m>1.09</m>
  </flow_model>
</Uintah_Include>
```

The constants for the Johnson-Cook damage model are in the file `JohnsonCookDamageAnnCopper.xml`. The contents of this file are shown below.

```
<?xml version='1.0' encoding='ISO-8859-1' ?>
<Uintah_Include>
  <damage_model type="johnson_cook">
    <D1>0.54</D1>
    <D2>4.89</D2>
    <D3>-3.03</D3>
    <D4>0.014</D4>
    <D5>1.12</D5>
  </damage_model>
</Uintah_Include>
```

The constants for the Mie-Gruneisen model (as implemented in the Uintah Computational Framework) are in the file `MieGruneisenEOSAnnCopper.xml`. The contents of this file are shown below.

```
<?xml version='1.0' encoding='ISO-8859-1' ?>
<Uintah_Include>
  <equation_of_state type="mie_gruneisen">
    <C_0>3940</C_0>
    <Gamma_0>2.02</Gamma_0>
    <S_alpha>1.489</S_alpha>
  </equation_of_state>
</Uintah_Include>
```

As can be seen from the input file, any other plasticity model, damage model and equation of state can be used to replace the Johnson-Cook and Mie-Gruneisen models without any extra effort (provided the models have been implemented and the data exist).

The material data can easily be taken from a material database or specified for a new material in an input file kept at a centralized location. At this stage material data for a range of materials is kept in the directory `.../Uintah/StandAlone/inputs/MPM/MaterialData`.

**Adding new models** In the parallel implementation of the stress update algorithm, sockets have been added to allow for the incorporation of a variety of plasticity, damage, yield, and bifurcation models without requiring any change in the stress update code. The algorithm is shown in Algorithm 9.1. The equation of state, plasticity model, yield condition, damage model, and the stability criterion are all polymorphic objects created using a factory idiom in C++ ([17]).

Addition of a new model requires the following steps (the example below is only for the flow stress model but the same idea applies to other models) :

1. Creation of a new class that encapsulates the plasticity model. The template for this class can be copied from the existing plasticity models. The data that is unique to the new model are specified in the form of

   - A structure containing the constants for the plasticity model.

Table 9.1: Stress Update Algorithm

**Persistent**:Initial moduli, temperature, porosity,
scalar damage, equation of state, plasticity model,
yield condition, stability criterion, damage model
**Temporary**:Particle state at time $t$
**Output:** Particle state at time $t + \Delta t$

**For** *all the patches in the domain*
Read the particle data and initialize updated data storage
**For** *all the particles in the patch*
Compute the velocity gradient and the rate of deformation tensor
Compute the deformation gradient and the rotation tensor
Rotate the Cauchy stress and the rate of deformation tensor
to the material configuration
Compute the current shear modulus and melting temperature
Compute the pressure using the equation of state,
update the hydrostatic stress, and
compute the trial deviatoric stress
Compute the flow stress using the plasticity model
Evaluate the yield function
**If** *particle is elastic*
Update the elastic deviatoric stress from the trial stress
Rotate the stress back to laboratory coordinates
Update the particle state
**Else**
Compute the elastic-plastic deviatoric stress
Compute updated porosity, scalar damage, and
temperature increase due to plastic work
Compute elastic-plastic tangent modulus and evaluate stability condition
Rotate the stress back to laboratory coordinates
Update the particle state
**End If**
**If** *Temperature* > *Melt Temperature* or *Porosity* > *Critical Porosity* or *Unstable*
Tag particle as failed
**End If**
Convert failed particles into a material with a different velocity field
**End For**
**End For**

- Particle variables that specify the variables that evolve in the plasticity model.

2. The implementation of the plasticity model involves the following steps.
   - Reading the input file for the model constants in the constructor.
   - Adding the variables that evolve in the plasticity model appropriately to the task graph.
   - Adding the appropriate flow stress calculation method.
3. The `PlasticityModelFactory` is then modified so that it recognizes the added plasticity model.

### 9.5.7 Contact

When multiple materials are specified in the input file, each material interacts with its own field variables. In other words, each material has its own mass, velocity, acceleration, etc. Without any mechanism for their interaction, each material would behave as if it were the only one in the domain. Contact models provide the mechanism by which to specify rules for inter material interactions. There are a number of contact models from which to choose, the use of each is described next. See the input file segment in Section 9.5.4 for an example of their proper placement in the input file, namely, after all of the MPM materials have been described.

The simplest contact model is the `null` model, which indicates that no inter material interactions are to take place. This is typically only used in single material simulations. Its usage looks like:

```
<contact>
  <type>null</type>
</contact>
```

The next simplest model is the `single_velocity` model. The basic MPM formulation provides "free" no-slip, no-interpenetration contact, assuming that all particle data communicates with a single field on the grid. For a single material simulation with multiple objects, that is the case. If one wishes to achieve that behavior in Uintah-MPM when multiple materials are present, the `single_velocity` contact model should be used. It is specified as:

```
<contact>
  <type>single_velocity</type>
  <materials>[0,1]</materials>
</contact>
```

Note that for this, and all of the contact models, the `<materials>` tag is optional. If it is omitted, the assumption is that all materials will interact via the same contact model. (This will be further discussed below.)

The ultimate in contact models is the `friction` contact model. For a full description, the reader is directed to the paper by Bardenhagen et al.[7]. Briefly, the model both overcomes some deficiences in the single velocity field contact (either the "free" contact or the model described above, which behave identically), and it enables some additional features. With single velocity field contact, initially adjacent objects are treated as if they are effectively stuck together. The friction contact model overcomes this by detecting if materials are approaching or departing at a given node. If they are approaching, contact is "enforced" and if they are departing, another check is

made to determine if the objects are in compression or tension. If they are in compression, then they are still rebounding from each other, and so contact is enforced. If tension is detected, they are allowed to move apart independently. Frictional sliding is allowed, based on the value specified for `<mu>` and the normal force between the objects. An example of the use of this model is given here:

```
<contact>
   <type>friction</type>
   <materials>[0,1,2]</materials>
   <mu> 0.5 </mu>
</contact>
```

A slightly simplified version of the friction model is the `<approach>` model. It is the same as the frictional model above, except that it doesn't make the additional check on the traction between two bodies at each node. At times, it is necessary to neglect this, but some loss of energy will result. Specification is of the model is also nearly identical:

```
<contact>
   <type>approach</type>
   <materials>[0,1,2]</materials>
   <mu> 0.5 </mu>
</contact>
```

Finally, the contact infrastructure is also used to provide a moving displacement boundary condition. Imagine a billet being smashed by a rigid platen, for example. Usage of this model, known as `<specified>` contact, looks like:

```
<contact>
  <type>specified</type>
  <filename>TXC.txt</filename>
  <materials>[0,1,2]</materials>
  <master_material>0</master_material>
  <direction>[1,1,1]</direction>
  <stop_time>1.0 </stop_time>
  <velocity_after_stop>[0, 0, 0]</velocity_after_stop>
</contact>
```

For reasons of backwards compatibility, the `<type>specified</type>` is interchangable with `<type>rigid</t` By default, when either model is chosen, material 0 is the "rigid" material, although this can be over ridden by the use of the `<master_material>` field. If no `<filename>` field is specified, then the particles of the rigid material proceed with the velocity that they were given as their initial condition, either until the reach a computational boundary, or until the simulation time has reached `<stop_time>,` after which, their velocity becomes that given in the `<velocity_after_stop>` field. The `<direction>` field indicates in which cartesian directions contact should be specified. Values of 1 indicate that contact should be specified, 0 indicates that the subject materials should be allowed to slide in that direction. If a `<filename>` field *is* specified, then the user can create a text file which contains four entries per line. These are:

```
time1 velocity_x1 velocity_y1 velocity_z1
time2 velocity_x2 velocity_y2 velocity_z2
     .
     .
     .
```

The velocity of the rigid material particles will be set to these values, based on linear interpolation between times, until `<stop_time>` is reached. Note, one should not try to apply traction boundary conditions (via the `<PhysicalBC>` tag), to the rigid material used in this type of contact, as this constitutes trying to mix displacement and traction boundary conditions.

Finally, it is possible to specify more than one contact model. Suppose one has a simulation with three materials, one rigid, and the other two deformable. The user may want to have the rigid material interact in a rigid manner with the other two materials, while the two deformable materials interact with each other in a single velocity field manner. Specification for this, assuming the rigid material is 0 would look like:

```
<contact>
    <type>single_velocity</type>
    <materials>[1,2]</materials>
</contact>

<contact>
    <type>specified</type>
    <filename>prof.txt</filename>
    <stop_time>1.0</stop_time>
    <direction>[0, 0, 1]</direction>
</contact>
```

An example of this usage can be found in `inputs/MPM/twoblock-single-rigid.ups`.

### 9.5.8 BoundaryConditions

Boundary conditions must be specified on each face of the computational domain ($x^-, x^+, y^-, y^+, z^-, z^+$) for each material. An example of their specification is as follows, where the entire `<Grid>` field is included for context:

```
<Grid>
   <BoundaryConditions>
     <Face side = "x-">
         <BCType id = "all" var = "Dirichlet" label = "Velocity">
               <value> [0.0,0.0,0.0] </value>
         </BCType>
     </Face>
     <Face side = "x+">
        <BCType id = "all" var = "Neumann" label = "Velocity">
             <value> [0.0,0.0,0.0] </value>
        </BCType>
     </Face>
     <Face side = "y-">
       <BCType id = "all" var = "Dirichlet" label = "Velocity">
              <value> [0.0,0.0,0.0] </value>
       </BCType>
     </Face>
     <Face side = "y+">
       <BCType id = "all" var = "Neumann" label = "Velocity">
              <value> [0.0,0.0,0.0] </value>
       </BCType>
     </Face>
     <Face side = "z-">
       <BCType id = "all" var = "symmetry" label = "Symmetric"> </BCType>
```

```
      </Face>
      <Face side = "z+">
        <BCType id = "all" var = "symmetry" label = "Symmetric"> </BCType>
      </Face>
    </BoundaryConditions>
    <Level>
```

... See Section 2.10 ...

```
    </Level>
 </Grid>
```

The three main types of numerical boundary conditions (BCs) that can be applied are "Neumann", "Dirichlet", and "Symmetric", and the use of each is illustrated above. In the case of MPM simulations, Neumann BCs are used when one wishes to allow particles to advect freely out of the computational domain. Dirichlet BCs are used to specify a velocity, zero or otherwise (indicated by the `<value>` tag), on one of the computational boundaries. Symmetric BCs are used to indicate a plane of symmetry. This has a variety of uses. The most obvious is simply when a simulation of interest has symmetry that one can take advantage of to reduce the cost of a calculation. Similarly, since Uintah is a three-dimensional code, if one wishes to achieve plane-strain conditions, this can be done by carrying out a simulation that is one cell thick with Symmetric BCs applied to each face of the plane, as in the example above. Finally, Symmetric BCs also provide a free slip boundary.

There is also the field `id = "all"` . In principal, one could set different boundary condition types for different materials. In practice, this is rarely used, so the usage illustrated here should be used.

### 9.5.9 Physical Boundary Conditions

It is often more convenient to apply a specified load at the MPM particles. The load may be a function of time. Such a load versus time curve is called a **load curve**. In Uintah, the load curve infrastructure is available for general use (and not only for particles). However, it has been implemented only for a special case of pressure loading. Namely, a surface is specified through the use of the `<geom_object>` description, and a pressure vs. time curve is described by specifying their values at discrete points in time, between which linear interpolation is used to find values at any time. At $t = 0$, those particles in the vicinity of the the surface are tagged with a load curve ID, and those particles are assigned external forces such that the desired pressure is achieved.

We invoke the load curve in the `<MPM>` section (See Section 9.5.3)of the input file using `<use_load_curves>` t
The default value is `<use_load_curves> false </use_load_curves>`.

In Uintah, a load curve infrastructure is implemented in the file
`.../MPM/PhysicalBC/LoadCurve.h`. This file is essentially a templated structure that has the following private data

```
// Load curve information
std::vector<double> d_time;
std::vector<T> d_load;
int d_id;
```

The variable `d_id` is the load curve ID, `d_time` is the time, and `d_load` is the load. Note that the load can have any form - scalar, vector, matrix, etc.

In our current implementation, the actual specification of the load curve information is in the <PhysicalBC> section of the input file. The implementation is limited in that it applies only to pressure boundary conditions for some special geometries (the implementation is in .../MPM/PhysicalBC/Pressure However, the load curve template can be used in other, more general, contexts.

A sample input file specification of a pressure load curve is shown below. In this case, a pressure is applied to the inside and outside of a cylinder. The pressure is ramped up from 0 to 1 GPa on the inside and from 0 to 0.1 MPa on the outside over a time of 10 microsecs.

```
<PhysicalBC>
  <MPM>
    <pressure>
      <geom_object>
        <cylinder label = "inner cylinder">
          <bottom>           [0.0,0.0,0.0]   </bottom>
          <top>              [0.0,0.0,.02]   </top>
          <radius>           0.5             </radius>
        </cylinder>
      </geom_object>
      <load_curve>
        <id>1</id>
        <time_point>
          <time> 0 </time>
          <load> 0 </load>
        </time_point>
        <time_point>
          <time> 1.0e-5 </time>
          <load> 1.0e9 </load>
        </time_point>
      </load_curve>
    </pressure>
    <pressure>
      <geom_object>
        <cylinder label = "outer cylinder">
          <bottom>           [0.0,0.0,0.0]   </bottom>
          <top>              [0.0,0.0,.02]   </top>
          <radius>           1.0             </radius>
        </cylinder>
      </geom_object>
      <load_curve>
        <id>2</id>
        <time_point>
          <time> 0 </time>
          <load> 0 </load>
        </time_point>
        <time_point>
          <time> 1.0e-5 </time>
          <load> 101325.0 </load>
        </time_point>
      </load_curve>
    </pressure>
  </MPM>
</PhysicalBC>
```

The complete input file can be found in `inputs/MPM/thickCylinderMPM.ups`. An additional example which is used to achieve triaxial loading can be found at `inputs/MPM/TXC.ups`. There, the material geometry is a block, and so the regions described are flat surfaces upon which the pressure

is applied.

### 9.5.10 On the Fly DataAnalysis

In the event that one wishes to monitor the data for a small region of a simulation at a rate that is more frequent than the what the DataArchiver can reasonably provide (for reasons of data storage and effect on run time), Uintah provides a `<DataAnalysis>` feature. As it applies to MPM, it allows one to specify a group of particles, by assigning those particles a particular value of the `<color>` parameter. In addition, a list of variables and a frequency of output is provided. Then, at run time, a sub-directory (`particleExtract/L-0` ) is created inside the uda which contains a series of files, named according to their particle IDs, one for each tagged particle. Each of these files contains the time and position for that particle, along with whatever other data is specified. **To use this feature, one must include the `<withColor> true </withColor>` tag in the `<MPM>` section of the input file.** (See Section 9.5.3.)

The following input file snippet is taken from `inputs/MPM/disks.ups`

```
<DataAnalysis>
   <Module name="particleExtract">

    <material>disks</material>
    <samplingFrequency> 1e10 </samplingFrequency>
    <timeStart>          0   </timeStart>
    <timeStop>          100  </timeStop>
    <colorThreshold>
      0
    </colorThreshold>

    <Variables>
      <analyze label="p.velocity"/>
      <analyze label="p.stress"/>
    </Variables>

  </Module>
</DataAnalysis>
```

For all particles that are assigned a color greater than the `<colorThreshold>,` the variables `p.velocity` and `p.stress` are saved every every 1/`<samplingFrequency>` time units, starting at `<timeStart>` until `<timeStop>`.

It is also possible to save grid based data with this module, see Section 8 for more information.

### 9.5.11 Prescribed Motion

The prescribed motion capability in Uintah allows the user to prescribe arbitrary material deformations and superimposed rotations. This capability is particularly useful in verifying that the constitutive model is behaving as expected and is frame indifferent. To prescribe material motion the following tag must be included in the `<MPM>` section of the input file:

```
<MPM>
    <UsePrescribedDeformation>true</UsePrescribedDeformation>
</MPM>
```

The desired motion must then be specified in a file named `time_defgrad_rotation` . The format of this file is as follows:

```
t0 F11 F12 F13 F21 F22 F23 F31 F32 F33 theta0 a0 a1 a2
t1 F11 F12 F13 F21 F22 F23 F31 F32 F33 theta1 a0 a1 a2
. . .
tn F11 F12 F13 F21 F22 F23 F31 F32 F33 thetan a0 a1 a2
```

where the first column is time, columns two through ten are the nine components of the prescribed deformation gradient, the eleventh column is the desired rotation angle, and the remaining three columns are the three components of the axis of prescribed rotation. The components of the deformation gradient are linearly interpolated for times between those specified in the table. The axis of rotation may be changed for each specified time. As a result, the angle of rotation about the specified axis linearly increases from zero to the specified value at the end of the specified interval. For example, the following table:

```
0 1 0 0 0 1 0 0 0 1 0 0 0 0
1 1 0 0 0 1 0 0 0 1 90 0 0 1
2 1 0 0 0 1 0 0 0 1 91 0 0 1
```

specifies a pure rotation (no stretch) about the 3-axis. At time=0 the material will have rotated 90 degrees about the 3-axis. At time=2 the material will have rotated an additional 91 degrees about the 3-axis for a total of 181 degrees of rotation. As a warning to the user, it is possible to specify the deformation gradient such that interpolating between to entries in the table results in a singular deformation gradient. For example:

```
0 1 0 0 0 1 0 0 0 1 0 0 0 0
1 1 0 0 0 1 0 0 0 1 0 0 0 1
2 -1 0 0 0 -1 0 0 0 1 0 0 0 1
```

would result in the simulation failing due to a negative jacobian error between time=1 and time=2 since the 11 and 22 components are linearly varying from 1 to -1 during that time, which will attempt to invert the computational cell. The deformation gradient at time=2 corresponds to a 180 degree rotation about the 3-axis, and can be accomplished using the rotation feature described above.

As a final example the table:

```
0 1 0 0 0 1 0 0 0 1 0 0 0 0
1 0.5 0 0 0 0.5 0 0 0 0.5 45 0 1 0
2 0.5 0 0 0.5 0.5 0 0 0 0.5 90 0 0 1
```

would result in 50% hydrostatic compression at time=1 with a 45 degree superimposed rotation about the 2-axis, followed by simple shear and a 90 degree rotation about the 3-axis between time=1 and time=2.

### 9.5.12 Cohesive Zones

A cohesive zone formulation is available in Uintah based on the description by Daphalapurkar, et al. [19]. As in their implementation, that in Uintah has several limitations. It is limited to a 2D implementation, and the cohesive zone segments are assumed to not rotate or deform.

In order to use cohesive zones, the following field must be added to the `<MPM>` section of the input file:

```
<MPM>
    <use_cohesive_zones>true</use_cohesive_zones>
</MPM>
```

The traction functions used in Uintah are those given in Eq. 15 of [19]. These require 4 input parameters. They are $\boldsymbol{\sigma}_{max}$, $\tau_{max}$, $\delta_n$ and $\delta_t$, the cohesive strengths in the normal and shear directions, and the displacement jumps in the normal and tangential directions corresponding to the maximum normal and shear strength values, respectively.

In an input file, the description of a cohesive zone looks like:

```
<cohesive_zone>
  <sig_max> 240. </sig_max>
  <tau_max> 240. </tau_max>
  <delta_n> 0.00004   </delta_n>
  <delta_t> 0.0000933 </delta_t>
  <cz_filename>HOM.txt</cz_filename>
</cohesive_zone>
```

Note that in addition to the four parameters listed above, a cohesive zone filename is also specified. The format of this file will be described below. Units on the strength and displacement correspond to the units for stress and length used in the remainder of the input file.

Cohesive zones describe a cohesion law between adjacent materials. As such, they take the place of a contact model. Thus, when using cohesive zones to describe the interaction of materials 1 and 2, the contact section of the input file would be:

```
<contact>
    <type>null</type>
    <materials>[1,2]</materials>
</contact>
```

Use of friction or approach contact to describe interaction between objects subsequent to decohesion should be possible and is being investigated.

The traction that is applied to the two materials governed by a cohesive zone model is based on the displacement between those two materials, both normal and tangential. The two adjacent materials are referred to in the implementation as the "Top" and "Bottom" materials. A normal and tangential vector describes the orientation of the cohesive zone surface. The convention for the normal vector is that it points in the direction from the bottom material to the top material. With this information in hand, we can describe the format of the `<cz_filename>` mentioned above.

```
px1 py1 pz1 length1 normx1 normy1 normz1 tangx1 tangy1 tangz1 botmat1 topmat1
px2 py2 pz2 length2 normx2 normy2 normz2 tangx2 tangy2 tangz2 botmat2 topmat2
. . .
pxN pyN pzN lengthN normxN normyN normzN tangxN tangyN tangzN botmatN topmatN
```

where the first three columns are the x, y and z coordinates of the position, the fourth column is the length, the fifth through seventh column is the normal direction (x, y, z) and the eighth through tenth column is the tangential direction (x, y, z). Finally, the eleventh and twelth columns are the bottom and top material indices, respectively.

An example of 3 cohesive zone segments follows:

```
2.5125 0.0 0.025 0.00125 0.0 1.0 0.0 1.0 0.0 0.0 1 2
2.5375 0.0 0.025 0.00125 0.0 1.0 0.0 1.0 0.0 0.0 1 2
2.5625 0.0 0.025 0.00125 0.0 1.0 0.0 1.0 0.0 0.0 1 2
```

As a 2D simulation in Uintah is actually a 3D simulation that is 1 cell thick, the "length" parameter described above is actually going to be an area. Namely, the length in the plane of the simulation multiplied by the domain thickness in the out of plane direction.

### 9.5.13 Particle Insertion

MPM has the ability to insert or transport blocks of particles into or around the computational domain. The functionality uses a time threshold for activation of the insertion. Currently, capabilities include translation some distances x, y and z and initiation of a new "initial" velocity vector. Particles, defined by color, specified as an integer, in the geometry object section of the input file, can have a limitless number of transformations applied to them. There are no limits to how many geometry objects can be specified, however, each transformation can only act on one color index. Thus movement of more than one block of particles can require multiple input lines.

This functionality is defined in a text file in the order:

```
<time> <color> <trans x> <trans y> <trans z> <new x vel> <new y vel> <new z vel>
```

During the first timestep in which the current physical time plus the calculated $\Delta$t for the current timestep exceeds the time specified for a color block, the particles of that color will be translated along the three coordinates and given a new velocity. Each line in the file can be used to define a unique transformation for one particle color group. For instance, if a file contained the line:

```
0.1 1 10 10 0 0 0 8
```

after '0.1 s' of physical time any particle of color '1' will be translated 10 units in the positive x and y direction, 0 units in the positive z direction and given a new velocity of 8 units/s in the positive z direction, with no velocity in the x or y direction.

Particle insertion is activated and directed with the following flags found in the MPM section of the input file:

```
<MPM>
    <withColor>         true   </withColor>
    <InsertParticles>   true   </InsertParticles>
    <InsertParticlesFile>  "path/to/file.txt" </InsertParticlesFile>
</MPM>
```

An example problem exists in "inputs/MPM/" named Extrude that demonstrates particle insertion. "extrude.ups" defines the problem setup, "extrude.xml" defines the geometry objects (also where the color is defined) and then pulled in to "extrude.ups", and "insert.dat" which defines the times, translations and new velocity of the particle blocks. Figure 9.5 shows an image of a simulation in progress that uses particle insertion. The image shows a stream of rubbery material flying into the domain and folding on itself. Another particularly useful idea to note from the image, is the secondary box above the normal domain, in which the particles to be inserted reside before they are inserted. Current application of particle insertion tends to follow this motif.

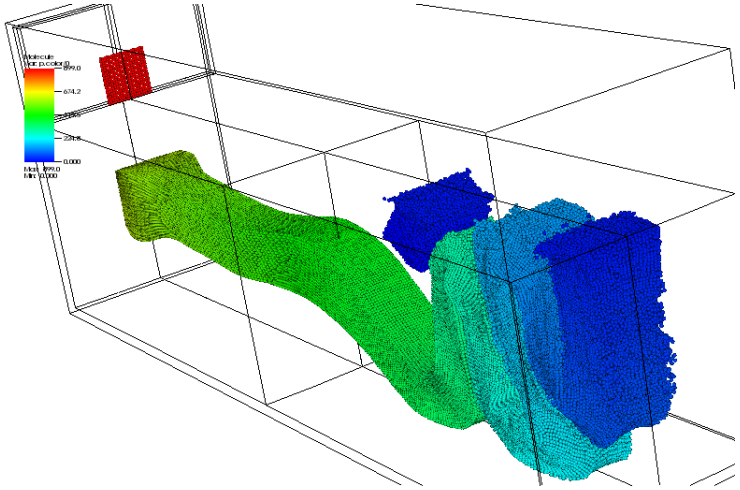


Figure 9.5: Particles being inserted from top box into bottom box.

## 9.6 Examples

The following examples are meant to be illustrative of a variety of capabilities of Uintah-MPM, but are by no means exhaustive. Input files for the examples given here can be found in:

`inputs/UintahRelease/MPM`

Additional (mostly undocumented) input files that exercise a greater range of code capabilities can be found in:

`inputs/MPM`

### Colliding Disks

#### Problem Description

This is an implementation of an example calculation from [55] in which two elastic disks collide and rebound. See Section 7.3 of that manuscript for a description of the problem.

#### Simulation Specifics

**Component used:** MPM

**Input file name:** disks_sulsky.ups

**Command used to run input file:**
sus inputs/UintahRelease/MPM/disks_sulsky.ups

**Simulation Domain:** 1.0 x 1.0 x 0.05 m

**Cell Spacing:**
.05 x .05 x .05 m (Level 0)

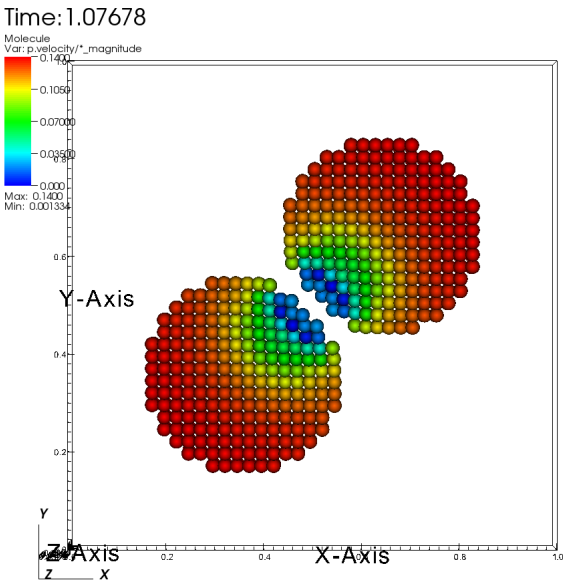



Figure 9.6: Colliding elastic disks. Particles colored according to velocity magnitude.

**Example Runtimes:**

4 seconds (1 processor, 3.16 GHz Xeon)

**Physical time simulated:** 3.0 seconds

**Associate VisIt session:** disks.session

## Results

Figure 9.6 shows a snapshot of the simulation, as the disks are beginning to collide.

Additional data is available within the uda in the form of "dat" files. In this case, both the kinetic and strain energies are avaiable and can be plotted to create a graph similar to that in Fig. 5a of [55]. e.g. using gnuplot:

```
cd disks.uda.000
gnuplot
gnuplot> plot "StrainEnergy.dat", "KineticEnergy.dat"
gnuplot> quit
```

# Taylor Impact Test

## Problem Description

This is a simulation of an Taylor impact experiment calculation from [26] in a copper cylinder at 718 K that is fired at a rigid anvil at 188 m/s. The copper cylinder has a length of 30 mm and a diameter of 6 mm. The cylinder rebounds from the anvil after 100 $\mu$s.

**Simulation Specifics**

**Component used:** MPM

**Input file name:** taylorImpact.ups

**Command used to run input file:**
sus inputs/UintahRelease/MPM/taylorImpact.ups

**Simulation Domain:** 8 mm x 33 mm x 8 mm

**Cell Spacing:**
1/3 mm x 1/3 mm x 1/3 mm (Level 0)

**Example Runtimes:**
1 hour (1 processor, Xeon 3.16 GHz)

**Physical time simulated:** 100 $\mu$seconds

**Associate VisIt session:** taylorImpact.session

**Results**

Figure 9.7 shows a snapshot from the end of the simulation. There, the cylinder is allowed to slide laterally across the plate due to the following optional specification in the `<contact>` section:

```
<direction>[0,1,0]</direction>
```

Figure 9.8 shows a snapshot from the end of a similar simulation. In this case, the cylinder is restricted from sliding laterally across the plate by altering the `<contact>` section as follows:

```
<direction>[1,1,1]</direction>
```

## Sphere Rolling Down an Inclined Plane

**Problem Description**

Here, a sphere of soft plastic, initially at rest, rolls under the influence of gravity down a plane of a harder plastic. Gravity is oriented such that the plane is effectively angled at 45 degrees to the horizontal. This simulation demonstrates the effectiveness of the contact algorithm, described in [5]. Frictional contact, using a friction coefficient of $\mu = 0.495$ causes the ball to start rolling as it impacts the plane, after being dropped from barely above it. The same simulation is also run using a friction coefficient of $\mu = 0.0$. The difference in the results is shown below.

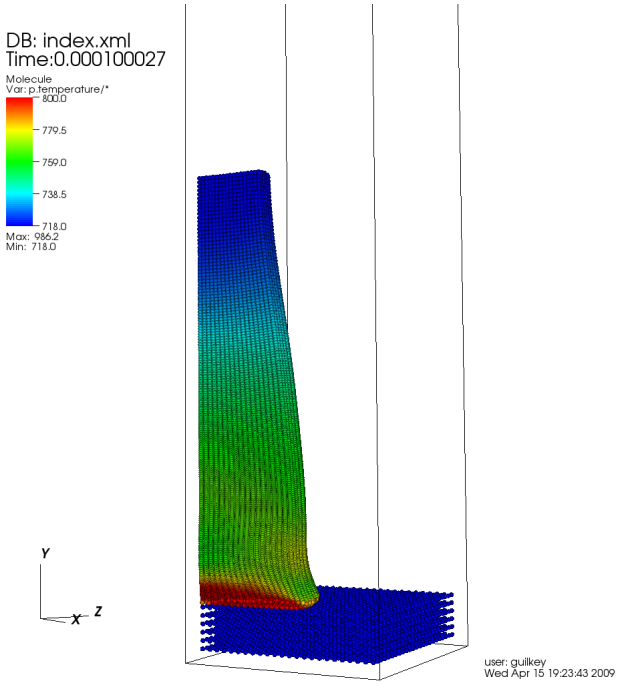



Figure 9.7: Taylor impact simulation with sliding between cylinder and target. Particles colored according to temperature.

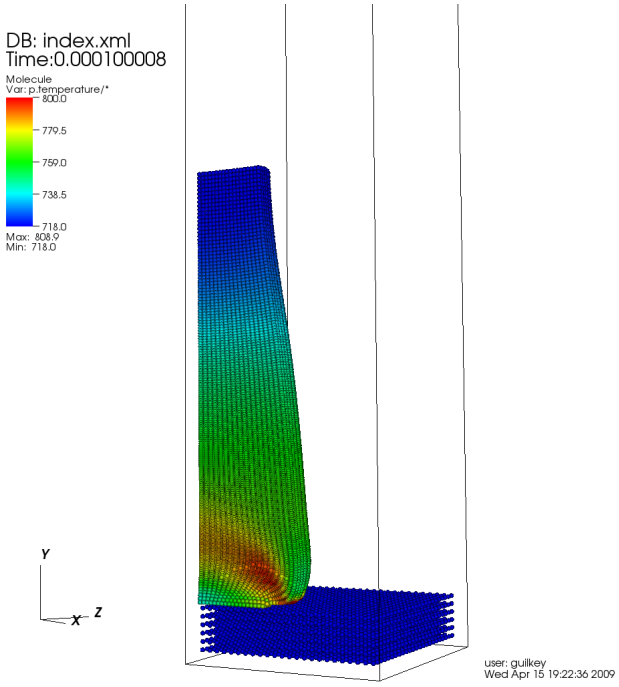



Figure 9.8: Taylor impact simulation with sliding prohibited between cylinder and target. Particles colored according to temperature.

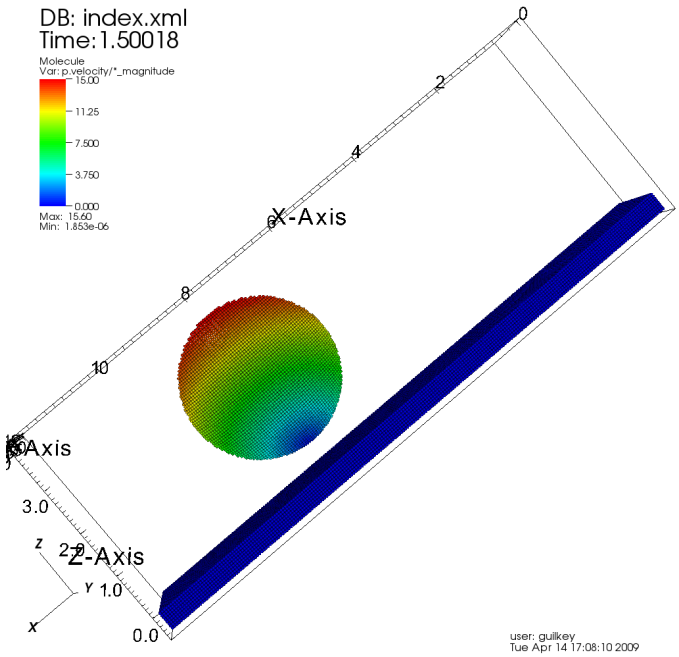



Figure 9.9: Sphere rolling down an "inclined" plane. The gravity vector is oriented at a 45 degree angle relative to the plane. Particles are colored by velocity magnitude. A friction coefficient of $\mu = 0.495$ is used. Particles are colored according to velocity magnitude, note that the particles at the top of the sphere are moving most rapidly, and those near the surface of the plane are basically stationary, as expected.

## Simulation Specifics

**Component used:** MPM

**Input file name:** inclinedPlaneSphere.ups

**Command used to run input file:**
sus inputs/UintahRelease/MPM/inclinedPlaneSphere.ups

**Simulation Domain:** 12.0 x 2.0 x 4.8 m

**Cell Spacing:**
.2 x .2 x .2 m (Level 0)

**Example Runtimes:**
2.7 hours (1 core, 3.16 GHz Xeon)

**Physical time simulated:** 2.2 seconds

**Associate VisIt session:** incplane.session

## Results

Figure 9.9 and Figure 9.10 show snapshots of the simulation, as the sphere is about halfway down the plane.

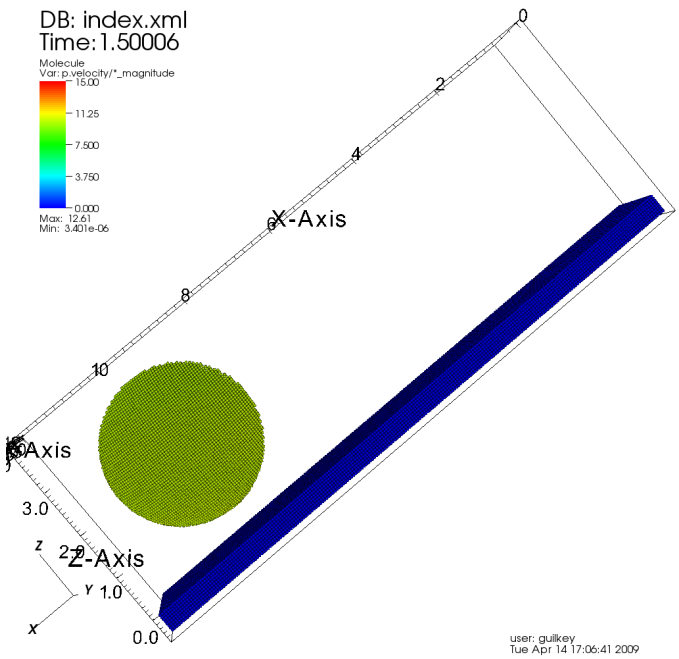



Figure 9.10: Sphere rolling down an "inclined" plane. The gravity vector is oriented at a 45 degree angle relative to the plane. Particles are colored by velocity magnitude. A friction coefficient of $\mu = 0.0$ is used. Particles are colored according to velocity magnitude. In this case, the particles throughout the sphere are moving at roughly the same velocity, because the sphere is sliding as it moves down the plane, as opposed to sticking and rolling.

## Crushing a Foam Microstructure

### Problem Description

This calculation demonstrates two important strength of MPM. The first is the ability to quickly generate a computational representation of complex geometries. The second is the ability of the method to handle large deformations, including self contact.

In particular, in this calculation a small sample of foam, the geometry for which was collected using microCT, is represented via material points. The sample is crushed to 87.5% compaction through the use of a rigid plate, which acts as a constant velocity boundary condition on the top of the sample. This calculation is a small example of those described in [13]. The geometry of the foam is created by image procesing the CT data, and based on the intensity of each voxel in the image data, the space represented by that voxel either recieves a particle with the material properties of the foam's constituent material, or is left as void space. This particle representation avoids the time consuming steps required to build a suitable unstructured mesh for this very complicated geometry.

### Simulation Specifics

**Component used:** MPM

**Input file name:** foam.ups

**Instruction to run input file:** First, copy foam.ups and foam.pts.gz to the same directory as sus. Adjust the number of patches in the ups file based on the number of processors available

to you for this run. First, uncompress the pts file:

```
gunzip foam.pts.gz
```

Then the command:

```
tools/pfs/pfs foam.ups
```

will divide the foam.pts file, which contains the geometric description of the foam, into number of patches smaller files, named foam.pts.0, foam.pts.1, etc. This is done so that for large simulations, each processor is only reading that data which it needs, and prevents the thrashing of the file system that would occur if each processor needed to read the entire pts file. This command only needs to be done once, or anytime the patch distibution is changed. Note that this step must be done even if only one processor is available.

To run this simulation:

```
mpirun -np NP sus foam.ups
```

where NP is the number of processors being used.

**Simulation Domain:** 0.2 X 0.2 X 0.2125 mm

**Number of Computational Cells:**
102 X 102 X 85 (Level 0)

**Example Runtimes:**
2.4 hours (4 cores, 3.16 GHz Xeon)

**Physical time simulated:** 3.75 seconds

**Associated VisIt session 1:** foam.iso.session

**Associated VisIt session 2:** foam.part.session

## Results

Figure 9.11 shows a snapshot of the simulation via isosurfacing, as the foam is at about 50% compaction.

Figure 9.12 shows a snapshot of the simulation via particles colored by equivalent stress as the foam is at about 60% compaction.

In this simulation, the reaction forces at 5 of the 6 computational boundaries are also recorded and can be viewed using a simple plotting package such as gnuplot. At each timestep, the internal force at each of the boundaries is accumulated and stored in "dat" files within the uda, e.g. BndyForce_zminus.dat. Because the reaction force is a vector, it is enclosed in square brackets which may be removed by use of a script in the inputs directory:

```
cd foam.uda.000
../inputs/ICE/Scripts/removeBraces BndyForce\_zminus.dat
gnuplot
gnuplot> plot "BndyForce\_zminus.dat" using 1:4
gnuplot> quit
```

These reaction forces are similar to what would be measured on a mechanical testing device, and help to understand the material behavior.

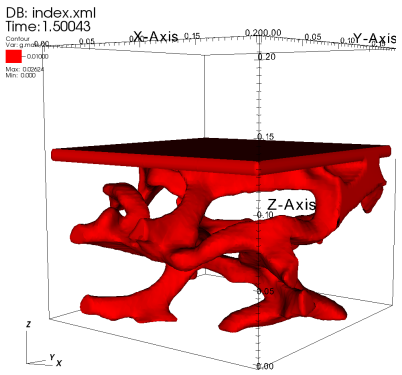



Figure 9.11: Compaction of a foam microstructure shown via isosurfacing.

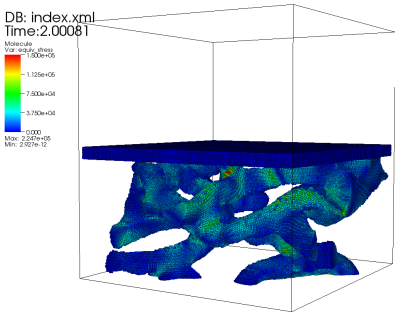



Figure 9.12: Compaction of a foam microstructure rendered as particles colored by equivalent stress.

## Hole in an Elastic Plate

### Problem Description

A flat plate with a hole in the center is loaded in tension. To achieve a quasi-static solution, the load is applied slowly and a viscous damping force is used to reduce transients in the solution. As such, this simulation demonstrates those two capabilities. Specifically, take note of:

```
<use_load_curves> true </use_load_curves>
<artificial_damping_coeff>1.0</artificial_damping_coeff>
```

in the `<MPM>` section of the input file, and:

```
<PhysicalBC>
  <MPM>
    <pressure>
      .
      .
      .
```

section below that.

### Simulation Specifics

**Component used:** MPM

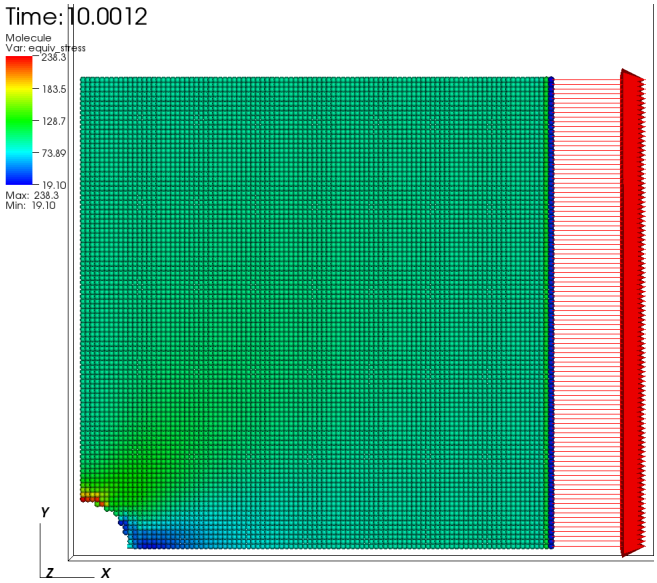



Figure 9.13: Elastic plate with a hole loaded in tension. Particles are colored by equivalent stress, vectors indicate applied load.

**Input file name:** holePlate.ups

**Command used to run input file:**
sus inputs/UintahRelease/MPM/holePlate.ups

**Simulation Domain:** 5.0 m x 5.0 m x 0.1 m

**Cell Spacing:**
0.1 m x 0.1 m x 0.1 m (Level 0)

**Example Runtimes:**
2 minutes (1 processor, Xeon 3.16 GHz)

**Physical time simulated:** 10 seconds

**Associate VisIt session:** holeInPlate.session

## Results

Figure 9.13 shows a snapshot of the equivalent stress throughout the plate, as well as the load applied to the vectors near the edge of the plate. Expected maximum stress is $300Pa$. The $238Pa$ maximum observed here is significantly lower, but upon doubling the resolution in the x and y directions, the maximum stress is $308Pa$. To recreate this image, select Controls in the upper left corner of the screen. Select Expressions, then click the New button. Now select Insert Function, then Tensor, then effective_tensor. The last step is to select Insert Variable, then Tensor, then p. stress.

# Tungsten Sphere Impacting a Steel Target

## Problem Description

A $1mm$ tungsten sphere with an initial velocity of $5000m/s$ impacts a steel target. Axisymmetric conditions are used in this case, conversion of the input file to the full 3D simulation is straightforward. The user may wish to do both simulations of both to gain confidence in the applicability of axisymmetry.

This simulation exercises the `elastic_plastic` constitutive model for the steel material. This includes sub-models for equations of state, variable shear modulus, melting, plasticity, etc. The tungsten is modeled using the `comp_neo_hook_plastic,` which is simple vonMises plasticity with linear hardening. One difficulty with using the more sophisticated models is that parameters can be difficult to find for many materials.

## Simulation Specifics

**Component used:** MPM

**Input file name:** WSphereIntoSteel.axi.ups

**Command used to run input file:**
sus inputs/UintahRelease/MPM/WSphereIntoSteel.axi.ups

**Simulation Domain:** 1.0 cm x 1.5 cm x axisymmetric

**Cell Spacing:**
0.333 mm x 0.333 mm x axisymmetry (Level 0)

**Example Runtimes:**
15 seconds (1 processor, Xeon 3.16 GHz)

**Physical time simulated:** 4 $\mu$seconds

**Associate VisIt session:** WSphereSteel.session

## Results

Figure 9.14 shows the initial configuration for this simulation, with particles colored by the magnitude of their velocity. Figure 9.15 shows the state of the simulation after $4\mu$seconds this simulation, with particles still colored by velocity magnitude.

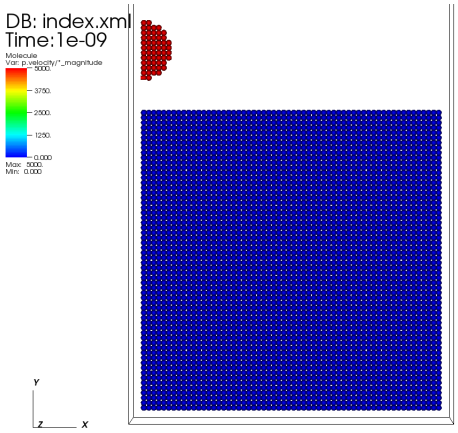



Figure 9.14: Initial configuration of hypervelocity impact of tungsten sphere into a steel target. Particles are colored by velocity magnitude.

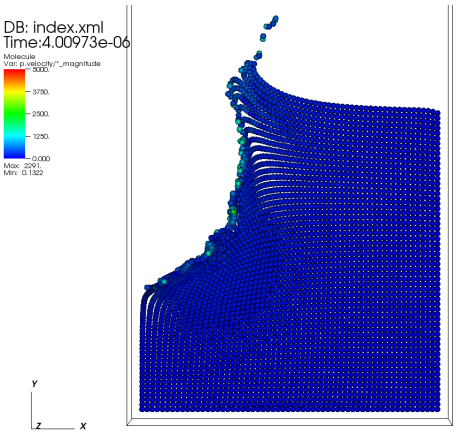



Figure 9.15: State of the tunsgsten and steel after 4$\mu$seconds. Particles are colored by velocity magnitude.

### 9.6.1 Method Of Manufactured Solutions (MMS)

There are three manufactured solutions available in Uintah for nonlinear elastic constitutive models. The input files are available in the `inputs/MPM` folder.

```
AA.ups (Axis Aligned MMS)
GenVortex.ups (Genralized Vortex MMS)
Ring_MMS.ups (Expanding Ring MMS)
```

All these input files have the following tag included in the `<MPM>` section of the input file:

```
<MPM>
   <RunMMSProblem>Name of the MMS</RunMMSProblem>
</MPM>
```

The exact solutions for these problems are available in `puda`, and the call to extract the error is

```
puda -AA_MMS_2 AA_MMS.uda (for AxisAligned MMS)
puda -GV_MMS GenVortex.uda (for Generalized Vortex MMS)
puda -ER_MMS Ring_MMS.uda (for Expanding Ring MMS)
```

The current implementation allows the user to add a new manufactured solution in the MPM component in a relatively-straight forward way. The current implementation of these manufactured solutions are located in src/CCA/Components/MPM/MMS folder. The following files require modifications either to change the exisiting MMS or add a new one.

1) src/CCA/Components/MPM/MPMFlags.cc
2) src/StandAlone/inputs/UPS_SPEC/mpm_spec.xml
3) src/CCA/Components/MPM/MMS/MMS.cc
4) src/CCA/Components/MPM/ConstitutiveModel/CNH_MMS.cc (Right now, all these MMS use the same constitutive model. User can change the constitutive model accordingly)
5) src/StandAlone/tools/puda/puda.cc

Following are the sequential steps to add a new MMS to the existing framework.
1) In MPMFlags.cc, add another if condition for the new MMS string in the following loop

```
if(d_mms_type=="AxisAligned"){
    d_mms_type = "AxisAligned";
  } else if(d_mms_type=="GeneralizedVortex"){
    d_mms_type = "GeneralizedVortex";
  } else if(d_mms_type=="ExpandingRing"){
    d_mms_type = "ExpandingRing";
  } else if(d_mms_type=="AxisAligned3L"){
    d_mms_type = "AxisAligned3L";
  }
}
```

2) Add the same string in the <RunMMSProblem> tag located in the mpm_spec.xml file.
3) There are two member functions available in MMS.cc.

```
MMS::initializeParticleForMMS
MMS::computeExternalForceForMMS
```

Similar to step 1, add another if condition in both the member functions for the new MMS. In the initializeParticleForMMS function, initialize the particle data at time $t = 0$, and in the computeExternalForceForMMS function, code the analytical body forces. The exisiting analytical solutions can be used as a guide.
4) Add the exact solution in the src/StandAlone/tools/puda folder. Look at the AA_MMS.cc, GV_MMS.cc, and ER_MMS.cc for reference. Add the option for the new MMS in the puda.cc file.
5) If the new MMS has a non-zero stress, necessary modifications needs to be made in the initializeCMData function located in that particular constitutive model (CNH_MMS.cc for reference).

# Bibliography


[1] F. H. Abed and G. Z. Voyiadjis. A consistent modified Zerilli-Armstrong flow stress model for bcc and fcc metals for elevated temperatures. *Acta Mechanica*, 175:1–18, 2005.

[2] B. Banerjee. Material point method simulations of fragmenting cylinders. In *Proc. 17th ASCE Engineering Mechanics Conference (EM2004)*, Newark, Delaware, 2004.

[3] B. Banerjee. MPM validation: Sphere-cylinder impact: Low resolution simulations. Technical Report C-SAFE-CD-IR-04-002, Center for the Simulation of Accidental Fires and Explosions, University of Utah, USA, 2004.

[4] B. Banerjee. Simulation of impact and fragmentation with the material point method. In *Proc. 11th International Conference on Fracture*, Turin, Italy, 2005.

[5] S. G. Bardenhagen, J. E. Guilkey, K. M Roessig, J. U. BrackBill, W. M. Witzel, and J. C. Foster. An improved contact algorithm for the material point method and application to stress propagation in granular material. *Computer Methods in the Engineering Sciences*, 2(4):509–522, 2001.

[6] S.G. Bardenhagen, J.U. Brackbill, and D. Sulsky. The material-point method for granular materials. *Comput. Methods Appl. Mech. Engrg.*, 187:529–541, 2000.

[7] S.G. Bardenhagen, J.E. Guilkey, K.M. Roessig, J.U. Brackbill, W.M. Witzel, and J.C. Foster. An improved contact algorithm for the material point method and application to stress propagation in granular material. *Computer Modeling in Engineering and Sciences*, 2:509–522, 2001.

[8] S.G. Bardenhagen and E.M. Kober. The generalized interpolation material point method. *Computer Modeling in Engineering and Sciences*, 5:477–495, 2004.

[9] Z. P. Bazant and T. Belytschko. Wave propagation in a strain-softening bar: Exact solution. *ASCE J. Engg. Mech*, 111(3):381–389, 1985.

[10] R. Becker. Ring fragmentation predictions using the gurson model with material stability conditions as failure criteria. *Int. J. Solids Struct.*, 39:3555–3580, 2002.

[11] J.U. Brackbill and H.M. Ruppel. Flip: A low-dissipation, particle-in-cell method for fluid flows in two dimensions. *J. Comp. Phys.*, 65:314–343, 1986.

[12] R. M. Brannon and S. Leelavanichkul. A multi-stage return algorithm for solving the classical damage component of constitutive models for rocks, ceramics, and other rock-like media. *Int. J. Fracture*, 163:133–149, 2010.

[13] A.D. Brydon, S.G. Bardenhagen, E.A. Miller, and G.T. Seidler. Simulation of the densification of real open-celled foam microstructures. *Journal of the Mechanics and Physics of Solids*, 53:2638–2660, 2005.

[14] L. Burakovsky, D. L. Preston, and R. R. Silbar. Analysis of dislocation mechanism for melting of elements: pressure dependence. *J. Appl. Phys.*, 88(11):6294–6301, 2000.

[15] S. R. Chen and G. T. Gray. Constitutive behavior of tantalum and tantalum-tungsten alloys. *Metall. Mater. Trans. A*, 27A:2994–3006, 1996.

[16] C. C. Chu and A. Needleman. Void nucleation effects in biaxially stretched sheets. *ASME J. Engg. Mater. Tech.*, 102:249–256, 1980.

[17] J. O. Coplien. *Advanced C++ Programming Styles and Idioms*. Addison-Wesley, Reading, MA, 1992.

[18] L. Cueto-Felgueroso, G. Mosqueira, F. Navarrina, and M. Casteleiro. On the galerkin formulation of the smoothed particle hydrodynamics method. *Int. J. Numer. Meth. Engng*, 60:1475–1512, 2004.

[19] N.P. Daphalapurkar, H. Lu, D. Coker, and R. Komanduri. Simulation of dynamic crack growth using the generalized interpolation material point (gimp) method. *Int. J. Fracture*, 143:79–102, 2007.

[20] D. C. Drucker. A definition of stable inelastic material. *J. Appl. Mech.*, 26:101–106, 1959.

[21] P. S. Follansbee and U. F. Kocks. A constitutive description of the deformation of copper based on the use of the mechanical threshold stress as an internal state variable. *Acta Metall.*, 36:82–93, 1988.

[22] D. M. Goto, J. F. Bingert, W. R. Reed, and R. K. Garrett. Anisotropy-corrected MTS constitutive strength modeling in HY-100 steel. *Scripta Mater.*, 42:1125–1131, 2000.

[23] J. E. Guilkey and J. A. Weiss. Implicit time integration for the material point method: Quantitative and algorithmic comparisons with the finite element method. *Int. J. Numer. Meth. Engng.*, 57(9):1323–1338, 2003.

[24] Y Guo and JA Nairn. Three-dimensional dynamic fracture analysis using the material point method. *Computer Modeling in Engineering and Sciences*, 6:295–308, 2004.

[25] A. L. Gurson. Continuum theory of ductile rupture by void nucleation and growth: Part 1. Yield criteria and flow rules for porous ductile media. *ASME J. Engg. Mater. Tech.*, 99:2–15, 1977.

[26] W. H. Gust. High impact deformation of metal cylinders at elevated temperatures. *J. Appl. Phys.*, 53(5):3566–3575, 1982.

[27] S. Hao, W. K. Liu, and D. Qian. Localization-induced band and cohesive model. *J. Appl. Mech.*, 67:803–812, 2000.

[28] F.H. Harlow. The particle-in-cell computing method for fluid dynamics. *Methods Comput. Phys.*, 3:319–343, 1963.

[29] R. Hill and J. W. Hutchinson. Bifurcation phenomena in the plane tension test. *J. Mech. Phys. Solids*, 23:239–264, 1975.

[30] K. G. Hoge and A. K. Mukherjee. The temperature and strain rate dependence of the flow stress of tantalum. *J. Mater. Sci.*, 12:1666–1672, 1977.

[31] G. R. Johnson and W. H. Cook. A constitutive model and data for metals subjected to large strains, high strain rates and high temperatures. In *Proc. 7th International Symposium on Ballistics*, pages 541–547, 1983.

[32] G. R. Johnson and W. H. Cook. Fracture characteristics of three metals subjected to various strains, strain rates, temperatures and pressures. *Int. J. Eng. Fract. Mech.*, 21:31–48, 1985.

[33] J. N. Johnson and F. L. Addessio. Tensile plasticity and ductile fracture. *J. Appl. Phys.*, 64(12):6699–6712, 1988.

[34] B.A. Kashiwa. A multifield model and method for fluid-structure interaction dynamics. Technical Report LA-UR-01-1136, Los Alamos National Laboratory, Los Alamos, 2001.

[35] U. F. Kocks. Realistic constitutive relations for metal plasticity. *Materials Science and Engrg.*, A317:181–187, 2001.

[36] F. L. Lederman, M. B. Salamon, and L. W. Shacklette. Experimental verification of scaling and test of the universality hypothesis from specific heat data. *Phys. Rev. B*, 9(7):2981–2988, 1974.

[37] J. Ma, H. Lu, and R. Komanduri. Structured mesh refinement in generalized interpolation material point method (gimp) for simulation of dynamic problems. *Computer Modeling in Engineering and Sciences*, 12:213–227, 2006.

[38] P. J. Maudlin and S. K. Schiferl. Computational anisotropic plasticity for high-rate forming applications. *Comput. Methods Appl. Mech. Engrg.*, 131:1–30, 1996.

[39] R. Menikoff and T.D. Sewell. Complete equation of state for beta-hmx and implications for initiation. In *APS Topical Conference Shock Compression of Condensed Matter*, Portland, Oregon, 2003. APS.

[40] M.-H. Nadal and P. Le Poac. Continuous model for the shear modulus as a function of pressure and temperature up to the melting point: analysis and ultrasonic validation. *J. Appl. Phys.*, 93(5):2472–2480, 2003.

[41] S. Nemat-Nasser. Rate-independent finite-deformation elastoplasticity: a new explicit constitutive algorithm. *Mech. Mater.*, 11:235–249, 1991.

[42] S. Nemat-Nasser and D. T. Chung. An explicit constitutive algorithm for large-strain, large-strain-rate elastic-viscoplasticity. *Comput. Meth. Appl. Mech. Engrg*, 95(2):205–219, 1992.

[43] P. Perzyna. Constitutive modelling of dissipative solids for localization and fracture. In Perzyna P., editor, *Localization and Fracture Phenomena in Inelastic Solids: CISM Courses and Lectures No. 386*, pages 99–241. SpringerWien, New York, 1998.

[44] J.-P Poirier. *Introduction to the Physics of the Earth's Interior*. Cambridge University Press, Cambridge, UK, 1991.

[45] D. L. Preston, D. L. Tonks, and D. C. Wallace. Model of plastic deformation for extreme loading conditions. *J. Appl. Phys.*, 93(1):211–220, 2003.

[46] S. Ramaswamy and N. Aravas. Finite element implementation of gradient plasticity models Part I: Gradient-dependent yield functions. *Comput. Methods Appl. Mech. Engrg.*, 163:11–32, 1998.

[47] S. Ramaswamy and N. Aravas. Finite element implementation of gradient plasticity models Part II: Gradient-dependent evolution equations. *Comput. Methods Appl. Mech. Engrg.*, 163:33–53, 1998.

[48] G. Ravichandran, A. J. Rosakis, J. Hodowany, and P. Rosakis. On the conversion of plastic work into heat during high-strain-rate deformation. In *Proc. , 12th APS Topical Conference on Shock Compression of Condensed Matter*, pages 557–562. American Physical Society, 2001.

[49] J. W. Rudnicki and J. R. Rice. Conditions for the localization of deformation in pressure-sensitive dilatant materials. *J. Mech. Phys. Solids*, 23:371–394, 1975.

[50] J. C. Simo and T. J. R. Hughes. *Computational Inelasticity*. Springer-Verlag, New York, 1998.

[51] JC Simo and TJR Hughes. *Computational Inelasticity*. Springer-Verlag, New York, 1998.

[52] D. J. Steinberg. Equation of state and strength properties of selected materials. Technical Report UCRL-MA-106439, Lawrence Livermore National Laboratory, Livermore, California, 1991.

[53] D. J. Steinberg, S. G. Cochran, and M. W. Guinan. A constitutive model for metals applicable at high-strain rate. *J. Appl. Phys.*, 51(3):1498–1504, 1980.

[54] D. J. Steinberg and C. M. Lund. A constitutive model for strain rates from $10^{-4}$ to $10^{6}$ $s^{-1}$. *J. Appl. Phys.*, 65(4):1528–1533, 1989.

[55] D. Sulsky, Z. Chen, and H.L. Schreyer. A particle method for history dependent materials. *Comput. Methods Appl. Mech. Engrg.*, 118:179–196, 1994.

[56] D. Sulsky and H.L. Schreyer. Axisymmetric form of the material point method with applications to upsetting and taylor impact problems. *Computer Methods in Applied Mechanics and Engineering*, 139:409–429, 1996.

[57] D. Sulsky, S. Zhou, and H.L. Schreyer. Application of a particle-in-cell method to solid mechanics. *Computer Physics Communications*, 87:236–252, 1995.

[58] V. Tvergaard and A. Needleman. Analysis of the cup-cone fracture in a round tensile bar. *Acta Metall.*, 32(1):157–169, 1984.

[59] V. Tvergaard and A. Needleman. Ductile failure modes in dynamically loaded notched bars. In J. W. Ju, D. Krajcinovic, and H. L. Schreyer, editors, *Damage Mechanics in Engineering Materials: AMD 109/MD 24*, pages 117–128. American Society of Mechanical Engineers, New York, NY, 1990.

[60] Y. P. Varshni. Temperature dependence of the elastic constants. *Physical Rev. B*, 2(10):3952–3958, 1970.

[61] P. C. Wallstedt and J. E. Guilkey. An evaluation of explicit time integration schemes for use with the generalized interpolation material point method. *J. Comp. Phys.*, 227:9628–9642, 2008.

[62] L. H. Wang and S. N. Atluri. An analysis of an explicit algorithm and the radial return algorithm, and a proposed modification, in finite elasticity. *Computational Mechanics*, 13:380–389, 1994.

[63] M. L. Wilkins. *Computer Simulation of Dynamic Phenomena*. Springer-Verlag, Berlin, 1999.

[64] F. J. Zerilli. Dislocation mechanics-based constitutive equations. *Metall. Mater. Trans. A*, 35A:2547–2555, 2004.

[65] F. J. Zerilli and R. W. Armstrong. Dislocation-mechanics-based constitutive relations for material dynamics calculations. *J. Appl. Phys.*, 61(5):1816–1825, 1987.

[66] F. J. Zerilli and R. W. Armstrong. Constitutive relations for the plastic deformation of metals. In *High-Pressure Science and Technology - 1993*, pages 989–992, Colorado Springs, Colorado, 1993. American Institute of Physics.

[67] F. J. Zerilli and Armstrong R. W. A constitutive equation for the dynamic deformation behavior of polymers. *J. Mater. Sci.*, 42:4562–4574, 2007.

[68] M. A. Zocher, P. J. Maudlin, S. R. Chen, and E. C. Flower-Maudlin. An evaluation of several hardening models using Taylor cylinder impact data. In *Proc. , European Congress on Computational Methods in Applied Sciences and Engineering*, Barcelona, Spain, 2000. ECCOMAS.

# Chapter 10

# MPMICE

## 10.1 Introduction

MPMICE is a marriage of the multi-material ICE method, described in Section 8 and MPM, described in Section 9. The equations of motion solved for both fluid and solid are essentially the same, although the physical behavior of these two states of matter differ, largely due to their constitutive relationships. MPM is used to track the evolution of solid materials in a Lagrangian frame of reference, while fluids are evolved in the Eulerian frame.

## 10.2 Theory - Algorithm Description

At this time, the reader is directed to the manuscript by Guilkey, Harman and Banerjee [3] for the theoretical and algorithmic description of the method.

## 10.3 Solid State Kinetic Models

A generalized reaction model for solid state kinetics based on the assumption that the temperature dependence of the rate can be separated from the reaction model as embodied by the equation:

$$\frac{d\alpha}{dt} = k(T)f(\alpha) \tag{10.1}$$

is implemented in Uintah and named **SolidReactionModel**. To use the **SolidReactionModel** one must specify both a temperature dependent rate constant model and a rate model. The model is a grid based model, so should work with both MPM and ICE materials. In the case where an MPM material is used as the reactant or product, the thermodynamic quantities that are interpolated to the grid are used to calculate reaction rates.

Additional rate constant and rate models may be added by either subclassing **RateConstantModel** or **RateModel**. Examples of this can be found in the **src/CCA/Components/Models/SolidReactionModel** directory. Two models currently exist for the rate constant $k(T)$, Arrhenius and Modified Arrhenius. These two models have the forms:

$$k(T) = Ae^{\frac{-E_a}{RT}} \tag{10.2}$$

and:

$$k(T) = AT^b e^{\frac{-Ea}{RT}} \tag{10.3}$$

Various rate models of different classes exist for use. These classes include reaction-order models, diffusion models, geometrical contraction models and nucleation models. The following table shows the various models available. Models were taken from [9] and [4].

| **Model** | $f(\alpha)$ | **Uintah 'type'** |
|---|---|---|
| **Reaction-order Models** | | |
| Nth Order | $(1-\alpha)^n$ | NthOrder |
| **Diffusion Models** | | |
| 1-D Diffusion | $1/2\alpha$ | Diffusion |
| 2-D Diffusion | $-1/ln(1-\alpha)$ | Diffusion |
| 3-D Diffusion | $3/2(1-\alpha)^{2/3}(1-(1-\alpha)^{1/3})$ | Diffusion |
| 4-D Diffusion | $3/2(1/\alpha^{1/3}-1)$ | Diffusion |
| **Geometrical Contraction Models** | | |
| Contracting Cylinder | $2\sqrt{1-\alpha}$ | ContractingCylinder |
| Contracting Sphere | $3(1-\alpha)^{2/3}$ | ContractingSphere |
| **Nucleation Models** | | |
| Power | $a\alpha^b$ | Power |
| Avarami-Erofe'ev | $a(1-\alpha)(-ln(1-\alpha))^b$ | AvaramiErofeev |

The input file specification is as follows:

```
<Models>
  <Model type="SolidReactionModel">
    <RateConstantModel type="type">
       ....
    </RateConstantModel>
    <RateModel type="type">
       ....
    </RateModel>
  </Model>
<Models>
```

Here, the type attribute for **RateConstantModel** should be either **Arrhenius** or **ModifiedArrhenius** which both take an activation energy **Ea**, and frequency factor **A**. In addition, the modified Arrhenius model takes a temperature dependence exponent, **b**.

The specification of type attribute for **RateModel** should be one of those listed in the previous table. The **NthOrder** model must have a positive integral value for the reaction order **n**. The **Diffusion** based models require a **dimension** value that should be 1, 2, 3 or 4 depending on the dimensionality of the desired rate model. Both geometric contraction models take no additional input parameters. Both nucleation based models, **Power** and **AvaramiErofeev**, take both an **a** and a **b** input parameter.

## 10.4 HE Reaction Models

Three models exist for reaction of high explosive materials. Each simulation using one of these models utilize MPMICE's material interactions as its foundation. The components work by taking several material specific constants as well as a reactant and product material from the model input section of the .ups file. Following are brief descriptions of each model, as well as their input parameters.

### 10.4.1 Simple Burn

Simple Burn, as the name implies, is a simple model of combustion of HMX based on the rate equation:

$$\dot{m} = AP^{0.778} \tag{10.4}$$

Where $\dot{m}$ is the mass flux, $P$ is the pressure and $n$ is the pressure dependence coefficient. The pressure coefficient in Equation (10.4) is that of HMX. The models input section for a Simple Burn simulation takes the form:

```
<Models>
  <Model type="Simple_Burn">
    <fromMaterial> reactant      </fromMaterial>
    <toMaterial>   product       </toMaterial>
    <Active>       true          </Active>
    <ThresholdTemp>       450.0  </ThresholdTemp>
    <ThresholdPressure> 50000.0  </ThresholdPressure>
    <Enthalpy>        2000000.0  </Enthalpy>
    <BurnCoeff>            75.3  </BurnCoeff>
    <refPressure>      101325.0  </refPressure>
  </Model>
</Models>
```

The first two tags take names of materials previously defined in the input file, defining both reactant and product used by the model. See Section 8.3.4 and 9.5.4 for in depth description for defining materials. <Active> is a debugging parameter that takes a boolean value indicating whether the model is on (i.e. the actual computations take place during the timestep). True is the value to set for <Active> in most situations. Each of the other parameters take double values. Threshold temperature and pressure tags define two criteria the cell must have in order to be flagged burning. The reference pressure is used to scale the cell centered pressure as well as make it an unitless value. The burn coefficient corresponds to $A$ in the rate equation. Enthalpy is simply the enthalpy value for conversion of reactant to product.

### 10.4.2 Steady Burn

Steady Burn is a more accurate model than Simple Burn. It is based on WSB model of combustion developed by Ward, Son and Brewster in [11]. WSB is based on a simplified two-step chemical model with an initial zero-order, thermally activated ($E_c > 0$), mildly exothermic, solid-to-gas reaction, modeled as a thermal decomposition of the solid:

$$A(solid) \rightarrow B(gas) \tag{10.5}$$

Intermediate $B$, in the presence of any gas phase collision partner $M$, reacts in a highly exothermic fashion producing a flame. This step is modelled as a second-order, gas phase, free radical chain reaction based on the assumption that $E_g = 0$:

$$B(gas) + M(gas) \rightarrow C(gas) + M(gas) \tag{10.6}$$

As such, this second equation represents the reaction in the gas phase that causes heat convection back to the surface that activate the first reaction. In Steady Burn, a solution is found by iteratively solving two equations: one for mass burning rate $\dot{m}$ and one for surface temperature $T_s$. Mass flux is initially solved with an assumed value $T_s$ (in the model set to $850.0K$) using WSB:

$$\dot{m}\left(T_s\right) = \sqrt{\frac{\kappa_c \rho_c A_c R T_s^2 \exp\left(\frac{-E_c}{RT_s}\right)}{C_p E_c \left(T_s - T_0 - \frac{Q_c}{2C_p}\right)}} \tag{10.7}$$

The solution to this equation is used to refine the surface temperature and vice-versus until a self-consistent solution for surface temperature and mass flux has been found. The surface temperature equation takes the form:

$$T_s(\dot{m}, P) = T_0 + \frac{Q_c}{C_p} + \frac{Q_g}{Cp\left(1 + \frac{x_g\left(\dot{m}, P\right)}{x_{cd}(\dot{m})}\right)} \tag{10.8}$$

$x_g$ in the third term of Equation (10.8) is the flame standoff distance, computed from:

$$x_g\left(\dot{m}, P\right) = \frac{2x_{cd}\left(\dot{m}\right)}{\sqrt{1 + D_a\left(\dot{m}, P\right)} - 1} \tag{10.9}$$

where $x_{cd}$ and $D_a$ are the convective-diffusive length and Damkohler number, respectively:

$$x_{cd}\left(\dot{m}\right) = \frac{\kappa_g}{\dot{m} C_p} \tag{10.10}$$

$$D_a\left(\dot{m}, P\right) = \frac{4 B_g M C_p P^2}{R^2 \kappa_g} x_{cd}\left(\dot{m}\right)^2 \tag{10.11}$$

WSB model is valid as a 1D model, but needs extension to work in a 3D multimaterial CFD environment. As such, Steady Burn is WSB extended with logic for ignition of energetic materials and computation of surface area for burning cells. Ignition of a cell is based on three criteria:

- The cell must contain one particle of energetic solid
- The cell is near a surface of an energetic solid (e.g. ratio of minimum node-centered mass to maximum node-centered mass is less than 0.7)

- One neighboring cell must have at most two particles of energetic material

If a cell is ignited, the model will be applied and mass will be transferred from reactant material to product material. Total mass burned is computed using mass flux $\dot{m}$, $\Delta t$ of the timestep and the calculated surface area, found using:

$$A = \frac{\delta x \delta y \delta z}{\delta x|g_x| + \delta y|g_y| + \delta z|g_z|} \tag{10.12}$$

where $\delta x$, $\delta y$, and $\delta z$ are the dimensions of the cell and components of $\overrightarrow{g}$ are the normalized density gradients of the particle mass in a cell. A more thorough examination of Steady Burn can be read about in [12].

The following table describes the input parameters for Steady Burn. The final column of the table indicates parameters for combustion of HMX.

| Steady Burn Input Parameters | | | |
|---|---|---|---|
| **Tag** | **Type** | **Description** | **HMX Value** |
| <fromMaterial> | String | 'Name' of reactant material (mass source) | |
| <toMaterial> | String | 'Name' of product material (mass sink) | |
| <IdealGasConst> | double | Ideal gas constant ($R$) | $8.314 J/(K \times mol)$ |
| <PreExpCondPh> | double | Condensed phase pre-exponential coefficient ($A_c$) | $1.637 \times 10^{15} s^{-1}$ |
| <ActEnergyCondPh> | double | Condensed phase activation energy ($E_c$) | $1.76 \times 10^5 J/mol$ |
| <PreExpGasPh> | double | Gas phase frequency factor ($B_g$) | $1.6 \times 10^{-3} m^3/(kg \times s \times K)$ |
| <CondPhaseHeat> | double | Condensed phase heat release per unit mass ($Q_c$) | $4.0 \times 10^5 J/kg$ |
| <GasPhaseHeat> | double | Gas phase heat release per unit mass ($Q_g$) | $3.018 \times 10^6 J/kg$ |
| <HeatConductGasPh> | double | Thermal conductivity of gas ($\kappa_g$) | $0.07 W/(m \times K)$ |
| <HeatConductCondPh> | double | Thermal conductivity of condensed phase ($\kappa_c$) | $0.02 W/(m \times K)$ |
| <SpecificHeatBoth> | double | Specific heat at constant pressure ($c_p$) | $1.4 \times 10^3 J/(kg \times K)$ |
| <MoleWeightGasPh> | double | Molecular weight of gas ($W$) | $3.42 \times 10^{-2} kg/mol$ |
| <BoundaryParticles> | int | Max # of particles a cell can have and be burning | Resolution dependent |
| <ThresholdPressure> | double | Threshold pressure cell must have $\geq$ to burn mass | $50000 Pa$ |
| <IgnitionTemp> | double | Temperature cell must have $\geq$ to be burning | $550 K$ |

### 10.4.3 Unsteady Burn

Unsteady Burn is a model developed at the University of Utah as an extension of Steady Burn to better represent mass burning rates when pressure at the burning surface fluctuates. A pressure-coupled response is accounted for in the model such that, qualitatively a pressure increase causes gas phase reaction rates to increase as well as move the gas phase reactions closer to the burning surface. Increase of near surface gas phase reactions increases the rate of thermally activated solid state reactions, ultimately causing a higher steady burn rate. Unsteady Burn more accurately models the transition from low pressure to high pressure than Steady Burn by taking into account the initially overshot burn rate at the time when the pressure increases, and the relaxation period to steady burn rate. Similarly, Unsteady Burn models undershot pressures during pressure drops.

The model is an extension of Steady Burn by partial decoupling of the gas phase and solid state Equations (10.7) and (10.8). An expression for the temperature gradient of the solid:

$$\beta = (T_s - T_0)\,\frac{mc_p}{\kappa_c} \tag{10.13}$$

is reaarranged for $(T_s - T_0)$ and substituted in Equation (10.7) leading to the quadradic equation:

$$\dot{m}^2 - \frac{2\beta\kappa_c}{Q_c}\dot{m} + \frac{2A_c R T_s^2 \kappa_c \rho_c}{E_c Q_C}\exp\left(\frac{-E_c}{RT_s}\right) = 0 \tag{10.14}$$

which allows independent tracking of temperature gradient $\beta$ and surface temperature $T_s$. The gas phase response is computed using a runnning average of $T_s$ as it approaches the steady burning value. A solid state response is obtained by computing a running average of $\beta$ as it approaches the steady burning value. A slow relaxation time for $\beta$ and a fast relaxation time for $T_s$ models the overshoot or undershoot in burn rate. Burning criteria for a cell is the same as Steady Burn. For more information on Unsteady Burn see [12].

The following table describes the input parameters for Unsteady Burn.

| Unsteady Burn Input Parameters | | |
|---|---|---|
| **Tag** | **Type** | **Description** |
| <fromMaterial> | String | 'Name' of reactant material (mass source) |
| <toMaterial> | String | 'Name' of product material (mass sink) |
| <IdealGasConst> | double | Ideal gas constant ($R$) |
| <PreExpCondPh> | double | Condensed phase pre-exponential coefficient ($A_c$) |
| <ActEnergyCondPh> | double | Condensed phase activation energy ($E_c$) |
| <PreExpGasPh> | double | Gas phase frequency factor ($B_g$) |
| <CondPhaseHeat> | double | Condensed phase heat release per unit mass ($Q_c$) |
| <GasPhaseHeat> | double | Gas phase heat release per unit mass ($Q_g$) |
| <HeatConductGasPh> | double | Thermal conductivity of gas ($\kappa_g$) |
| <HeatConductCondPh> | double | Thermal conductivity of condensed phase ($\kappa_c$) |
| <SpecificHeatBoth> | double | Specific heat at constant pressure ($c_p$) |
| <MoleWeightGasPh> | double | Molecular weight of gas ($W$) |
| <BoundaryParticles> | int | Max # of particles a cell can have and be burning |
| <BurnrateModCoef> | double | if $\neq$1.0, scale unsteady rate with steady rate as $\dot{m}_u = \dot{m}_s \left(\frac{\dot{m}_u}{\dot{m}_s}\right)^{B_m}$ |
| <CondUnsteadyCoef> | double | Coefficient for condensed phase pressure response relaxation |
| <GasUnsteadyCoef> | double | Coefficient for gas phase pressure response relaxation |
| <ThresholdPressure> | double | Threshold pressure cell must be $\geq$ to burn mass |
| <IgnitionTemp> | double | Temperature cell must be at $\geq$ to be burning |

### 10.4.4 Ignition & Growth

The Ignition & Growth model created by Lee and Tarver [5] to simulate shock-to-detonation transitions in condensed explosives. The rate equation takes the form:

$$\frac{dF}{dt} = I(1-F)^b \left(\frac{\rho}{\rho_0} - 1 - a\right)^x + G_1(1-F)^c F^d P^y + G_2(1-F)^e F^g P^z \tag{10.15}$$

where $F$ is the extent of reaction in a cell, $P$ is the pressure, $\rho$ and $\rho_0$ are the current and initial density of the explosive and $I$, $G_1$, $G_2$, $a$, $b$, $c$, $d$, $e$, $g$, $x$, $y$ and $z$ are constant fit parameters. The three terms are the ignition, growth and completion terms respectively. The ignition term is used to emulate hot-spot formation and strengthening and runs over $0 < F < F_{igmax}$ where $F_{igmax}$ is a constant. The growth term is used as a fast term for growth of the shock front and runs over $0 < F < F_{G1max}$ where $F_{G1max}$ is a constant. The completion term is a slow term used to model the precipitation of solid carbon at the end of reaction and runs over $F_{G2min} < F < 1$ where $F_{G2min}$ is a constant. The model input parameters are detailed in the following table.

| **Ignition & Growth Input Parameters** | | |
|---|---|---|
| **Tag** | **Type** | **Description** |
| <fromMaterial> | String | 'Name' of reactant material (mass source) |
| <toMaterial> | String | 'Name' of product material (mass sink) |
| <I> | double | Ignition rate constant (hot-spot frequency) |
| <G1> | double | Growth rate constant |
| <G2> | double | Completion rate constant |
| <a> | double | Ignition constant |
| <b> | double | Ignition exponent |
| <c> | double | Growth exponent |
| <d> | double | Growth exponent |
| <e> | double | Completion exponent |
| <g> | double | Completion exponent |
| <x> | double | Ignition density exponent |
| <y> | double | Growth pressure exponent |
| <z> | double | Completion pressure exponent |
| <Figmax> | double | Maximum reaction extent for hot-spot (ignition) term |
| <FG1max> | double | Maximum reaction extent for fast (growth) term |
| <FG2min> | double | Minimum reaction extent for slow (completion) term |
| <rho0> | double | The initial density of the explosive |
| <E0> | double | The energy of detonation |
| <ThresholdPressure> | double | Reaction is allowed to occur above this pressure |

### 10.4.5 JWL++

The JWL++ model by Souers et al. [6] is related to the Ignition and Growth model (see Section 10.4.4), but much simplified in its form for ease of rate constant fit. The rate is described by:

$$\frac{dF}{dt} = G(1-F)P^b \tag{10.16}$$

where $F$ is the extent of reaction, $P$ is the pressure and $G$ and $b$ are fit constants. Several other forms of the JWL++ model have been formulated, however this is the only one that is currently supported in Uintah. Input paramters are shown in the follwoing table.

| **Ignition & Growth Input Parameters** | | |
|---|---|---|
| **Tag** | **Type** | **Description** |
| <fromMaterial> | String | 'Name' of reactant material (mass source) |
| <toMaterial> | String | 'Name' of product material (mass sink) |
| <G> | double | Growth rate constant |
| <b> | double | Completion rate constant |
| <rho0> | double | The initial density of the explosive |
| <E0> | double | The energy of detonation |
| <ThresholdPressure> | double | Reaction is allowed to occur above this pressure |
| <ThresholdVolFrac> | double | (Optional; default 0.01) Minimum volume of explosive in a cell for reaction |

### 10.4.6 DDT0

Deflagration-to-detonation model 0 (DDT0) was the first incarnation of a model that contains two rate models to represent three different reaction modes. The model is capable of surface burning, convective burning and detonation. Burning is accomplished using the same model as Simple Burn (see Section 10.4.1). Detonation is accounted for by the JWL++ model (see Section 10.4.5) presented by P. Clark Souers [6]. The simple threshold pressure separates detonation and deflagration regimes. In addition, a crack-size dependent model may be optionally used to allow convective burning to occur in the bulk material. The crack-size threshold is computed via an expression fit by Berghout et al. [1] The parameters are presented in the following table.

| **DDT0 Input Parameters** | | |
|---|---|---|
| **Tag** | **Type** | **Description** |
| <fromMaterial> | String | 'Name' of reactant material (mass source) |
| <toMaterial> | String | 'Name' of product material (mass sink) |
| <G> | double | Rate constant for detonation (JWL++ model) |
| <b> | double | Pressure exponent for detonation (JWL++ model) |
| <E0> | double | Energy of reaction for detonation (JWL++ model) |
| <ThresholdPressureJWL> | double | Threshold pressure for onset of detonation |
| <ThresholdVolFrac> | double | (Optional; default 0.01) Minimum volume fraction of reactant for detonation to occur in a cell |
| <Enthalpy> | double | Energy of reaction for deflagration (Simple Burn model) |
| <BurnCoeff> | double | Rate constant for deflagration (Simple Burn model) |
| <refPressure> | double | Reference pressure for deflagration (Simple Burn model) |
| <ThresholdTemp> | double | Threshold temperature for combustion (Simple Burn model) |
| <TresholdPressureSB> | double | Threshold pressure required for combustion (Simple Burn model) |
| <useCrackModel> | boolean | (Optional; default false) Switch that allows convective burning |
| <Gcrack> | double | (Required for 'useCrackModel') Rate constant for convective deflagration |
| <CrackVolThreshold> | double | (Optional; default 1e-14) Volume fraction of reactant above temperature needed for convective deflagration |
| <nCrack> | double | (Required for 'useCrackModel') Pressure exponent for convective deflagration |

### 10.4.7 DDT1

Deflagration-to-detonation model 1 (DDT1) [7] was the second incarnation of a model that contains two rate models to represent three different reaction modes. The model is capable of surface burning, convective burning and detonation. Burning is accomplished using the same model as Steady Burn (see Section 10.4.2). Detonation is accounted for by the JWL++ model (see Section 10.4.5) presented by P. Clark Souers [6]. The simple threshold pressure separates detonation and deflagration regimes. In addition, a crack-size dependent model may be optionally used to allow convective burning to occur in the bulk material. The crack-size threshold is computed via an expression fit by Berghout et al. [1] The parameters are presented in the following table.

| DDT1 Input Parameters | | |
|---|---|---|
| **Tag** | **Type** | **Description** |
| <fromMaterial> | String | 'Name' of reactant material (mass source) |
| <toMaterial> | String | 'Name' of product material (mass sink) |
| <burnMaterial> | String | (Optional; default 'toMaterial') 'Name' of product material for deflagration |
| <G> | double | Rate constant for detonation (JWL++ model) |
| <b> | double | Pressure exponent for detonation (JWL++ model) |
| <E0> | double | Energy of reaction for detonation (JWL++ model) |
| <ThresholdPressureJWL> | double | Threshold pressure for detonation |
| <ThresholdVolFrac> | double | (Optional; default 0.01) Minimum volume fraction of reactant for detonation to occur in a cell |
| <IdealGasConst> | double | Ideal gas constant ($R$) |
| <PreExpCondPh> | double | Condensed phase pre-exponential coefficient ($A_c$) |
| <ActEnergyCondPh> | double | Condensed phase activation energy ($E_c$) |
| <PreExpGasPh> | double | Gas phase frequency factor ($B_g$) |
| <CondPhaseHeat> | double | Condensed phase heat release per unit mass ($Q_c$) |
| <GasPhaseHeat> | double | Gas phase heat release per unit mass ($Q_g$) |
| <HeatConductGasPh> | double | Thermal conductivity of gas ($\kappa_g$) |
| <HeatConductCondPh> | double | Thermal conductivity of condensed phase ($\kappa_c$) |
| <SpecificHeatBoth> | double | Specific heat at constant pressure ($c_p$) |
| <MoleWeightGasPh> | double | Molecular weight of gas ($W$) |
| <BoundaryParticles> | int | Max # of particles a cell can have and be burning |
| <ThresholdPressure> | double | Threshold pressure cell must have $\geq$ to burn mass |
| <IgnitionTemp> | double | Temperature cell must have $\geq$ to be burning |
| <TresholdPressureSB> | double | Threshold pressure required for combustion |
| <useCrackModel> | boolean | (Optional; default false) Switch that allows convective burning |

### Dynamic Output Intervals

The dynamic output intervals section is used to change how frequently the output interval and check point interval are saved. The intervals can be changed when the pressure is a cell with reactant is greater than the set pressure threshold and/or when detonation is detected. This only works when using the DDT1 reaction model.

| Dynamic Output Intervals Input Parameters | | |
|---|---|---|
| **Tag** | **Type** | **Description** |
| <PressureThreshold> | double | Pressure threshold to switch output interval (Pa) |
| <newOutputInterval> | double | Output interval after switch is reached |
| <newCheckPointInterval> | double | Check point interval after switch is reached |
| <remainingTimesteps> | double | Number of timesteps after detonation when the simulation will shut down |

The input file specification is as follows:

```
<Models>
....
<adjust_IO_intervals>
<PressureSwitch>
<PressureThreshold> 4.0e9 </PressureThreshold>
<newOutputInterval> 1e-7 </newOutputInterval>
<newCheckPointInterval> 1e-7 </newCheckPointInterval>
</PressureSwitch>

<DetonationDetected>
<remainingTimesteps> 20 </remainingTimesteps>
<newOutputInterval> 1e-6</newOutputInterval>
<newCheckPointInterval> 1e-6 </newCheckPointInterval>
</DetonationDetected>
</adjust_IO_intervals>
....
</Models>
```

### Induction Time

To accurately represent the propagation of deflagration an "induction" period, or wait time was introduced. This induction period is the time a cell must wait before reactant mass would be converted to product gas using the WSB burn model [11]. The induction time is dependent on the surrounding pressure and the size of the cell. The induction time model is based off experimentally determined flame propagation on a surface as seen in equation 10.17 [10], where P is the dimensionless pressure ($p/p_0$) and $S_f$ is the flame propagation in $cm/s$. Equation 10.17 is used in determining the induction time as seen by equation 10.18 where x is the size of the cell and A is a constant used to speed up or slow down the propagation of convective deflagration. A varies depending on the length of the cell but should be used to give the correct propagation of convective deflagration only. The model determines which direction the flame is coming from in turn adjusting A according to the angle of penetration. For instance if the flame is propagating along a surface but not into the solid A = 1 but if the flame is propagating directly into the surface A equals the value set in the input file.

$$S_f = 0.259P^{0.538} \tag{10.17}$$

$$\tau = \frac{\Delta x A}{S_f} \tag{10.18}$$

| Dynamic Output Intervals Input Parameters | | |
|---|---|---|
| **Tag** | **Type** | **Description** |
| <useIndcutionTime> | boolean | (Optional; default false) Switch that slows down deflagration propagation |
| <IgnitionConst> | double | Constant used to speed up or slow down the convective deflagration propagation |
| <PressureShift> | double | Pressure used to make dimensionless pressure ($p_0$) |
| <ExponentialConst> | double | Exponential constant used in flame propagation equation |
| <PreexopConst> | double | Pre-exponential constant used in flame propagation equation |

The input file specification is as follows:

```
<Models>
....
  <useInductionTime>  true </useInductionTime>
  <IgnitionConst> 0.00009 </IgnitionConst>
  <PressureShift> 1.0e5 </PressureShift>
  <ExponentialConst> 0.538 </ExponentialConst>
  <PreexpoConst> 0.00259 </PreexpoConst>
....
</Models>
```

The pressure shift, exponential and pre-exponential constants presented above are given by Son et al. [10] for experimentally determined values for the explosive PBX9501.

## 10.5 Examples

### Mach 2 Wedge

**Problem Description**

This is a simulation of a symmetric $20^o$ wedge traveling through initially quiescent air at Mach 2.0. A shock forms at the leading edge of the wedge and an expansion fan over its top. Consultation of oblique shock tables, e.g. [8] (pp.308-309) reveals that the angle of the leading shock compares quite well with the expected value. In addition, this simulation demonstrates a few other useful features of the fluid-structure interaction capability. In this case, the structure is rigid, and as such, essentially provides a boundary condition to the compressible flow calculation. Furthermore, the geometry of the wedge is described via a triangulated surface, rather than the geometric primitives usually used. This allows the user to study flow around arbitrarily complex objects, without the difficulty of generating a body fitted mesh around that object.

**Simulation Specifics**

**Component used:** rmpmice (Rigid MPM-ICE)

**Input file name:** Mach2wedge.ups

**Command used to run input file:** sus inputs/UintahRelease/MPMICE/Mach2wedge.ups
(Note: The files wedge40.pts and wedge40.tri must also be copied to the same directory as sus.)

**Simulation Domain:** 0.25 x 0.0375 x 0.001 m

**Cell Spacing:**
.0005 x .0005 x .001 m (Level 0)

**Example Runtimes:**
20 minutes (1 processor, 3.16 GHz Xeon)

**Physical time simulated:** 0.3 milliseconds

**Associated visit session:** M2wedge.session

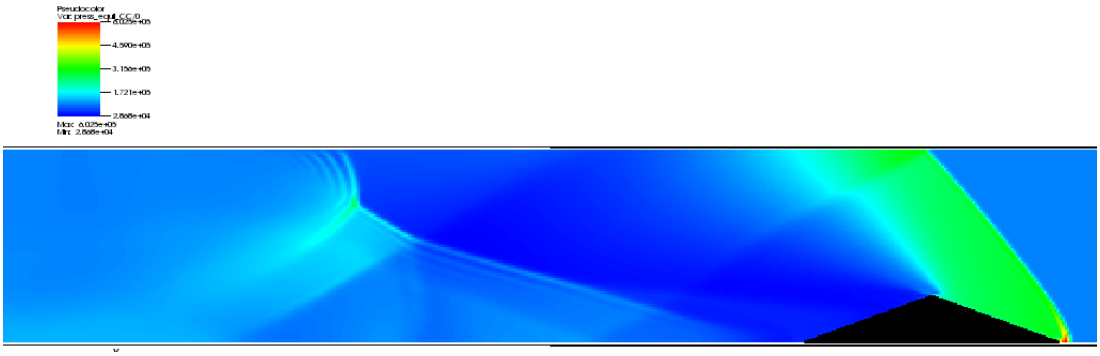


Figure 10.1: $20^o$ wedge moving at Mach 2.0 through initially stationary air. Contour plot depicts pressure.

## Results

Figure 10.1 shows a snapshot of the simulation. Contour plot depicts pressure and reflects the presence of a leading shock and an expansion fan.

## Cylinder in a Crossflow

### Problem Description

In this example the domain is initially filled with air moving at a uniform velocity of $0.03m/s$ A ridgid cylinder $O.D. = 0.02m$ is placed $0.1m$ from the inlet and a passive scalar is injected into the domain through a $0.002m$ hole on in the inlet boundary of the domain. A velocity perturbation is placed upstream of the cylinder to produce an instablity that will help trigger the onset of the Kármán vortex street.

## Simulation Specifics

**Component used:** rmpmice (Rigid MPM-ICE)

**Input file name:** `cylinderCrossFlow.ups`

**Command used to run input file:**

```
mpirun -np 6 sus inputs/UintahRelease/MPMICE/cylinderCrossFlow.ups
```

**Simulation Domain:** 0.3 x 0.15 x 0.001 m

**Cell Spacing:**
.00015 x .001 x .001 m (Level 0)

**Example Runtimes:**
7ish hrs (6 processor, 3.16 GHz Xeon)

**Physical time simulated:** 60 seconds

**Associated visit session:** cyl_crossFlow.session

# Results

Figure 10.2 shows a snapshot of the simulation at time $t = 60sec$. The contour plot of the passive scalar shows the Kármán vortex street behind the cylinder at $Re = 700$. A movie of the results is located at

```
movies/cyl_crossFlow.mpg
```

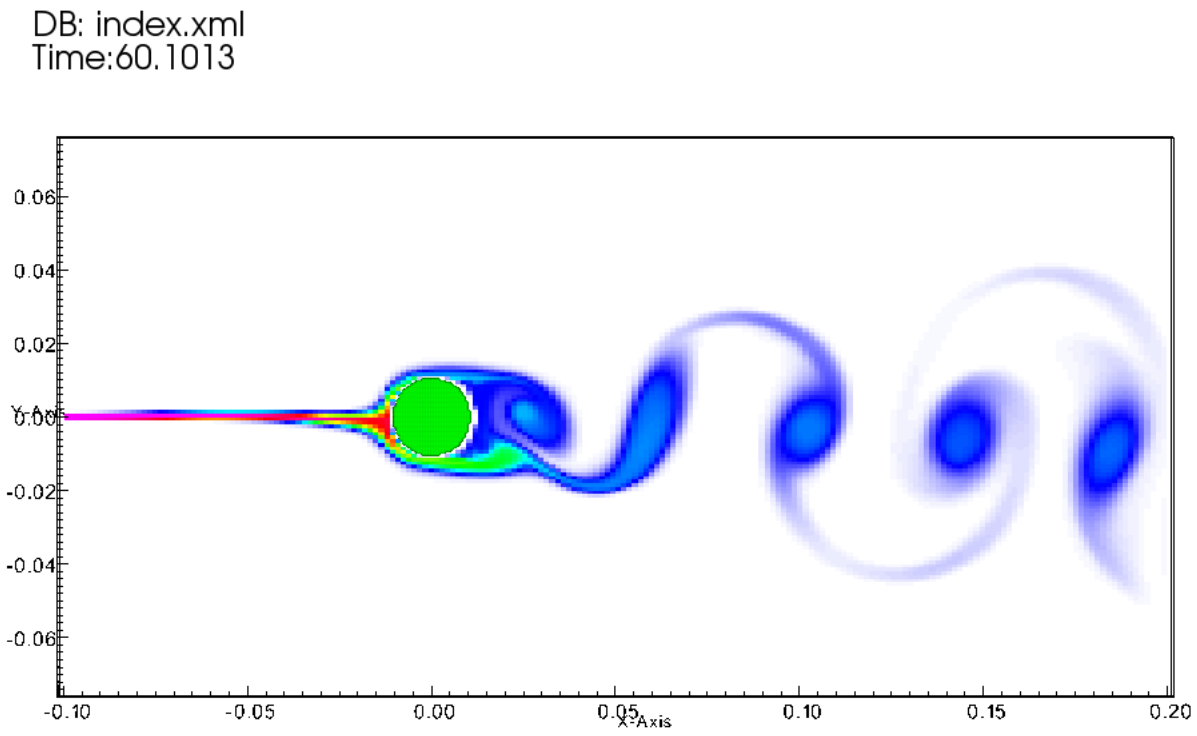



Figure 10.2: Flow over a stationary cylinder, $Re = 700$, a passive scalar is used as a flow marker

# Copper Clad Rate Stick (aka "Cylinder Test")

## Problem Description

This is a two-dimensional version of the "cylinder test" which is used to characterize equations of state for explosive products. In those tests, a copper tube is filled with a high explosive and a detonation is initiated at one end. Various means are used to measure the velocity of the tube as the high pressure product gases expand inside of it.

Here, a cylinder ($r = 2.54cm$) of QM100 is jacketed with a copper cylinder that has a wall thickness of $0.52cm$. Detonation is initiated by giving a thin layer of the explosive a high initial velocity in the axial direction which generates a pressure that is sufficiently high to reach trigger the detonation model. As the detonation proceeds, the copper is pushed out of the domain by the expanding product gases.

Note that in this example, to make run times brief, the domain is very short in the axial direction, and is probably not sufficient for the detonation to reach steady state. Additionally, the domain has been reduced to two dimensions, as symmetry is assumed in the Z-plane. Finally, the spatial resolution of $1.0mm$ is a bit coarse to achieve convergent results. The full three dimensional result can quickly be obtained by commenting out the symmetry condition on the z+ plane and uncommenting the Neumann conditions, as well as changing the spatial extents and resolution in the Z direction to match those in the Y direction.

## Simulation Specifics

**Component used:** mpmice (MPM-ICE)

**Input file name:** `QM100CuRS.ups`

**Command used to run input file:**

```
sus inputs/UintahRelease/MPMICE/QM100CuRS.ups
```

**Simulation Domain:** 0.055 x 0.032 x 0.0005 m

**Cell Spacing:**
1.0 mm x 1.0 mm x 1.0 mm (Level 0)

**Example Runtimes:**
20 minutes (1 processor, 3.16 GHz Xeon)

**Physical time simulated:** 30 $\mu$seconds

**Associated visit session:** QM100.session

## Results

Figure 10.3 shows a snapshot of the simulation at time $t = 60sec$. Particles are colored by velocity magnitude, contours reflect the density of explosive, note the highly compressed region near the shock front.

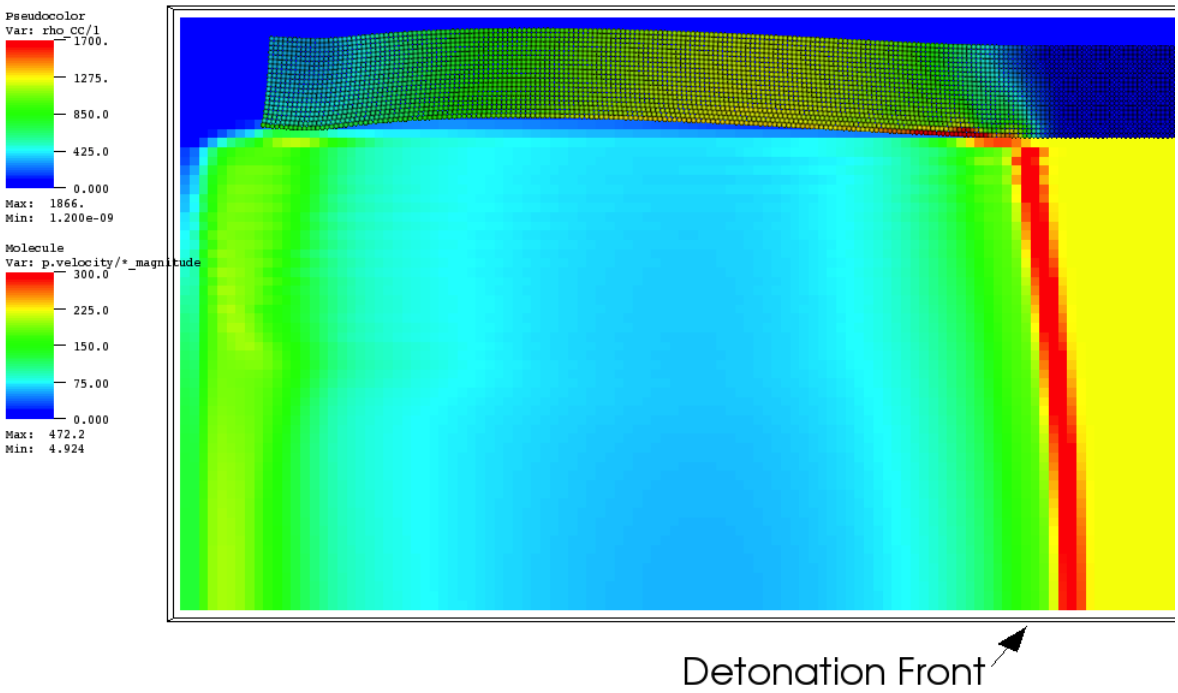



Figure 10.3: Detonation in a copper cylinder (2-D). Particles are colored by velocity magnitude, contours indicate density of unreacted explosive.

# Cylinder Pressurization Using Simple Burn

## Problem Description

This example demonstrates use of the Simple Burn algorithm in an explosive scenario. The exact situation consists of a cylinder of PBX encased in steel. For simplicity it is set up as a 2D simulation. It demonstrates Symmetric boundaries as a useful construct for simplifying the computational requirements of a problem. The end result is the pressurization of a quarter of a cylinder by combustion of PBX 9501. Damage and failure models simulate cylinder failure in a detonation scenario. The simulation as it stands falls far short of the required physical time simulated for actual detonation, but demonstrates how Simple Burn can be used to pressurize a cylinder. For description of Simple Burn see 10.4.1.

## Simulation Specifics

**Component used:** mpmice (MPM-ICE)

**Input file name:** guni2dRT.ups

**Preprocessing on input file:**

1) Comment out or remove <max_Timesteps> on line 21
2) Comment out <outputTimestepInterval> on line 96
3) add <outputInterval>5e-5<outputInterval> on line 97

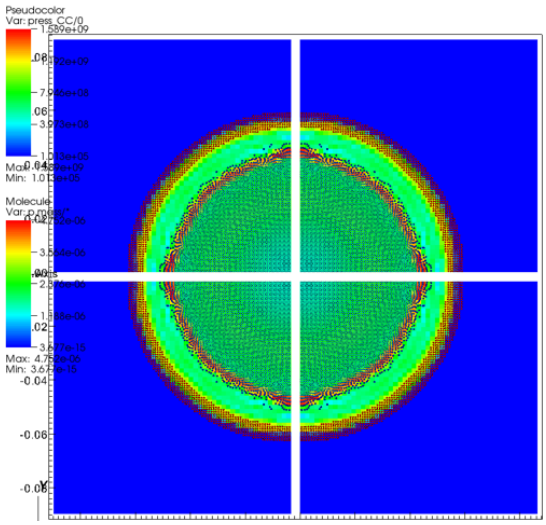
**Command used to run input file:** mpirun -np 4 sus inputs/UintahRelease/MPMICE/guni2dRT.ups

**Simulation Domain:** 8.636 x 8.636 x 0.16933 cm

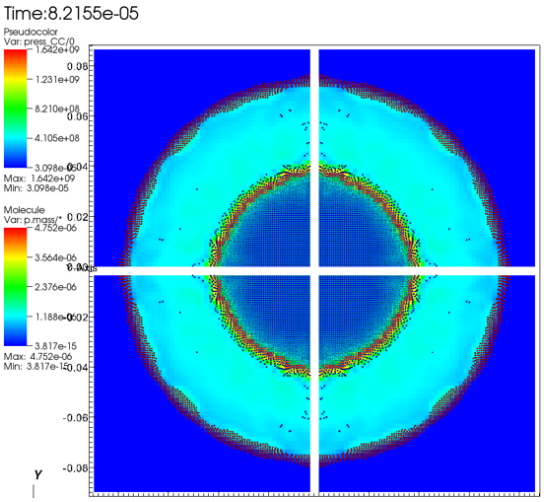
**Example Runtimes:**
2 minutes (1 processor, 2.8 GHz Xeon)

**Physical time simulated:** 8 microseconds

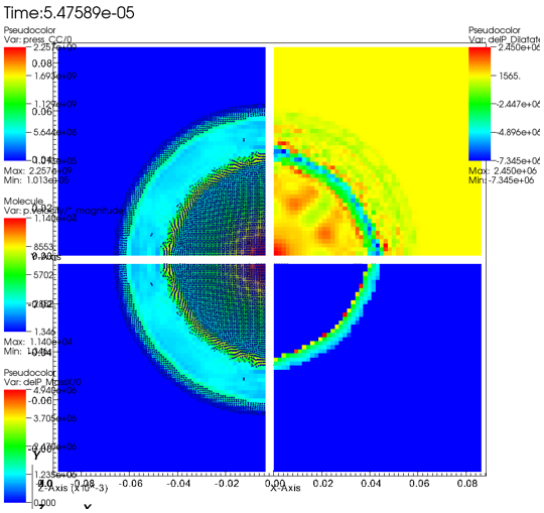
**Associated visit session:** SimpleBurn.session



(a) Receding PBX9501 leads to pressure increase in cylindrical steel shell




(b) Pressure increase causes cylinder to respond




(c) The left half of the image represents particles as spheres colored according to mass and pressure as background color. Top-right shows delP_Dilatate and bottom-right shows delP_MassX

Figure 10.4:

# Results

With the recession of mass comes a pressure increase that causes the case to expand outward. A snapshot of pressure after the 0.4 milliseconds can be seen in Figure 10.4a. At this time pressure has increased to three-fold its initial value. A later snapshot Figure 10.4b shows the response of the steel cylinder to increased pressure. Note that mass flux will scale according to 10.4. Another interesting view of the simulation can be seen in Figure 10.4c. On the left is the normal particle and pseudocolor map representing solid mass and pressure respectively. On the top right, change in pressure during the timestep can be seen (delP_Dilatate). The bottom shows change in pressure due to mass exchange (del_MassX). See table 8.3.10 for description of these variables.

# Exploding Cylinder Using Steady Burn

## Problem Description

This problem consistes of a cylinder initially at 600 K causing burning. Steady Burn acts as the model for burning of HE material. More information on Steady Burn can be found in 10.4.2. The cylinder is build from an outer shell of steel covering a hollow bored cylinder of PBX9501. The simulation demonstrates the violence of explosions when large voids allow rapid expansion of surface area due to collapse of explosive material into the bore. Information on the violence of explosions with solid and hollow cores can be attained in [12].

## Simulation Specifics

**Component used:** mpmice (MPM-ICE)

**Input file name:** SteadyBurn_2dRT.ups

**Preprocessing on input file:**

1) Comment out or remove <max_Timesteps>

2) Comment out <outputTimestepInterval> and uncomment <outputInterval> around line 101

**Command used to run input file:** mpirun -np 4 sus inputs/UintahRelease/MPMICE/SteadyBurn_2dRT.ups

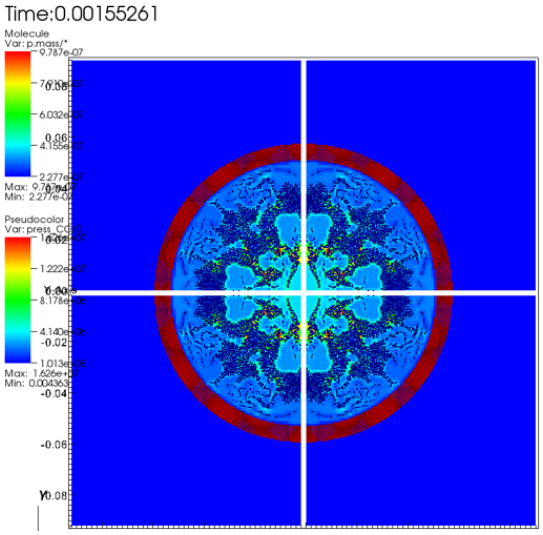
**Simulation Domain:** 9 x 9 x 0.1 cm

**Example Runtimes:**

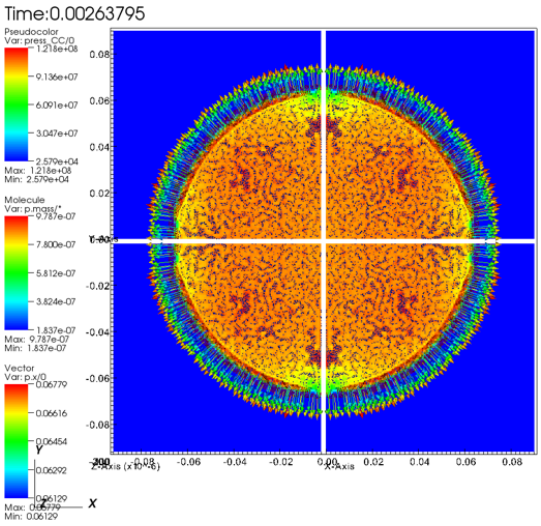
5 hours (1 processor, 2.8 GHz Xeon)

**Physical time simulated:** 3 milliseconds

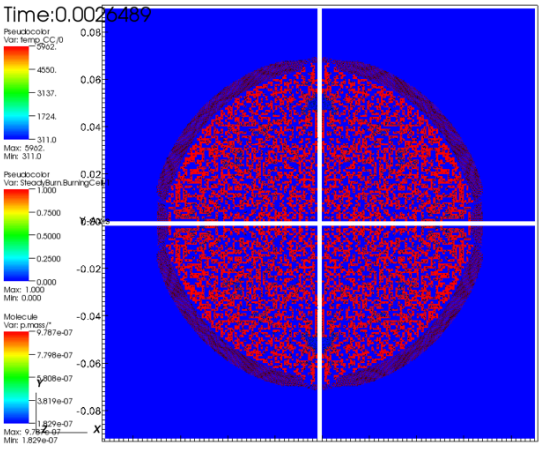
**Associated visit session:** SteadyBurn.session




(a) Collapse of PBX into hollow bore of explosive device




(b) Expansion of steel casing as explosion occurs–response to pressure build-up




(c) Burning Cells denoted by red squares

Figure 10.5:

# Results

Figure 10.5a shows a nice view of the cylinder as the PBX particles within is collapsing into the void, creating more burnable surface area resulting in more violent explosion. Figure 10.5b shows a view of the cylinder as the steel container begins to expand outward. Arrows represent the speed at which the particles in the steel case are expanding outward. Figure 10.5c shows cell flagged as burning by Steady Burn.

# T-Burner Example Using Unsteady Burn

## Problem Description

The T-Burner problem was inspired by an article by Jerry Finlinson, Richard Stalnaker and Fred Blomshield in which a T-Burner apparatus was pressurized to a given pressure and ignited [2]. The T-Burner composed of a cylinder with HMX on each circular ends, and a pressure inlet halfway between the HMX caps pumps pressure into the vessel parallel to those walls. Finlinson, et. al. measured pressure oscillations in the chamber and this simulation mimics the behavior found of Finlinson's 500 psi experiment. For simplicity and resource minimization, the simulation is set up as a 2D T-Burner. The graphs below shows the pressure oscillations over time compared with that from [2]. This simulation demonstrates the utility of Unsteady Burn in simulations where pressure oscillations occur in small places. For more information on Unsteady Burn see 10.4.3.

## Simulation Specifics

**Component used:** mpmice (MPM-ICE)

**Input file name:** `TBurner_2dRT.ups`

**Command used to run input file:** mpirun -np 4 sus TBurner_2dRT.ups

**Simulation Domain:** 0.822 x 0.138 x 0.003 m

**Example Runtimes:**
25 minutes (1 processor, 2.8 GHz Xeon)

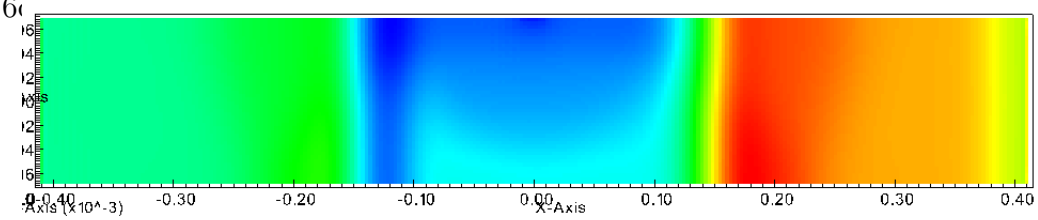
**Physical time simulated:** 0.46 milliseconds
0.46 milliseconds of simulation equates flag <max_Timesteps>410< /max_Timesteps>

Notes:
1)Remove line from input file to allow simulation to run full 0.25 seconds
2)Comment out <outputTimestepInterval> and uncomment <outputInterval> to make output $\Delta t$ constant

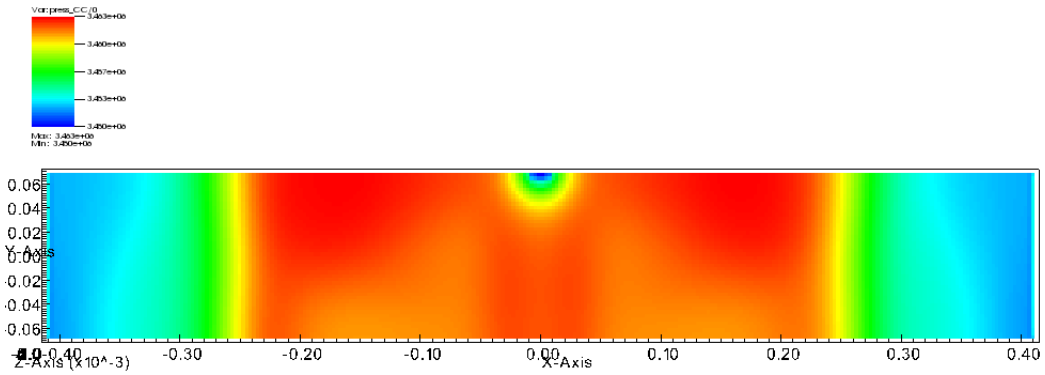
**Associated visit session:** TBurner.session

# Results

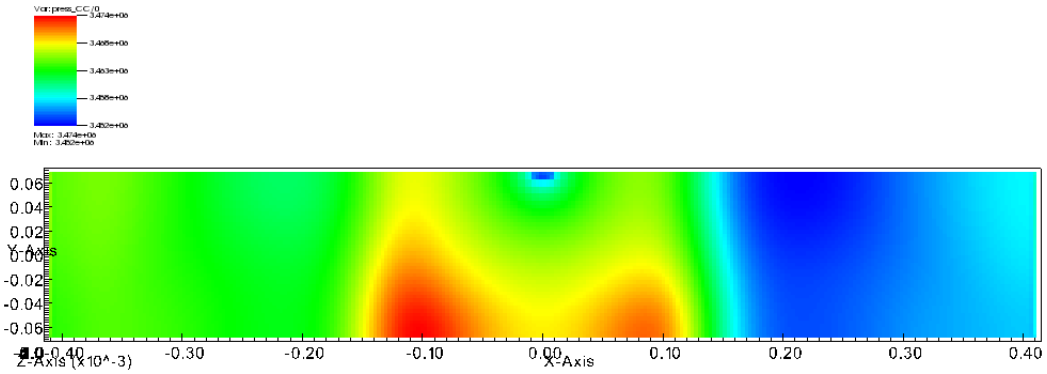
Figure 10.6a plot depicts
pressure anc PBX 9501.
Figure 10.6



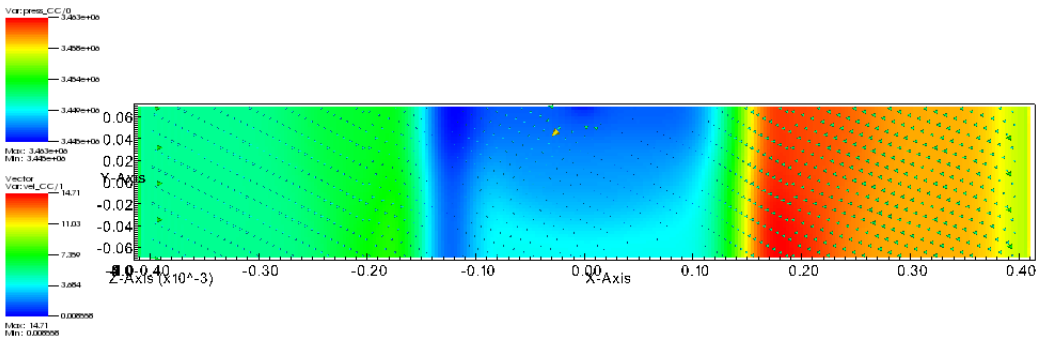

(a) Time 1: Oscillatory behavior in the form of a pressure wave in a T-Burner. Contour plot depicts pressure




(b) Time 2: Oscillatory behavior in the form of a pressure wave in a T-Burner. Contour plot depicts pressure




(c) Time 3: Oscillatory behavior in the form of a pressure wave in a T-Burner. Contour plot depicts pressure




(d) Velocity vectors of cell material. Shows how the pressure causes gas to move

Figure 10.6:

Figure 10.6d shows a snapshot of the simulation at the same instant as the previous figure. The contour plot depicts pressure. The arrows are vectors depicting the importance

# Lizard Lung

## Problem Description

This simulation provides an example of fluid-solid interaction using both a compliant MPM material, as well as a rigid material that is given a prescribed time-velocity profile. The latter acts as a piston that pushes or pulls air though a pair of valves, one of which opens and the other of which closes, depending on the direction of the flow. In addition to the input file, a second file, "breathe.txt", describes the velocity of the piston.

A post-processing script has been added, massFlowRateLL.m. This allows one to compute the mass flow rate through each of the valves. The script runs through matlab, and requires minor editing to modify the uda name, and whether output is wanted regarding the top or bottom valve.

## Simulation Specifics

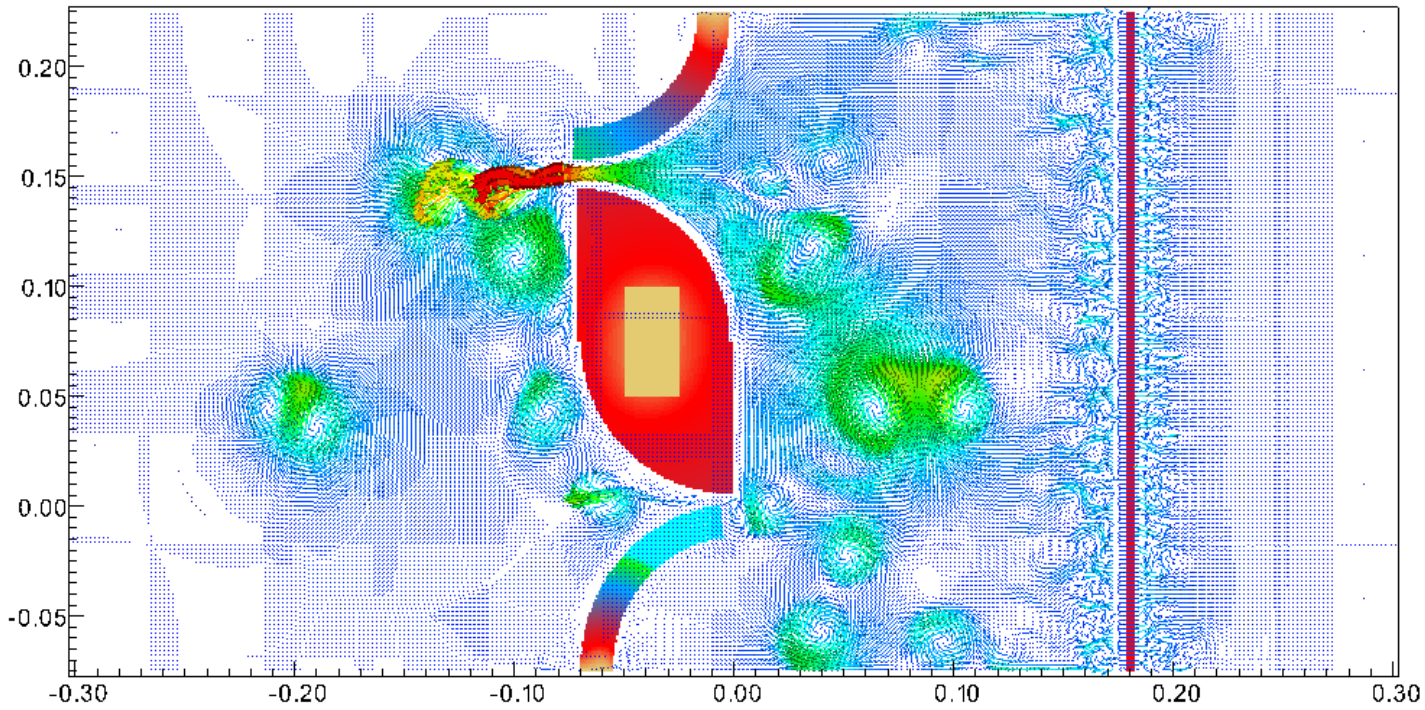
**Component used:** mpmice (MPM-ICE)

**Code modification required:** Because symmetry boundary conditions are used on the z- and z+ faces, without modification, the center section of the solid geometry, shown in Figure 10.7a, would rotate into a more favorable aerodynamic position, as there is nothing stopping it, other than its inertia. Therefore, SerialMPM.cc was modified to keep a block within that section fixed in space. The particles to hold fixed are identified by their "color":

```
// Update the particle's position and velocity
if(pColor[idx] < 1.0){
  pxnew[idx]           = px[idx]    + vel*delT*move_particles;
  pdispnew[idx]        = pdisp[idx] + vel*delT;
  pvelocitynew[idx]    = pvelocity[idx]    + acc*delT;
  // pxx is only useful if we're not in normal grid resetting mode.
  pxx[idx]             = px[idx]    + pdispnew[idx];
  pTempNew[idx]        = pTemperature[idx] + tempRate*delT;
  pTempPreNew[idx]     = pTemperature[idx]; // for thermal stress
} else {
  pxnew[idx]           = px[idx];
  pdispnew[idx]        = pdisp[idx];
  pvelocitynew[idx]    = pvelocity[idx];
  // pxx is only useful if we're not in normal grid resetting mode.
  pxx[idx]             = px[idx];
  pTempNew[idx]        = pTemperature[idx];
  pTempPreNew[idx]     = pTemperature[idx]; // for thermal stress
}
```

This modification occurs in interpolateToParticlesAndUpdate, around line 3450 in this file.

**Input file name:** `lizardLung4.ups`




(a) Air pushed and pulled by a piston through two check valves.

Figure 10.7:

**Instruction to run input file:**
Copy breathe.txt to the same directory as sus.

**Command used to run input file:** mpirun -np 16 sus
inputs/UintahRelease/MPMICE/lizardLung4.ups

**Simulation Domain:** 0.6 x 0.3 x 0.002 m

**Example Runtimes:**
18 hours (16 processors, 2.40 GHz Xeon) (Much less time is needed to see the behavior, 18 hours is enough for many piston oscillations.)

**Physical time simulated:** 10 seconds

# Results

Figure 10.7a shows a visualization of the geometry having been deformed by the movement of air caused by the motion of the piston at about x=0.18. The piston moves sinusoidally according to the velocity table prescribed in "breathe.txt".

# Bibliography


[1] H.L. Berghout, S.F. Son, C.B. skidmore, D.J. Idar, and B.W. Asay. Combustion of damaged pbx 9501 explosive. *Thermochimica Acta*, pages 261–277, 2002.

[2] J.C. Finlinson, R. Stalnaker, and Blomshield F.S. Hmx and rdx t-burner pressure coupled response at 200, 500, and 1000 psi. *Proceedings of 36th JANNAF Combustion Meeting*, 1999.

[3] J.E. Guilkey, T.B. Harman, and B. Banerjee. An eulerian-lagrangian approach for simulating explosions of energetic devices. *Computers and Structures*, 85:660–674, 2007.

[4] A. Khawam and D.R. Flanagan. Basics and application of solid-stat kinetics: A pharmaceutical perspective. *Journal of Pharmaceutical Science*, 95:472–498, 2006.

[5] E.L. Lee and C.M. Tarver. Phenomenological model of shock initiation in heterogeneouse explosives. *Physics of Fluids*, pages 2362–2372, 1980.

[6] P. C. Souers S. Anderson J. Mercer E. McGuire and P. Vitello. Jwl++: A simple reactive flow code package for detonation. *Propellants, Explosives, Pyrotechnics*, 25:54–58, 2000.

[7] J.R. Peterson and C.A. Wight. An eulerian-lagrangian computational model for deflagration and detonation of high explosives. *Combustion and Flame*, pages 2491–2499, 2012.

[8] M. A. Saad. *Compressible Fluid Flow*. Prentice-Hall, New Jersey, 1985.

[9] V. Sergey and C.A. Wight. Isothermal and nonisothermal reaction kinetics in solids: In search of ways toward consensus. *Journal of Physical Chemistry A*, 101:8279–8284, 1997.

[10] S. F. Son, B. W. Asay, E. M. Whitney, and H. L. Berghout. Flame spread across surfaces of PBX 9501. In Elsevier, editor, *Combustion Institute*, volume 31, pages 2063–2070, 2007.

[11] M.J. Ward, S.F. Son, and M.Q. Brewster. Steady deflagration of hmx with simple kinetics: A gas phase chain reaction model. *Combustion and Flame*, 114:556–568, 1998.

[12] C.A. Wight and E.G. Eddings. Science-based simulation tools for hazard assessment and mitigation. *Advancements in Energetic Materials and Chemical Propulsion*, pages 921–937, 2008.

# Chapter 11

# Wasatch

The Wasatch documentation is available in a separate document located at "doc/Components/Wasatch".

# Chapter 12

# Glossary

- Data Warehouse (NewDW, OldDW, DW) - The `Data Warehouse` is an abstraction (and implementation vehicle) used in Uintah to provide data to simulation components (across distributed memory spaces as necessary). OldDW refers to a DW from the previous time step. NewDW refers to the DW for the current time step. In practice, variables are usually pulled from the OldDW, updated, and placed in the NewDW.
- Time step - Uintah is a time dependent code. A time step refers to a unique point in simulation time. The state of the simulation is updated one time step at a time.
- Adaptive Mesh Refinement (AMR) - In brief, AMR allows spending less CPU time on "inactive" (less interesting) areas of the simulation, and spend more time computing where there are many particles reacting. Resolution is low in the center where things are stable, but high at the edges. This feature is in ICE, but not ARCHES.
- CCA - Common Component Architecture.
- CFD - Computational Fluid Dynamics modeling.
- DistCC - Parallel, distributed compiler.
- Doxygen - Doxygen (code documentation) web interface.
- GhostCells (and Extra Cells)
- Grid - The problem's physical domain. The number of cells in the grid determine the resolution of the simulation.
- Handle - Smart pointers. Handles track the number of references to a given object, and when the number reaches zero, de-allocates the memory.
- Level - Not a 'level' in 3d-space, but a level of recursion into an AMR grid. ARCHES doesn't support AMR or nonuniform cells, and therefore doesn't need recursion, so it works on a single level '1'.
- Material Point Method (MPM) - The main component for simulating structures (physical objects) in the UCF.

- Message Passing Interface (MPI) - Communication library used by many distributed software packages to communicate data between multiple processors. Besides send'ing and recv'ing data, data reduction (UCF Reduction Variables) is supported.
  - OpenMPI
- Patch - A physical region of the grid assigned one to each processor. The processor working on a patch will compute properties for each of the cells contained in the patch. Think of this as a big cube that contains hundreds of little cubes.
- Regression Tester (RT) - Runs nightly accuracy, memory, and completion tests on Uintah simulations.
- SCIRun - A Problem Solving Environment (PSE) originally used to provide core software building blocks for Uintah as well as an extensive visualization package for viewing Uintah data archives.
- SUS - Standalone Uintah Simulator. This is the main executable program in the Uintah project.
- SVN - Subversion code versioning system.
- Uintah - The general name of the C-SAFE simulation code. Sometimes also referred to as the UCF. The name comes from the Uintah mountain range in Utah.
- Uintah Computational Framework (UCF) - The core software infrastructure for Uintah.
  - Variables (CC, NC, FC) - Cell centered, Node centered, and Face centered (respectively) data structures used within the UCF.
- Uintah Data Archive (UDA) - The directory/file/data layout for storing Uintah simulation data.
- Uintah Problem Specification (UPS (Section 2.3)) - An XML based file used to specify Uintah simulation properties.
- Uintah Software Organization
  - Visualization
  - scinew - a wrapper for the C++ new() function that allows for memory tracking.

# Appendix A

# Bomb Units

Following is a table of conversion factors from bomb units to mks units. Bomb units are useful in small scale simulations that occur very quickly, such as detonation and deformation.

| **Measure** | **Conversion Factor** |
|---|---|
| mass | $\mu g$ |
| length | cm |
| time | $\mu s$ |
| kinematic viscosity | $1\frac{cm^2}{\mu s} = 10^2\frac{m^2}{s}$ |
| velocity | $1\frac{cm}{\mu s} = 10^4\frac{m}{s}$ |
| force | $1\frac{\mu g cm}{\mu s^2} = 10N$ |
| pressure | $\frac{10N}{cm^2} = 10^5 Pa$ |
| viscosity | $10^5 Pa\mu s = 10^{-1} Pas$ |
| density | $1\frac{\mu g}{cm^3} = 1\frac{g}{m^3}$ |
| heat capacity | $\frac{10Ncm}{\mu gK} = 10^8\frac{J}{kgK}$ |
| power | $\frac{10Ncm}{\mu s} = 10^5 W$ |
| thermal conductivity | $\frac{10^5 W}{cmK} = 10^7\frac{W}{mK}$ |
| surface energy | $\frac{10Ncm}{cm^2} = 10^3\frac{J}{m^2}$ |
| fracture toughness | $\frac{10N}{cm^{\frac{3}{2}}} = 10^{-2}\frac{N}{m^{\frac{3}{2}}}$ |
| enthalpy | $1\frac{J}{kg} = 10^{-8}\frac{cm^2}{\mu s^2}$ |