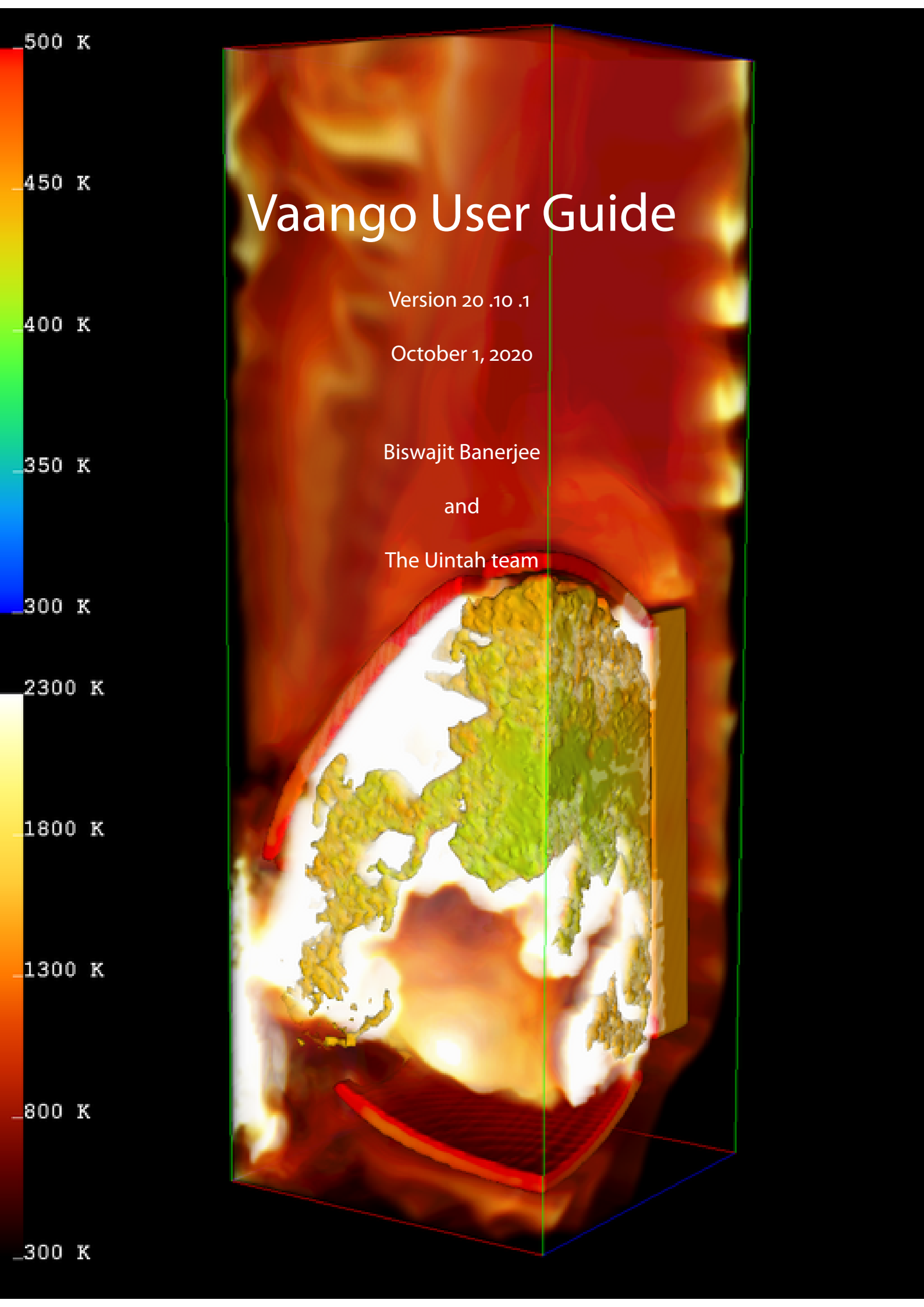


Vaango User Guide

Version 20 .10 .1

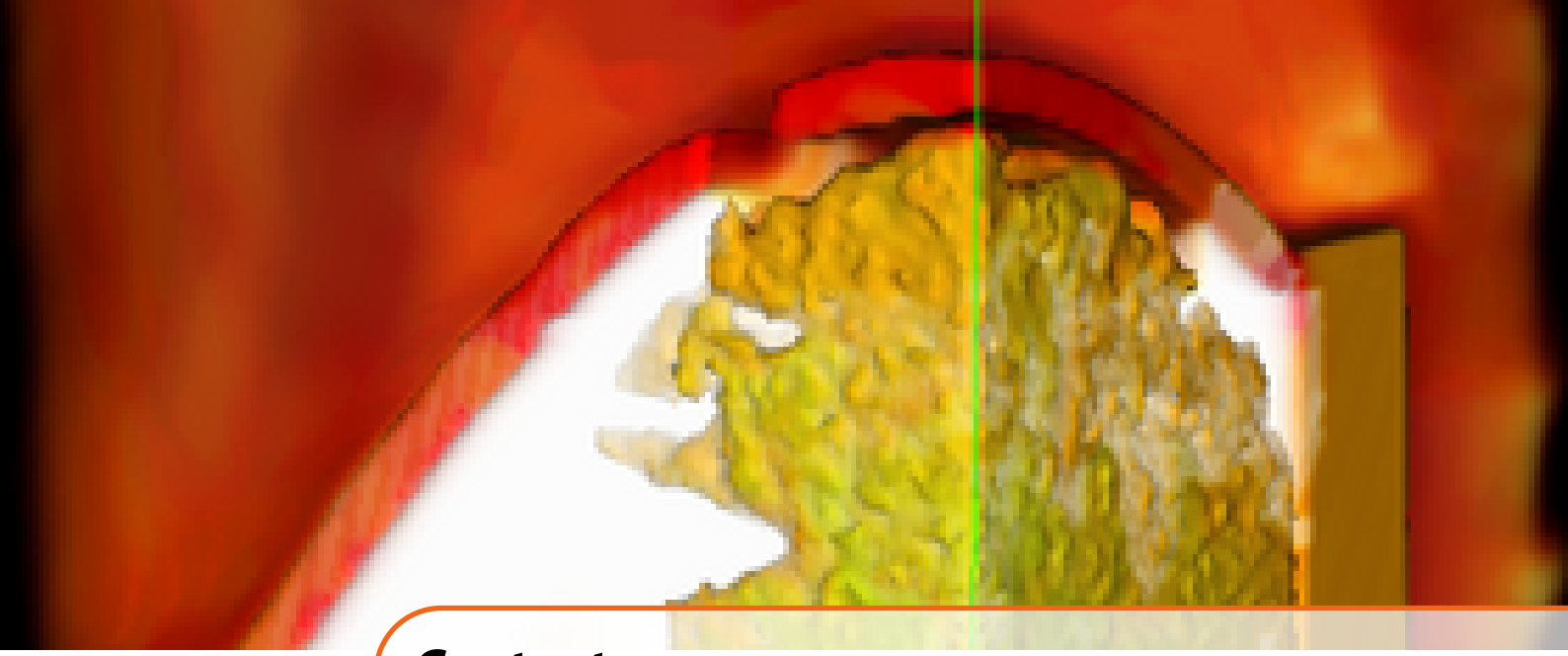
October 1, 2020

Biswajit Banerjee

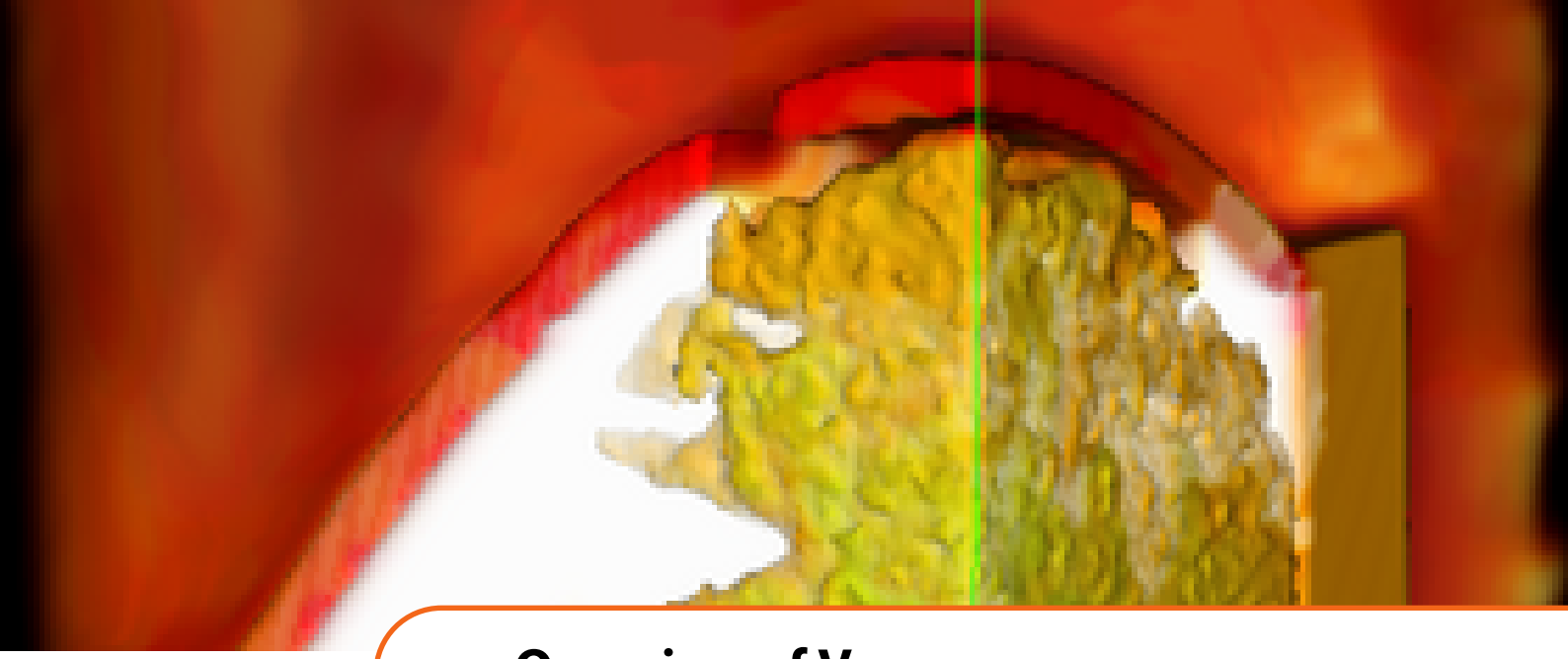
and

The Uintah team

The contents of this manual can and will change significantly over time. Please make sure that all the information is up to date.



Contents



1 — Overview of Vaango

1.1 History

Vaango has been developed using the Uintah computational Framework originally developed at the University of Utah Center for the Simulation of Accidental Fires and Explosions (C-SAFE). The code was forked in 2012 for a visual effects project in Callaghan Innovation, New Zealand, and subsequently open-sourced in 2015 after being acquired by Parresia Research Limited, New Zealand.

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

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 was later renamed to the Advanced Simulation and Computing (ASC) program.)

Center Objective

C-SAFE's primary objective was 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. The specific target simulation was the heating of an explosive device placed in a large hydrocarbon pool fire and the subsequent deflagration explosion and blast wave (Figure 1.1). In that simulation, the explosive device is a small cylindrical steel container (4 in. 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 could 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.

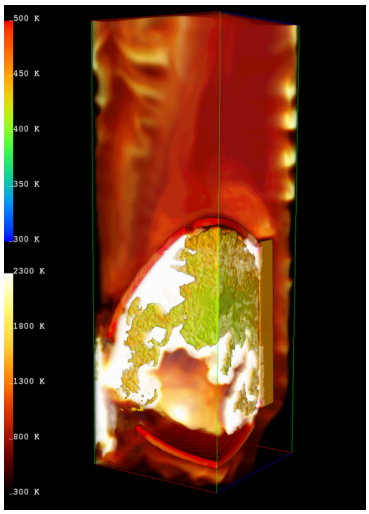


Figure 1.1: Target Simulation - Fire-Container-Explosion.

1.1.2 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. The source code and latest releases can be found at http://uintah.utah.edu.

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 Vaango Developers Guide for more information about the internal architecture of Uintah.

Software Ports

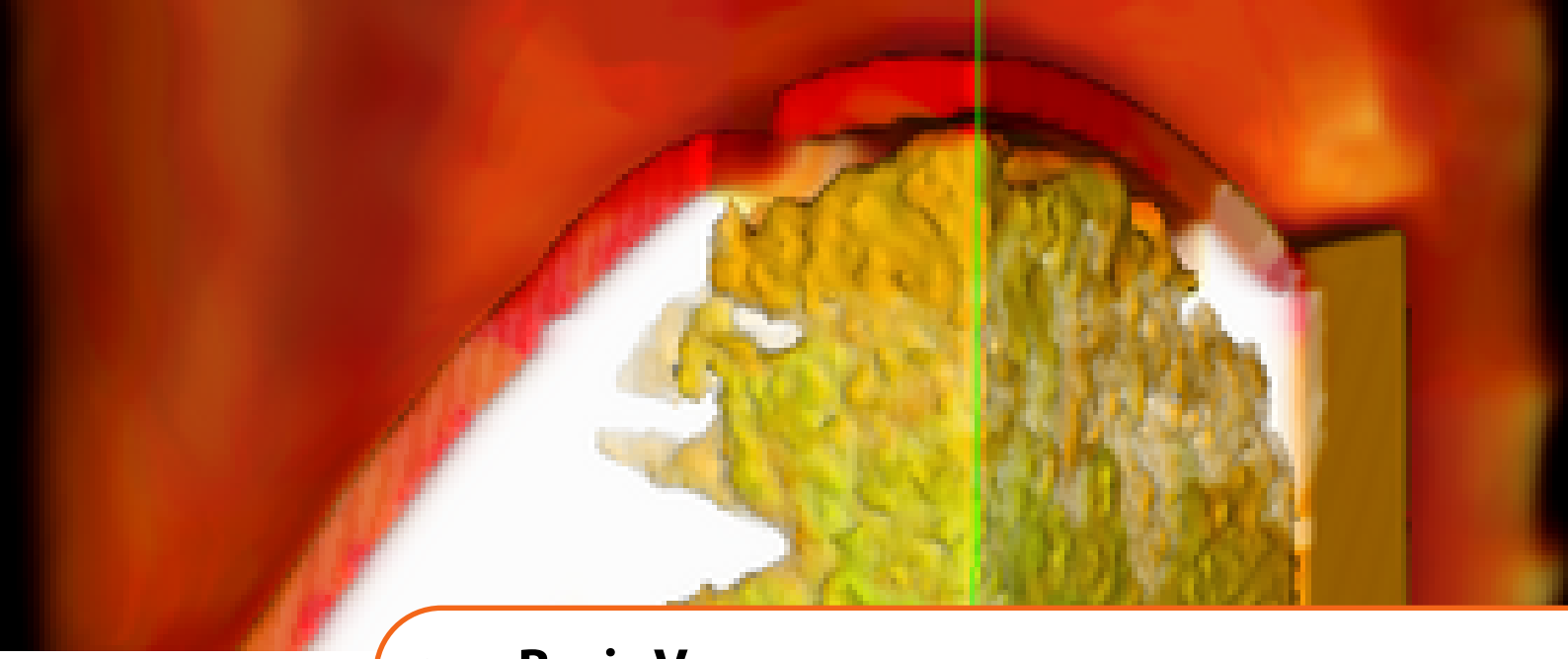
Uintah has been ported and runs well on a number of operating systems. These include Linux, Mac OSX, Windows, AIX, and HPuX. Simulating small problems is perfectly feasible on 2-4 processor desktops, while larger problems will need 100s to 1000s of processors on large computer clusters.

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.

1.2 Vaango software

The Vaango suite of tools is built on top of the Uintah computational framework with an emphasis on solid mechanics (mechanical and civil engineering) rather than chemistry and fluid dynamics. With that in mind, the fire simulation components have been removed while additional solid mechanics models and tools have been added relative to Uintah .



2 — Basic Vaango usage

Several executable programs have been developed using the Vaango Computational Framework (UCF). The primary code that drives the components implemented in Vaango is called vaango .

Although Uintah was developed for complex fire-explosion simulations, the general nature of the algorithms and the framework has allowed researchers to use the code to investigate a wide range of problems. The Vaango framework is general purpose enough to allow for the implementation of a variety of implicit and explicit algorithms on structured grids but focuses on particle based algorithms.

2.1 Installing the Vaango software

For information on downloading the Vaango software package and how to setup and build it, please refer to the Vaango Installation Guide.

2.2 Running Vaango

For single processor simulations, the vaango executable is run from the command line prompt like this:

```
vaango input.ups
```

where input.ups is an XML formatted input file. The Vaango software contains numerous example input files located in the src/StandAlone/inputs 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 vaango 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 flag is optional but sometimes necessary if the mpi environment is not automatically detected from within the vaango executable.

Vaango 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 Vaango Problem Specification

The VAANGO framework uses XML like input files to specify the various parameters required by simulation components. These are called Uintah Problem Specification files and have the extension .ups because they are directly based on the UINTAH input file format. The .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 VAANGO 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, and MPMICE the individual tags will be explained more fully.

The vaango 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, peridynamics as in:

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

2.5 Time Related Variables

VAANGO 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.

2.6.1 Saving data

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

Data can be saved based on a frequency setting that is either based on integer time intervals:

```
<outputTimestepInterval> 100 </outputTimestepInterval>
```

or real-valued timestep intervals:

```
<outputInterval> 1.0e-3 </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.

2.6.2 Reduction variables

While most variable data remains confined to a single patch and is transferred to adjacent patches only if needed, there are certain variables that need to be communicated to all the patches. Such variables are typically reduced before communication and are called Reduction variables.

MPM reduction variables that can be saved in the data archive include the following possibilities:

```
<save label = "TotalMass" />
<save label = "TotalMomentum" />
<save label = "ThermalEnergy" />
<save label = "KineticEnergy" />
<save label = "StrainEnergy" />
<save label = "AccStrainEnergy" />
<save label = "TotalVolumeDeformed" />
<save label = "CenterOfMassPosition" />
```

The accumulated strain energy, AccStrainEnergy , is useful for incremental material models where only an incremental strain energy computed in each step.

2.6.3 Check-pointing for restarts

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 checkpoint data that is created every .01 seconds of simulation time are shown below:

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

An alternative checkpointing scheme that saves data every 25 timesteps can be specified using:

```
<checkpoint cycle = "2" timestepInterval = "25"/>
```

To restart from a checkpointed archive, simply put -restart in the vaango command-line arguments and specify the .uda directory instead of a ups file (vaango 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, vaango copies all of the appropriate information from the old uda directory to its new uda directory.

Here are some examples:

```
./vaango -restart disks.uda.000 -nocopy
./vaango -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 the VAANGO Developers Manual.

2.8 Geometry creation

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.

See Chapter 3 for further detail.

2.8.1 Resolution

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.

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 0.5. 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">
```

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.

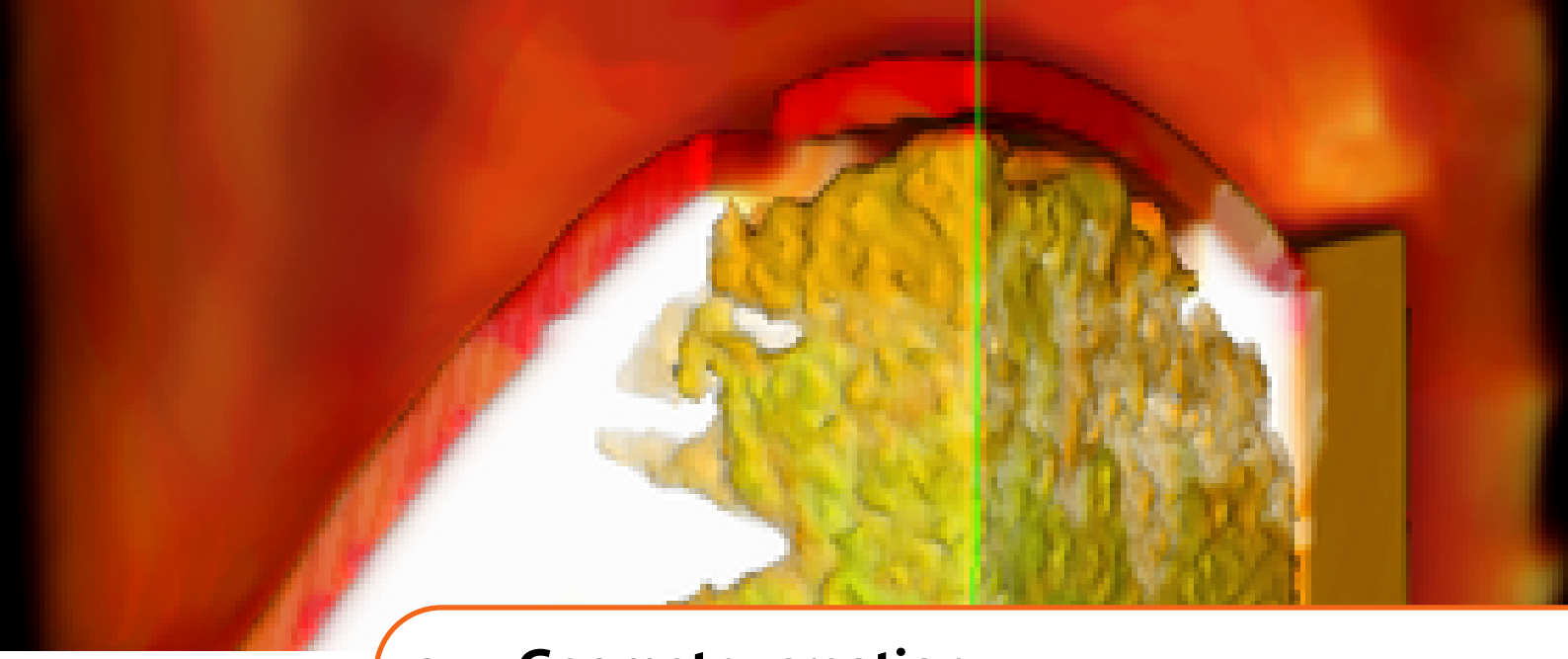
The grid is decomposed into a number of patches. For single processor problems, usually one patch is used for the entire domain. For multiple processor simulations, there must be at least one patch per processor. Patches are specified along the x,y,z directions of the grid using the <patches>[2,5,3] </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 vaango could use is $2 \times 5 \times 3 = 30$. 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 .



3 — Geometry creation

The creation of geometry objects within the computational domain is a crucial part of the simulation process. This chapter discusses the options available for creating geometry in Vaango .

> Example input files for creating the geometries discussed in this chapter can be found in inputs/MPM/GeometryPieceExamples .

3.1 Basic geometry objects

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:

3.1.1 Box

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

```
<box label="box_1">
  <min>[0.0, 0.0, 0.0]</min>
  <max>[1.0, 2.0, 3.0]</max>
</box>
```

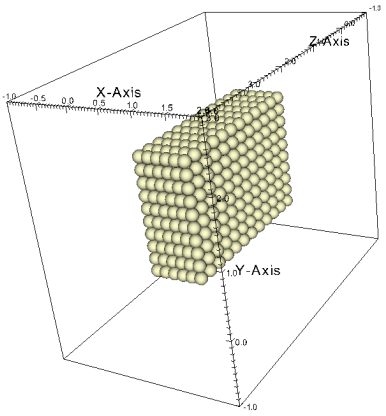
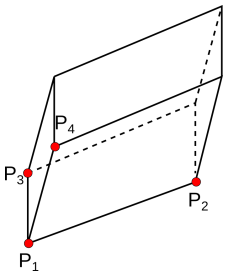


Figure 3.1: A box geometry piece.

3.1.2 Parallelepiped

parallelepiped requires the locations of four points that define the bounds of the parallelepiped. The point p1 is a corner while the the vectors p2 - p1 , p3 - p1 , and p4 - p1 represent the vectors along the three edges of the parallelepiped starting from p1 .

```
<parallelepiped label="box_1">
  <p1>[ 0.0, 0.0, 0.0]</p1>
  <p2>[ 1.0, 0.0, 0.0]</p2>
  <p3>[ 1.0, 2.0, 0.5]</p3>
  <p4>[-1.0, 1.0, 2.0]</p4>
</parallelepiped>
```



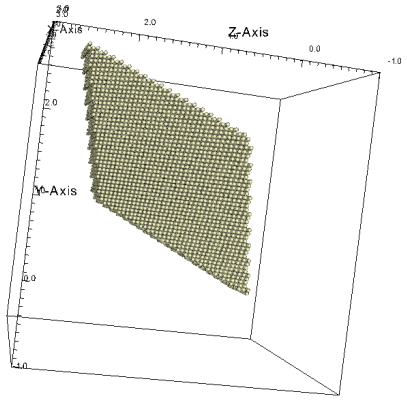


Figure 3.2: A parallelepiped geometry piece.

3.1.3 Sphere

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

```
<sphere label="ball_1">
  <origin>[ 0.0, 0.0, 0.0]</origin>
  <radius> 0.75 </radius>
</sphere>
```

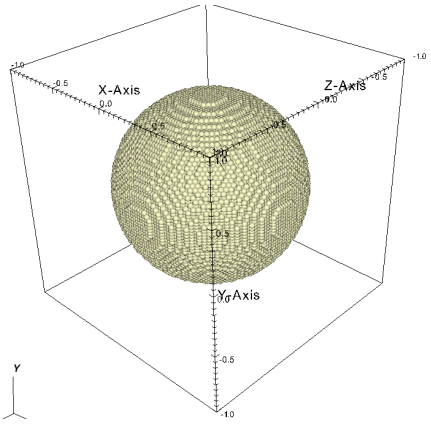


Figure 3.3: A sphere geometry piece.

3.1.4 Cylinder

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

```
<cylinder label="cyl_1">
  <bottom>[ -0.5, -0.5, -0.5]</bottom>
  <top>[ 1.5, 1.5, 1.5]</top>
  <radius> 0.5 </radius>
</cylinder>
```

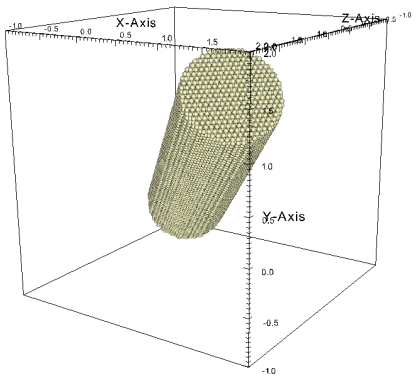


Figure 3.4: A cylinder geometry piece.

3.1.5 Cone

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.

```
<cone label="cone_1">
  <bottom>[ -0.5, -0.5, -0.5]</bottom>
  <top>[ 1.5, 1.5, 1.5]</top>
  <bottom_radius> 0.7 </bottom_radius>
  <top_radius> 0.1 </top_radius>
</cone>
```

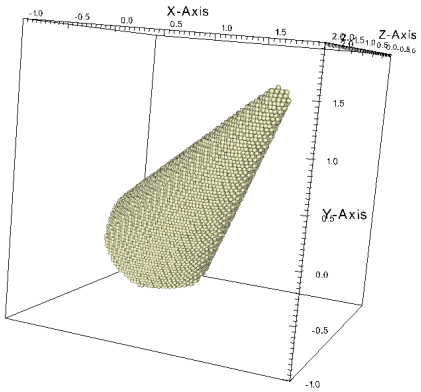


Figure 3.5: A cone geometry piece.

3.1.6 Ellipsoid

ellipsoid has an origin tag specified as a vector. An ellipsoid is defined by two orthogonal axis vectors v1 and v2 and three semi-axis lengths r1, r2, r3. The axis vectors must be orthogonal to within 1e-12 after dot product or the simulation will throw an exception.

```
<ellipsoid label="ellipsoid_1">
  <origin>[ 0.0, 0.0, 0.0]</origin>
  <v1> [0.353553, 0.353553, 0.4] </v1>
  <v2> [0.087038295569, -0.783349583783,
      0.615457362205] </v2>
  <r1> 0.6403119924052649 </r1>
  <r2> 1.0 </r2>
  <r3> 2.0 </r3>
</ellipsoid>
```

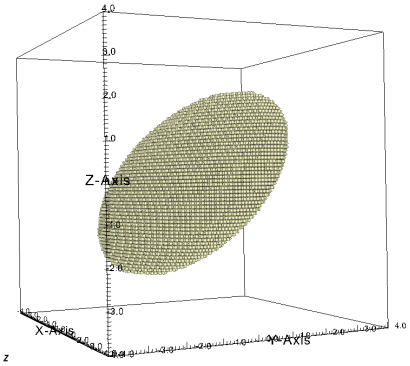


Figure 3.6: An ellipsoid geometry piece.

3.1.7 Torus

torus has a center , an axis vector, and major and minor radii. The major radius must be greater than the minor radius. The axis vector is converted internally into a unit vector and cannot have zero length.

```
<torus label="torus object">
  <center> [0.1, 0.1, 0.1] </center>
  <axis_vector>[ 1.5, 1.5, 1.5] </axis_vector>
  <major_radius> 1.0 </major_radius>
  <minor_radius> 0.5 </minor_radius>
</torus>
```

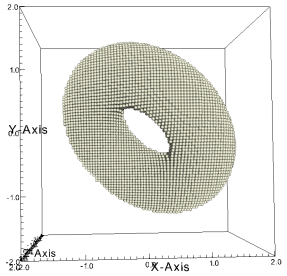


Figure 3.7: A torus geometry piece.

3.1.8 Triangulated surface

tri is a tag for describing a triangulated surface. The file_name_prefix tag specifies the file name to use for reading in the triangulated surface description and the points file.

```
<tri label = "goblet">
  <file_name_prefix> tri_goblet </file_name_prefix>
</tri>
```

The default behavior is to expect a triangulated surface file (file_name.tri) that contains a list of integers describing the connectivity of points specified in file_name.pts . Here is an excerpt from the tri_goblet.tri file:

```
0 49 6
0 6 18
18 6 8
18 8 22
....
```

The tri_goblet.pts file contains data of the form:

```
-2.6575 -0.366357 2.656
-2.69162 -0.57185 2.656
-2.61027 -0.577179 2.53656
-2.69395 -0.641317 2.656
-2.5085 -0.406282 2.39336
....
```

The tri tag can also be used to read triangulated surfaces in OBJ , PLY , and STL formats. The file_type tag specifies the type of file to be read. For a PLY file, the file has to have the .stl extension and we can use the following:

```
<tri label = "horse">
  <file_name_prefix> horse </file_name_prefix>
  <file_type> ply </file_type>
</tri>
```

For an OBJ file, with extension .obj , we need

```
<tri label = "teddy bear">
  <file_name_prefix> teddy </file_name_prefix>
  <file_type> obj </file_type>
</tri>
```

For a STL file, with extension .stl , the UPS file requires

```
<tri label = "panther">
  <file_name_prefix> panther </file_name_prefix>
  <file_type> stl </file_type>
</tri>
```

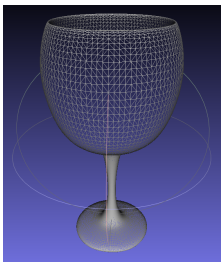


Figure 3.8: The triangle mesh.

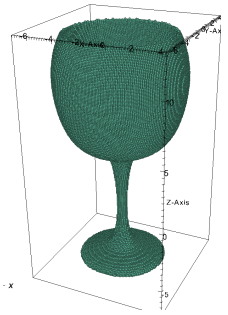


Figure 3.9: A tri geometry piece.

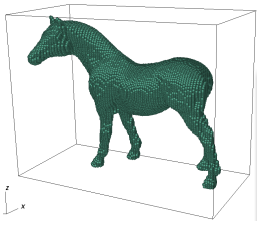


Figure 3.10: Particles created from a PLY mesh.

If you are unsure of the size of the domain that will fit your geometry, examine the output to get information on the bounding box of the object that has been read in.

tri geometry pieces can scaled , reflected , and translated . The order of the co-ordinates can also be rearranged to suit a given simulation.
To scale an object we use the scaling_factor tag:

```
<tri label = "panther">
  <file_name_prefix> panther </file_name_prefix>
  <file_type> stl </file_type>
  <scaling_factor> 0.5 </scaling_factor>
</tri>
```

To translate an object we use the translation_vector tag:

```
<tri label = "panther">
  <file_name_prefix> panther </file_name_prefix>
  <file_type> stl </file_type>
  <scaling_factor> 0.5 </scaling_factor>
  <translation_vector> [0.0, -15.0, 0.0] </
      translation_vector>
</tri>
```

To reflect an object about an axis we use the reflection_vector tag. Reflections are currently allowed only with reflect to the global coordinates and require a value of -1 to reflect and 1 to retain the current orientation.

```
<tri label = "panther">
  <file_name_prefix> panther </file_name_prefix>
  <file_type> stl </file_type>
  <scaling_factor> 0.5 </scaling_factor>
  <reflection_vector> [1.0, 1.0, -1.0] </
      reflection_vector>
</tri>
```

The order of the axes can also be change during the geometry set-up phase to align the object along a particular orientation. The axis_sequence tag is used for that purpose. Allowed values are $[1, 2, 3]$, $[2, 3, 1]$, $[3, 1, 2]$, $[1, 3, 2]$, $[3, 2, 1]$, and $[2, 1, 3]$.

```
<tri label = "panther">
  <file_name_prefix> panther </file_name_prefix>
  <file_type> stl </file_type>
  <scaling_factor> 0.5 </scaling_factor>
  <axis_sequence> [2,1,3] </axis_sequence>
</tri>
```

The order in which the scaling, translation, and reflection operations are carried out matters. Translation followed by scaling will lead to different results than scaling followed by translation. The same holds for reflection. Internally, objects are scaled first, reflected, and then translated.

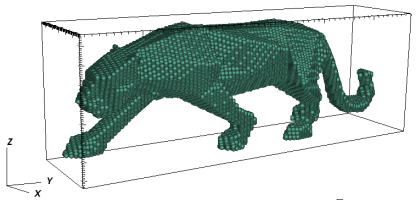


Figure 3.11: Particles created from a STL mesh.

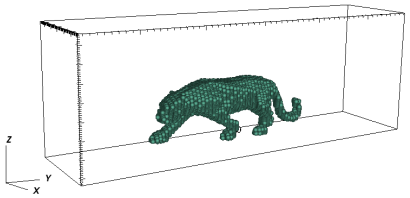


Figure 3.12: Scaled geometry.

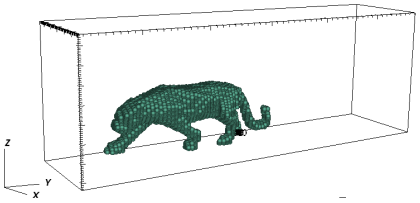


Figure 3.13: Scaled and translated geometry.

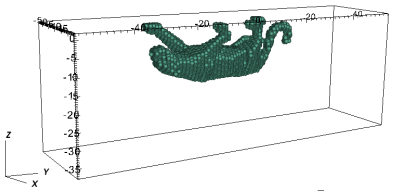


Figure 3.14: Reflected geometry in z-direction.

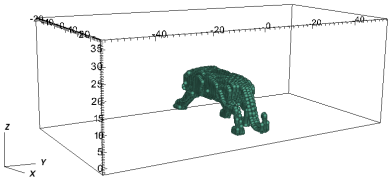


Figure 3.15: Change of coordinate sequence.

3.1.9 Boolean operations

The boolean operators on the geometry pieces include difference, intersection, and union . Multiple operators can be used to form very complex geometry pieces.

The difference takes two geometry pieces and subtracts the second geometry piece from the first geometry piece.

```
<difference>
  <box label="box_1">
    <min>[-0.75, -0.75, -0.75]</min>
    <max>[ 0.75,  0.75,  0.75]</max>
  </box>
  <sphere label="ball_1">
    <origin>[ 0.0, 0.0, 0.0]</origin>
    <radius> 1.00 </radius>
  </sphere>
</difference>
```

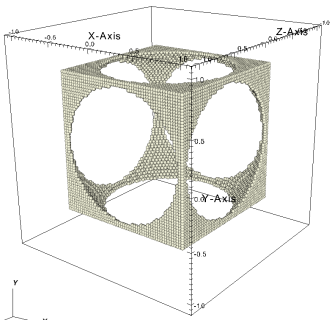


Figure 3.16: The difference of a sphere from a cube.

```
<difference>
  <sphere label="ball_1">
    <origin>[ 0.0, 0.0, 0.0]</origin>
    <radius> 1.00 </radius>
  </sphere>
  <box label="box_1">
    <min>[-0.75, -0.75, -0.75]</min>
    <max>[ 0.75,  0.75,  0.75]</max>
  </box>
</difference>
```

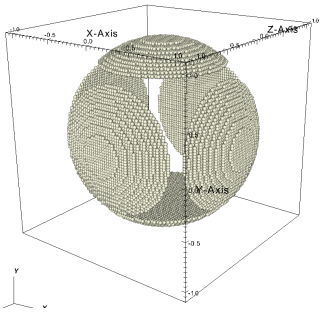


Figure 3.17: The difference of a cube from a sphere.

The intersection operator requires at least two geometry pieces in forming an intersection geometry piece.

```
<intersection>
  <box label="box_1">
    <min>[-0.75, -0.75, -0.75]</min>
    <max>[ 0.75,  0.75,  0.75]</max>
  </box>
  <sphere label="ball_1">
    <origin>[ 0.0, 0.0, 0.0]</origin>
    <radius> 1.00 </radius>
  </sphere>
</intersection>
```

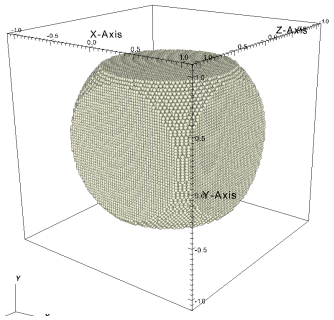


Figure 3.18: An intersection of a sphere and a cube.

The union operator aggregates a collection of geometry pieces.

```
<union>
  <box label="box_1">
    <min>[-0.75, -0.75, -0.75]</min>
    <max>[ 0.75,  0.75,  0.75]</max>
  </box>
  <sphere label="ball_1">
    <origin>[ 0.0, 0.0, 0.0]</origin>
    <radius> 1.00 </radius>
  </sphere>
</union>
```

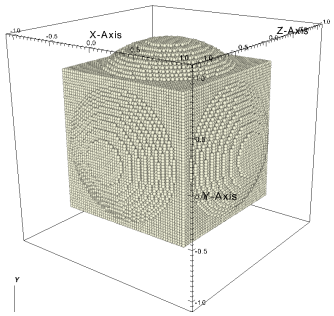


Figure 3.19: A union of a sphere and a cube.

3.2 Special geometry objects

A few special geometry objects exist that do not force the material points to be distributed according to the geometry of the background grid. These are discussed below.

3.2.1 Smooth Cylinder

smoothcyl is a geometry object that generates a body fit particle spatial distribution. This eliminates "stair-stepped" boundaries typical of the standard, grid-based, discretization scheme.

> The smoothcyl geometry is designed to work best with <interpolator>cpdi </interpolator> . Other algorithms may give erroneous answers.

The basic usage requires coordinates of the bottom and top center points, and a radius .

```
<smoothcyl label="cyl smooth">
  <bottom>[ -1.0, -1.0, -1.0]</bottom>
  <top>[ 1.0, 1.0, 1.0]</top>
  <radius> 0.5 </radius>
</smoothcyl>
```

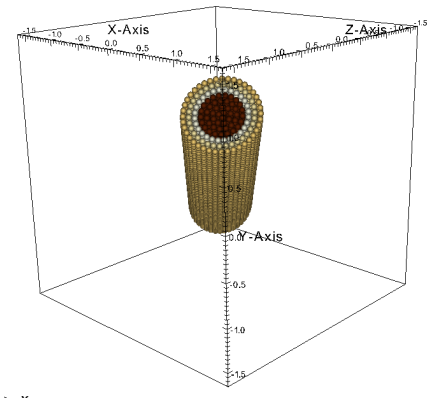


Figure 3.20: A smoothcyl geometry object. Particle volumes vary with radius.

A hollow smoothcyl can be created by specifying an additional thickness parameter. The thickness is required to be smaller than the radius. The particle and grid resolution should be chosen such that the thickness contains at least two layers of particles.

```
<smoothcyl label="cyl smooth">
  <bottom>[ -1.0, -1.0, -1.0]</bottom>
  <top>[ 1.0, 1.0, 1.0]</top>
  <radius> 0.5 </radius>
  <thickness> 0.2 </thickness>
</smoothcyl>
```

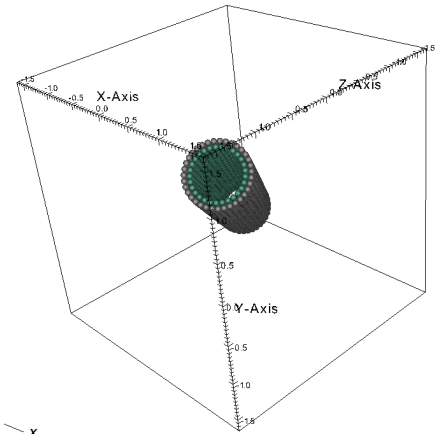


Figure 3.21: A hollow smoothcyl geometry object.

Endcaps can be added to the smoothcyl by specifying an additional endcap thickness parameter.

```
<smoothcyl label="cyl smooth">
  <bottom>[ -1.0, -1.0, -1.0]</bottom>
  <top>[ 1.0, 1.0, 1.0]</top>
  <radius> 0.5 </radius>
  <thickness> 0.2 </thickness>
  <endcap_thickness> 0.4 </endcap_thickness>
</smoothcyl>
```

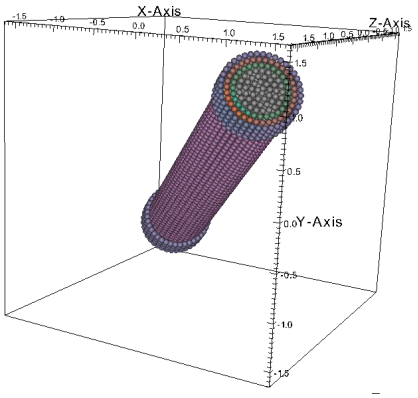


Figure 3.22: A hollow smoothcyl with endcaps geometry object.

Partial sectors of a smoothcyl can also be modeled by specifying arc information. The start angle and arc angle default to 0 and 360 and have to be specified in degrees. 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.

```
<smoothcyl label="cyl smooth">
  <bottom>[ -1.0, -1.0, -1.0]</bottom>
  <top>[ 1.0, 1.0, 1.0]</top>
  <radius> 0.5 </radius>
  <thickness> 0.2 </thickness>
  <endcap_thickness> 0.2 </endcap_thickness>
  <arc_start_angle_degree> 30 </
      arc_start_angle_degree>
  <arc_angle_degree> 90 </arc_angle_degree>
</smoothcyl>
```

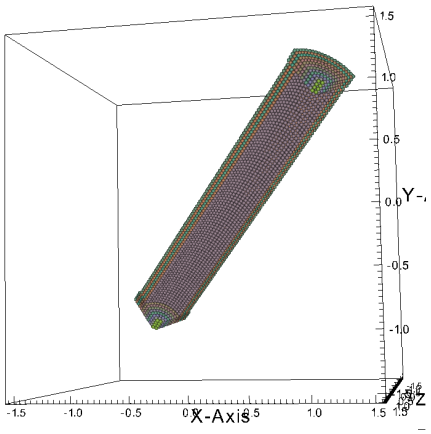


Figure 3.23: A partial hollow smoothcyl with endcaps.

Multiple smoothcyl geometries within a <geom_object> tag are 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.

3.2.2 Smooth sphere

If quarter-symmetry simulations are not required, a smoother approximation to a sphere can be modeled using the smooth_sphere tag. The center tag specifies the center of the sphere, the outer_radius tag is the outer radius, inner_radius the inner radius, and the num_radial_pts tag indicates the number of points through the thickness. Optionally, one can use the algorithm tag to determine which algorithm (spiral or equal_area) will be used to generate points and the output_file tag to save the points and volume to a file that can be read in by the file_geometry object creator.

```
<smooth_sphere label="sphere_1">
  <center>[ 0.0, 0.0, 0.0]</center>
  <outer_radius> 1.0 </outer_radius>
  <inner_radius> 0.9 </inner_radius>
  <num_radial_pts> 5 </num_radial_pts>
  <algorithm> spiral </algorithm>
  <output_file> generated_sphere.pts </
      output_file>
</smooth_sphere>
```

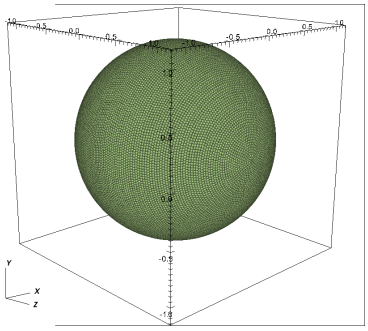


Figure 3.24: A hollow smooth_sphere geometry object generated with the spiral algorithm.

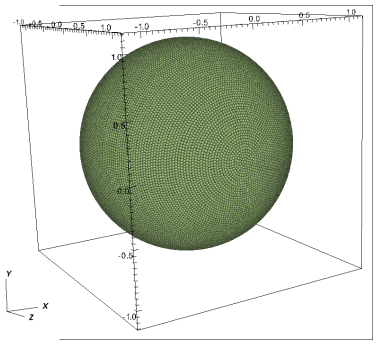


Figure 3.25: A hollow smooth_sphere geometry object generated with the equal_area algorithm.

3.2.3 Spherical membrane

The default discretization of a sphere leads to a rough approximation of the surface of a sphere. The sphere_membrane geometry allows more control over the distribution of particles while retaining the possibility of modeling quarter symmetry spheres. The origin tag specifies the center of the sphere, the radius tag is the outer radius, the thickness the thickness of the sphere wall, the num_lat label the number of points along the latitude and num_long label indicates the number of points along the longitude.

```
<sphere_membrane label="membrane">
  <origin>[ 0.0, 0.0, 0.0]</origin>
  <radius> 1.0 </radius>
  <thickness> 0.1 </thickness>
  <num_lat> 10 </num_lat>
  <num_long> 10 </num_long>
</sphere_membrane>
```

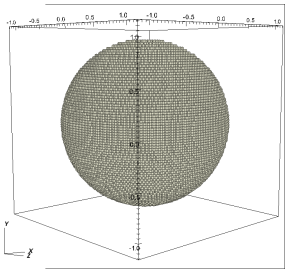


Figure 3.26: A hollow sphere_membrane geometry object.

3.2.4 Corrugated edge

The corrugated tag creates a plate with one corrugated edge. The particle spacing is determined from the grid size and the number of particles per grid cell. The input requires the minimum and maximum limits of the plate xymin and xymax, the thickness, the plate normal, the corr_edge edge that is to be corrugated (values of x+, y+, x-, and y- are allowed), the type of curve (sin or cos), the wavelength and the amplitude of the corrugation. A sample input can be:

```
<corrugated>
  <xymin>       [0.0,0.0,0.0]     </xymin>
  <xymax>       [20.0,20.0,0.0]   </xymax>
  <thickness>   1.0               </thickness>
  <normal>      [0.0,0.0,1.0]     </normal>
  <corr_edge>   x+                </corr_edge>
  <curve>       sin               </curve>
  <wavelength>  2.0               </wavelength>
  <amplitude>   2.0               </amplitude>
</corrugated>
```

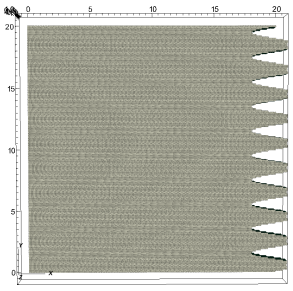


Figure 3.27: A corrugated edge geometry object.

3.2.5 Reading in an Abaqus format volume mesh

For certain geometries, none of the approaches discussed earlier work well and a three-dimensional mesh is the most viable approach for importing geometries into VAANGO. The abaqus_mesh geometry object has been designed to read in Abaqus format mesh data generated using Abaqus or meshing tools such as gmsh and stored in ASCII format in <filename>.inp. A typical input file specification for such data is:

```
<geom_object>
  <abaqus_mesh label = "torus">
    <file_name> Torus_6.inp </file_name>
  </abaqus_mesh>
  <res>[2,2,2]</res>
  <velocity>[0.0,0.0,0.0]</velocity>
  <temperature>300</temperature>
  <color>1</color>
</geom_object>
<geom_object>
  <abaqus_mesh label = "box 1">
    <file_name> Box_1.inp </file_name>
  </abaqus_mesh>
  <res>[2,2,2]</res>
  <velocity>[0.0,0.0,0.0]</velocity>
  <temperature>300</temperature>
  <color>2</color>
</geom_object>
....
```

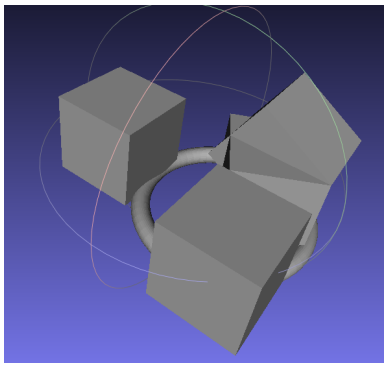


Figure 3.28: A geometry object that is meshed using gmsh.

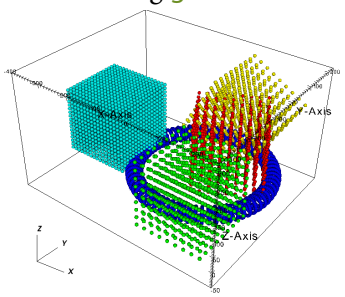


Figure 3.29: A abaqus_mesh geometry object.

If the user does not own an Abaqus license, input files can be generated using the following process.

1. Create the geometry and generate a .step file using a tool such as SolidWorks (commercial), the open-source FreeCAD, or Salome-MECA.
2. Clean up the geometry using MeshLab and make sure there are no self-intersecting triangles and non-manifold edges. Save the cleaned surface mesh geometry in the .stl format.
3. Use gmsh to generate a volume mesh from the cleaned geometry taking care to ensure a uniform distribution of tetrahedra. Save the mesh in Abaqus .inp format.

3.2.6 Specifying particle locations directly

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>
```

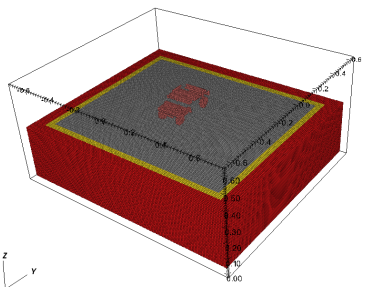


Figure 3.30: A file based geometry object.

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 vaango command. particleFileSplitter 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/particleFileSplitter executable, is:

```
particleFileSplitter input.ups
```

An associated utility that can be used to extract particle position and volume information from a UDA file is StandAlone/createFileGeomPieceFromUda . An example of its usage is

```
createFileGeomPieceFromUda file_geom_pfs_hummer_inp.uda.000 file_geom_pfs_hummer.dat
```

This utility extracts the material points and their volumes into files with a mat suffix. For example, for an input file containing 6 materials, the extracted files are named:

```
file_geom_pfs_hummer.dat.mat0
file_geom_pfs_hummer.dat.mat1
file_geom_pfs_hummer.dat.mat2
file_geom_pfs_hummer.dat.mat3
file_geom_pfs_hummer.dat.mat4
file_geom_pfs_hummer.dat.mat5
```

We can now create another input file file_geom.ups containing the file geometry objects and two patches:

```
<geom_object>
  <file label = "container">
    <name> file_geom_pfs_hummer.dat.mat0 </name>
    <var> p.volume </var>
  </file>
  <res>[2,2,2]</res>
  <velocity>[0.0,0.0,-10]</velocity>
  <temperature>300</temperature>
</geom_object>
<geom_object>
  <file label = "base">
    <name> file_geom_pfs_hummer.dat.mat1 </name>
    <var> p.volume </var>
  </file>
  <res>[2,2,2]</res>
  <velocity>[0.0,0.0,-10]</velocity>
  <temperature>300</temperature>
</geom_object>
...
```

To activate these objects we run particleFileSplitter :

```
particleFileSplitter file_geom.ups
```

to get the split files

```
file_geom_pfs_hummer.dat.mat0.0
file_geom_pfs_hummer.dat.mat0.1
file_geom_pfs_hummer.dat.mat1.0
file_geom_pfs_hummer.dat.mat1.1
....
```

After this step we can run the usual VAANGO simulation with

```
mpirun -np 2 vaango file_geom.ups
```

3.2.7 Using image data to create particles

Another option is available for initializing particle positions in MPM simulations from three-dimensional image data, such as might be collected via micro-CT scans or confocal microscopy. The image data are provided as 2-byte (16-bit/short) raw files, and usage in the input file is given as:

```
<geom_object>
  <image>
    <name> Almond_Kiss_2003112600.raw </name>
    <res> [425, 420, 260] </res>
    <threshold> [11, 200] </threshold>
  </image>
  <file label = "container">
    <name> Almond_Kiss_2003112600.pts </name>
    <format> bin </format>
  </file>
  <res>[1,1,1]</res>
  <velocity>[0.0,0.0,-10]</velocity>
  <temperature>300</temperature>
</geom_object>
```

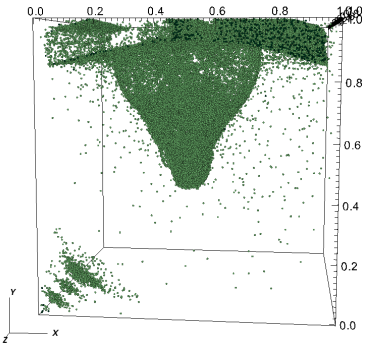


Figure 3.31: An image and file based geometry object.

Note that the geom_object resolution is typically given as $[1, 1, 1]$. The <image> section gives the name of the 3D image file, the resolution of the image in pixels in the various coordinate directions, and a threshold range. Particles will be generated at voxels within the specified range. The <file> section is the same as that described earlier.

The level section in the input file also requires the same resolution for consistency. For example,

```
<Level>
  <Box label = "1">
    <lower>[0, 0, 0]</lower>
    <upper>[1, 1, 1]</upper>
    <resolution>[425, 420, 260]</resolution>
    <patches>[1,1,1]</patches>
    <extraCells>[1,1,1]</extraCells>
  </Box>
</Level>
```

A different preprocessing utility is needed when using image data (for the same reasons described previously). Usage is as follows:

```
particleImageFileSplitter -b input.ups
```

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

A convenient tool for creating raw image files is the Slicer3D software which can be downloaded from https://www.slicer.org/. If the micro-CT image is loaded into Slider3D, it can be converted into NRRD format with separate files for the header (extension nrhd). If the compression option is turned off while saving as nrhd, a raw file containing 16-bit pixel values is created. This file can be used as input to VAANGO.

3.3 Adding particles to a simulation in progress

Vaango also contains Jim Guilkey's algorithm for inserting or transporting 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 which is 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.
Particle insertion is activated and directed with the following flags found in the MPM section of the input file:

```
  <MPM>
    <with_color> true </with_color>
    <insert_particles>true</insert_particles>
    <insert_particles_file>geom_insert_time.dat</
        insert_particles_file>
  </MPM>
```

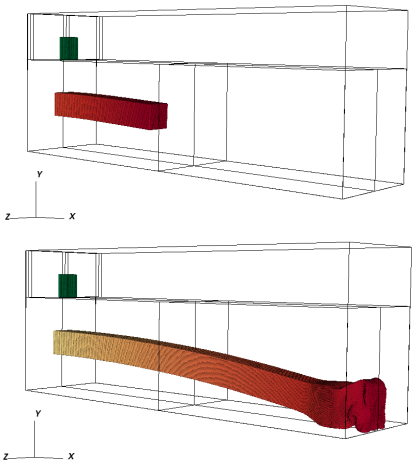


Figure 3.32: A geometry object with particle insertion.

The adjacent images show 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.
A typical input file, e.g., geom_insert.ups , will contain:

```
<MPM>
  ...
  <with_color> true </with_color>
  <insert_particles>true</insert_particles>
  <insert_particles_file>geom_insert_time.dat</insert_particles_file>
  ...
</MPM>
  <MaterialProperties>
  <MPM>
    <material name="goo">
      ....
      <include href="geom_insert_objects.xml"/>
    </material>
    ...
  </MPM>
</MaterialProperties>
<Grid>
  <BoundaryConditions>
    <Face side="x-">
      <BCType id="all" var="Dirichlet" label="Velocity">
        <value> [20.0,0.0,0.0] </value>
      </BCType>
    </Face>
    <Face side="x+">
      <BCType id="all" var="Dirichlet" label="Velocity">
        <value> [0.0,0.0,0.0] </value>
      </BCType>
    </Face>
    ...
  </BoundaryConditions>
  <Level>
    <Box label="output_box">
```

```
      <lower>[0.0, -1.0,-1.0]</lower>
      <upper>[8.0,  1.0, 1.0]</upper>
      <resolution>[160,40,40]</resolution>
      <patches>[2,1,2]</patches>
      <extraCells> [1,1,1] </extraCells>
    </Box>
    <Box label="input_box">
      <lower>[0.0,1.0,-1.0]</lower>
      <upper>[0.2,2.0, 1.0]</upper>
      <resolution>[4,20,40]</resolution>
      <patches>[2,1,2]</patches>
      <extraCells> [1,1,1] </extraCells>
    </Box>
  </Level>
</Grid>
```

The geom_insert_time.dat file is a text file containing:

```
<time> <color> <trans x> <trans y> <trans z> <new x vel> <new y vel> <new z vel>
```

For example,

```
0 0 -0.175 -1.25 0 20 0 0
0.00125 1 -0.175 -1.25 0 20 0 0
0.0025 2 -0.175 -1.25 0 20 0 0
0.00375 3 -0.175 -1.25 0 20 0 0
0.005 4 -0.175 -1.25 0 20 0 0
.....
```

During the first timestep in which the current physical time plus the calculated Δt 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.

The geom_insert_objects.xml file contains the geometry objects that will be pushed into the computational domain. An example is

```
<Uintah_Include>
  <geom_object>
    <box label="0">
      <min>[0.175, 1, -0.25]</min>
      <max>[0.2, 1.5, 0.25]</max>
    </box>
    <res>[2,2,2]</res>
    <velocity>[0.0,0.0,0.0]</velocity>
    <temperature>300.0</temperature>
    <color>0</color>
  </geom_object>
  <geom_object>
    <box label="1">
      <min>[0.175, 1, -0.25]</min>
      <max>[0.2, 1.5, 0.25]</max>
    </box>
    <res>[2,2,2]</res>
    <velocity>[0.0,0.0,0.0]</velocity>
    <temperature>300.0</temperature>
    <color>1</color>
  </geom_object>
  ....
</Uintah_Include>
```

Notice that all objects initially occupy the same position in the input box level.

An example problem exists in inputs/MPM/GeometryPieceExamples named geom_insert.ups that demonstrates particle insertion. The geom_insert_objects.xml file defines the geometry objects (also the color) and geom_insert_time.dat defines the times, translations and new velocity of the particle blocks.

3.4 OpenSCAD input file for mortar geometry

One way of creating complex geometries is to use a tool such as OpenSCAD. The script below shows how a mortar shell with fins can be created with OpenSCAD.

```
module linear_extrude_fs(height = 1, isteps = 20, twist = 0) {

  union() {

    for (i = [0 : 1: isteps-1]) {

        translate([0, 0, i*height/isteps])
        linear_extrude(
          height = height/isteps,
          twist = 0,
          scale = f_lefs((i+1)/isteps) / f_lefs(i/isteps)
        )
        scale(f_lefs(i/isteps))
        obj2D_lefs();
    }
  }
}

function f_lefs(x) =
  let(span = 150, start = 20, normpos = 45)
  sin(x*span + start) / sin(normpos);

stem_dia = 5;
stem_rad = stem_dia / 2;
stem_height = 20;
stem_faces = 12;

wing_thick = 2*PI*stem_rad/stem_faces;
echo(wing_thick);
num_wings = 6;
wing_length = 5;
wing_height = 10;
wing_points = [[0, 0], [wing_length, 0], [0, wing_height]];
wing_angle = 360/num_wings;
stem_face_angle = 360/stem_faces;

module obj2D_lefs() {
  translate([0,0])
  circle(stem_dia, $fn=stem_faces);
}

module uxo_stem(diameter = 5, height = 10) {
  cylinder(d = diameter, h = height, $fn=stem_faces);
}

module uxo_wing() {
  in_by = 1;
  translate([stem_rad - in_by, 0, -stem_height])
  rotate([90, 0, 0])
  linear_extrude(height = wing_thick, center = true)
  polygon(wing_points);
}

union() {
  color([1,0,0])
  linear_extrude_fs(height=30);

  translate([0, 0, -stem_height])
  uxo_stem(diameter = stem_dia, height = stem_height);

  for (i = [0 : num_wings-1]) {
```

```
    rotate([0, 0, 360/num_wings*i + stem_face_angle/2])
    uxo_wing();
  }
}
```

A plot of the resulting model can be seen in the adjacent figure. This input script can be easily modified to create other shapes for penetration studies.

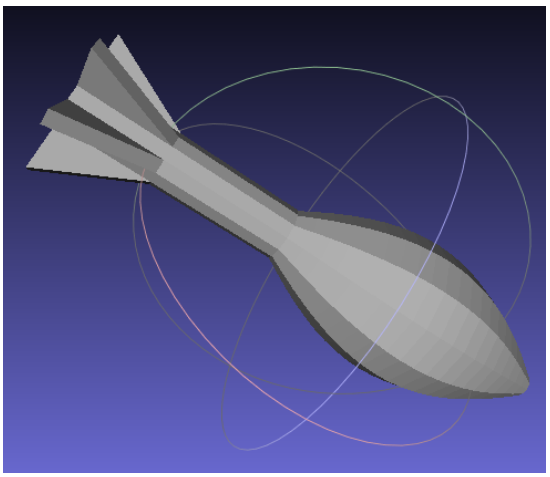
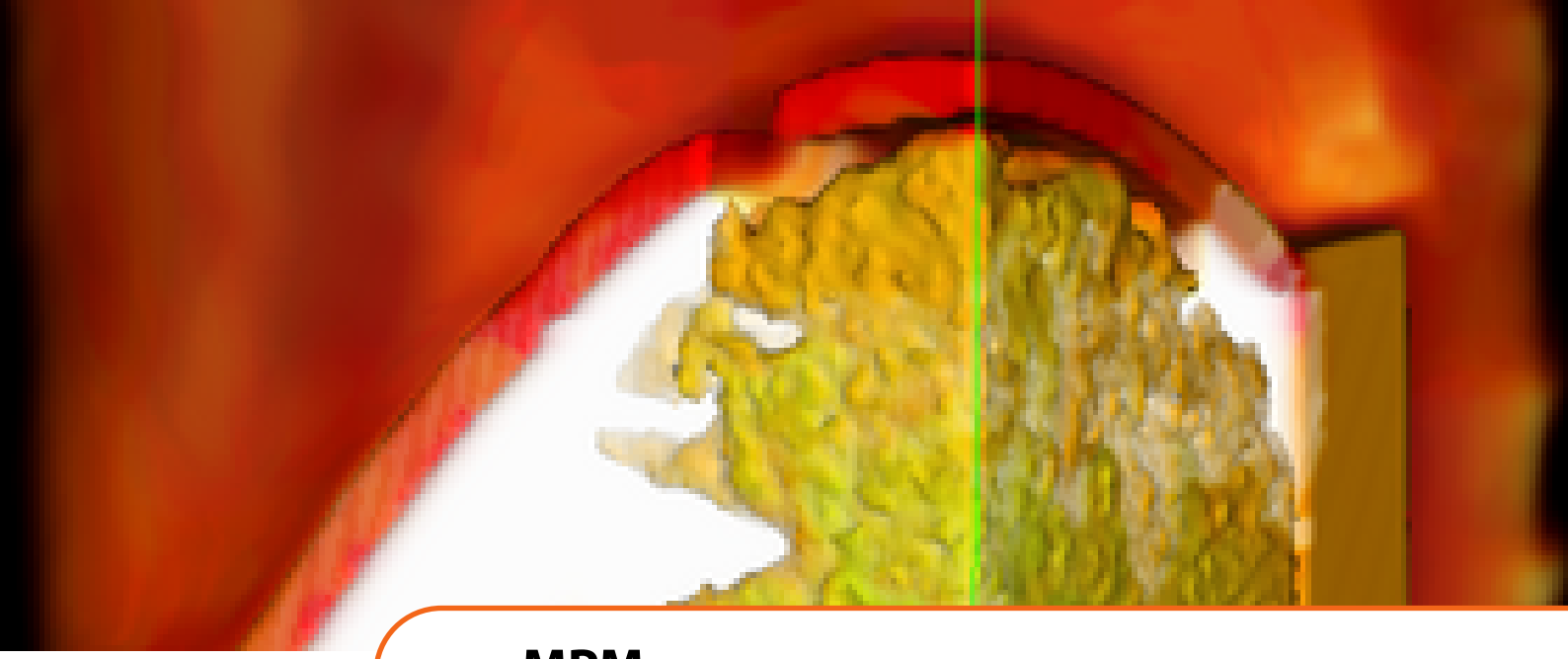


Figure 3.33: A mortar shell model generated with OpenSCAD.



4 — MPM

4.1 Introduction

The Material Point Method (MPM) is described in detail in the VAANGO theory manual. The method is used as an alternative to finite elements for problems where large deformations are expected that are not handled well by finite elements due to element locking and distortion.

An example of an MPM simulation of two disks initially approaching each other is shown in Figure 4.1. The simulation involves material points on an overlying mesh.

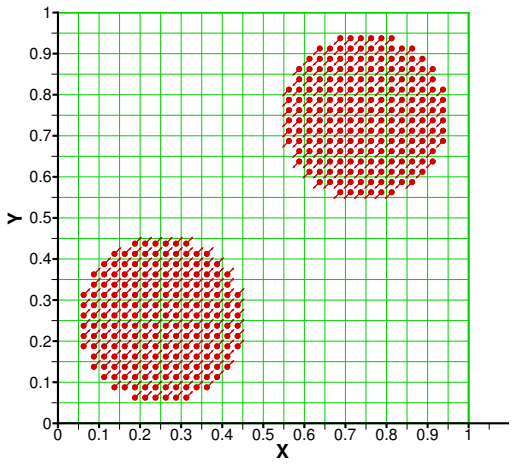


Figure 4.1: Initial particle representation of two colliding disks on an overlying mesh.

4.2 Vaango Specification

VAANGO 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.

4.2.1 Common Inputs

Each VAANGO component is invoked using a single executable called vaango , 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 4.3.

4.2.2 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>
```

4.2.3 MPM Flags

There are many options available when running MPM simulations. These are generally specified in the MPM section of the input file.

Most options have default values and are not required to be specified unless the defaults need to be changed.

Some of these options are described below:

- Axisymmetry: The axisymmetric flag is used to indicate that an axisymmetric simulation should be conducted instead of the usual three-dimensional simulation.
  Options: true or false
  Default: false.
  All axisymmetric simulations assume that the plane of rotational symmetry is the xy-plane, axisymmetric r-coordinate coincides with the grid x-direction, axisymmetric z-coordinate coincides with the grid y-direction, and that the axisymmetric θ-coordinate is in the z-direction of the grid. The Grid for axisymmetric computations must be only one cell thick in the z-direction. Plane strain calculations can be simulated by applying appropriate grid boundary conditions in 3D. Plane stress calculations are not allowed in Vaango.

```
<MPM>
  <axisymmetric> false </axisymmetric>
</MPM>
```

- Time integrator: The time_integrator flag determine the type of integration to be used in the MPM simulation.
  Options: explicit, implicit, or fracture.
  Default: explicit.
  If the explicit option is chosen integration is done using forward Euler, while implicit uses backward Euler. if the fracture option is chosen, some special integration features needed for FractureMPM are activated.

```
<MPM>
  <time_integrator> explicit </time_integrator>
</MPM>
```

- Interpolator: The interpolator flag indicates the interpolation technique to be used to interpolate from the grid to particles and to project particle data to the grid.
  Options: linear, gimp, 3rdorderBS 4thorderBS, cpdi, or cpti
  Default: linear.
  The linear option activates traditional MPM with linear grid interpolation functions. gimp activates the GIMP algorithm. 3rdorderBS and 4thorderBS activate third- and fourth-order B-spline interpolation. cpdi and cpti activate the CPDI and CPTI interpolators.

```
<MPM>
  <interpolator> gimp </interpolator>
</MPM>
```

  Options other than linear require an extraCells tag (ghost cells) when specifying the Grid:

```
<Grid>
  ...
  <extraCells> [1,1,1] </extraCells>
</Grid>
```

  For axisymmetric simulations the z-value of extraCells can be 0.
  The CPDI interpolator may also require a critical length value, cpdi_lcrit, that determines the maximum distance a particle can extend before its full extent is ignore by the CPDI algorithm.
  Default: 1.0e10.

```
<MPM>
  <interpolator> cpdi </interpolator>
  <cpdi_lcrit> 1.0e3 </cpdi_lcrit>
</MPM>
```

- Mass and velocity controls: To control the simulation in the event of small masses (particle or grid) or large velocities, VAANGO provides a few clamps on the minimum and maximum quantities that are allowed in a simulation. If these values are exceeded, particles are deleted.
  The minimum_particle_mass , minimum_mass_for_acc , and maximum_particle_velocity tags are used to set these values.
  Defaults: 3.0e-15, 1.0e-199, 3.0e105.

```
<MPM>
  <minimum_particle_mass> 1.0e-10 </minimum_particle_mass>
  <minimum_mass_for_acc> 1.0e-10 </minimum_mass_for_acc>
  <maximum_particle_velocity> 1.0e9 </maximum_particle_velocity>
</MPM>
```

- Particle color: The with_color flag indicates that color tags are being used in the simulation. Each geometry object can then be assigned a <color > tag which has an integer value.
  Options: true or false
  Default: false .
  The color flag is particularly useful for debugging simulations where a particular material is associated with a large number of geometry objects. It is also required for simulations with on-the-fly data analysis.

```
<MPM>
  <with_color> false </with_color>
</MPM>
```

- Grid reset: A fundamental feature of MPM is that the grid is reset after every time step. However, in some simulations we would like to compare VAANGO simulations with finite element simulations. A do_grid_reset flag is used to tell the simulation not to reset the grid in such simulations.
  Options: true or false
  Default: true .

```
<MPM>
  <do_grid_reset> true </do_grid_reset>
</MPM>
```

- Damping: Two types of damping are allowed in VAANGO : a velocity-based damping that is applied during the integration of the acceleration, and a Richtmyer-von Neumann artificial viscosity that is slightly more complex (see the Theory Manual for details).
  Velocity-based damping is always active and controlled by the artificial_damping_coeff flag.
  Default: 0.0

```
<MPM>
  <artificial_damping_coeff> 0.0 </artificial_damping_coeff>
</MPM>
```

  Richtmyer-von Neumann viscosity is activated by the artificial_viscosity flag and requires two parameters, artificial_viscosity_coeff1 and artificial_viscosity_coeff1 .
  Options: true or false
  Defaults: false, 0.2, 2.0

```
<MPM>
  <artificial_viscosity> false </artificial_viscosity>
  <artificial_viscosity_coeff1> 0.2 </artificial_viscosity_coeff1>
  <artificial_viscosity_coeff2> 2.0 </artificial_viscosity_coeff2>
</MPM>
```

  Viscous heating is automatically turned on when the artificial_viscosity flag is on. If artificial_viscosity is off, viscous heating can still be activated using the artificial_viscosity_heating tag. However, this is typically not allowed in a simulation.
  Options: true or false
  Default: false

```
<MPM>
  <artificial_viscosity_heating> false </artificial_viscosity_heating>
</MPM>
```

- Deformation gradient: Deformation gradient computation algorithms can be switched by activating the deformation_gradient tag. See the VAANGO Theory Manual for further information.
  Options: first_order , sybcycling , taylor_series , or cayley_hamilton
  Default: first_order

```
<MPM>
  <deformation_gradient algorithm="taylor_series">
    <num_terms> 5 </num_terms>
  </deformation_gradient>
</MPM>
```

  A parameter, num_terms , indicating the number of terms of the Taylor series to be evaluated can also be added to the tag. The default value is 1.
  A pressure stabilization step can be added to the deformation gradient computation by activating the do_pressure_stabilization flag. See the VAANGO Theory Manual for details. Options: true or false
  Default: false

```
<MPM>
  <do_pressure_stabilization> false </do_pressure_stabilization>
</MPM>
```

- Load curves: Simulations where loads are applied directly to MPM particles require load curves. The use_load_curves tag indicates whether a set of load curves is expected in the input file.
  Options: true or false
  Default: false

```
<MPM>
  <use_load_curves> false </use_load_curves>
</MPM>
```

  The default method computing the forces at particles can be inaccurate because particle surfaces areas are not computed with enough accuracy. These errors can be mitigated to some extent by using a CPDI -like interpolation scheme to interpolate particle forces to the corners of a particle domain. This feature can be activated with the use_CBDI_boundary_condition flag.
  Options: true or false
  Default: false

```
<MPM>
  <use_CBDI_boundary_condition> false </use_CBDI_boundary_condition>
</MPM>
```

  If the use_load_curves flags is false but you still want some way of controlling the magnitude of the external force applied to a particle, you can use the forceBC_force_increment_factor tag to increment the external force at all the particles every timestep. However, this flag is ideally avoided in simulations.
  Default: 0.0

```
<MPM>
  <forceBC_force_increment_factor> 0.0 </forceBC_force_increment_factor>
</MPM>
```

- Prescribed deformations: Some special simulations require a set of prescribed deformations to be applied to particles instead of the standard method of computing deformation gradients from velocity gradients. The simulation can be directed to search for these by activating the use_prescribed_deformation flag. The prescribed values of deformation are stored in a prescribed_deformation_file . An exact_deformation flag can also be specified to make sure that the time increment Δt is changed to match prescribed deformation times exactly.

Options: true or false
Defaults: false, false

```
<MPM>
  <use_prescribed_deformation> true </use_prescribed_deformation>
  <prescribed_deformation_file> deformation_file.dat </
      prescribed_deformation_file>
  <exact_deformation> false </exact_deformation>
</MPM>
```

- Rotating coordinate system: VAANGO also give you the option of performing your computations write respect to a rotating coordinate system. This feature is activated by the rotating_coordinate_system tag and requires a center of rotation, an axis, and angular rotation speed, and optionally, a body reference point.
Defaults: [0, 0, 0], [0, 0, 1], 0.0, [0, 0, 0]

```
<MPM>
  <rotating_coordinate_system>
    <rotation_center> [1, 0, 0] </rotation_center>
    <rotation_axis> [0, 0, 1] </rotation_axis>
    <rotation_speed_angular> 10 </rotation_speed_angular>
    <body_reference_point> [0, 0, 0] </body_reference_point>
  </rotating_coordinate_system>
</MPM>
```

The body force that is generated by rotation (or any other body force such as gravity), can be used to initialize particle stresses by activating the initialize_stress_using_body_force tag.
Options: true or false
Defaults: false, false

```
<MPM>
  <initialize_stress_using_body_force> false </
      initialize_stress_using_body_force>
</MPM>
```

- Surface normals: Normals to object surfaces are estimated by default during every time step. These are required for contact computations. For some bimaterial contact problems, these computed normals may not be collinear. To force collinearity, an average of the two normals is taken when the collinear_bimaterial_contact_normals flag is activated.
Options: true or false
Default: false

```
<MPM>
  <collinear_bimaterial_contact_normals> false </
      collinear_bimaterial_contact_normals>
</MPM>
```

- Boundary traction faces: For some special simulations, we may require the contact tractions at the domain boundaries to be computed and saved. This feature is activated using the boundary_traction_faces flag which takes a list of text strings as input. The allowed values of xminus ,
Options: xplus , yminus , yplus , zminus , and zplus .
Default: [] .

```
<MPM>
  <boundary_traction_faces>[xminus ,xplus ,zminus]</boundary_traction_faces>
</MPM>
```

- Cohesive zones: The cohesive zone capability of VAANGO can be activated with the use_cohesive_zones tag. See the VAANGO Theory Manual for details.
Options: true or false
Default: false

```
<MPM>
  <use_cohesive_zones> false </use_cohesive_zones>
</MPM>
```

- Coupled thermal problems: MPM can also be used to solve heat conduction problems and certain thermal effects are implemented in Vaango .
  The most basic effect is thermal expansion which can be activated with the do_thermal_expansion flag.
  Options: true or false
  Default: false

```
<MPM>
  <do_thermal_expansion> false </do_thermal_expansion>
</MPM>
```

  Another effect that can be activated using a flag is heating due to contact friction, do_contact_friction_heating .
  Options: true or false
  Default: false

```
<MPM>
  <do_contact_friction_heating> false </do_contact_friction_heating>
</MPM>
```

  The full heat conduction machinery in Vaango can be activated using one of three flags: do_explicit_heat_conduction , do_implicit_heat_conduction , or do_transient_implicit_heat_conduction . See the Vaango Theory Manual for further information.
  Options: true or false
  Defaults: false, false, false

```
<MPM>
  <do_explicit_heat_conduction> false </do_explicit_heat_conduction>
  <do_implicit_heat_conduction> false </do_implicit_heat_conduction>
  <do_transient_implicit_heat_conduction> false </
      do_transient_implicit_heat_conduction>
</MPM>
```

  If PETSc is used for implicit heat conduction, you can flush the solver using the extra_solver_flushes flag to save memory. However, this flag currently has no effect on the solve.
- Particle removal: Failed or "localized" particles can be removed or isolated from a simulation using the erosion tag. Available erosion algorithms are applied directly only to selected material models, but can be applied in general when used with conjunction with the material model tag <d_basic_damage> .
  Options: none , RemoveMass , AllowNoTension , ZeroStress , AllowNoShear , or BrittleDamage
  Default: none

```
<MPM>
  <erosion algorithm="BrittleDamage"> </erosion>
</MPM>
```

  Vaango also provides the option of deleting particles that have been isolated from the rest of the simulation by checking failed particles that are inside a cell containing only a few other particles. This capability can be activated by using the flag delete_rogue_particles .
  Options: true or false
  Default: false

```
<MPM>
  <delete_rogue_particles> false </delete_rogue_particles>
</MPM>
```

- Particle addition/insertion: There are two types of particle addition algorithms in Vaango :
  1. Convert failed particles to a different material
  2. Insert particles into an on-going simulation

  To allow the conversion of failed particles, the input file requires a the material to be converted and the material a failed particle is to be converted into. Given these requirements, the algorithm can be

activated using can_add_MPM_material in conjunction with the flags create_new_particles and manual_new_material.
Options: true or false
Defaults: false, false, false

```
<MPM>
  <can_add_MPM_material> true </can_add_MPM_material>
  <create_new_particles> true </create_new_particles>
  <manual_new_material> true </manual_new_material>
</MPM>
```

Particle insertion is used for simulations similar to those shown in Section 3.3. The flag insert_particles is used to activate this functionality.
Options: true or false
Default: false

```
<MPM>
  <insert_particles> true </insert_particles>
  <insert_particles_file> insert_particle_times.dat </insert_particles_file>
</MPM>
```

- Manufactured solutions: Several manufactured solutions have been implemented in Vaango to test the MPM algorithm and material models. A manufactured solution simulation can be run by activating the run_MMS_problem flag.
  Options: none, AxisAligned, GeneralizedVortex, ExpandingRing, AxisAligned3L, UniaxialStrainHarmonic, UniaxialStrainHomogeneousLinear, or UniaxialStrainHomogeneousQuadratic
  Default: none

```
<MPM>
  <run_MMS_problem> GeneralizedVortex </run_MMS_problem>
</MPM>
```

- Adaptive mesh refinement: The MPM implementation in Vaango also allows for a rudimentary adaptive mesh and particle refinement capability. This feature can be activated using the AMR flag.
  Options: true or false Defaults: false

```
<MPM>
  <AMR> true </AMR>
  <use_gradient_enhanced_velocity_projection> false </
      use_gradient_enhanced_velocity_projection>
  <refine_particles> true </refine_particles>
</MPM>
```

The MPM flags used in Vaango change frequently and the latest set can be found in inputs/UPS_SPEC/mpm_spec.xml and MPM/MPMFlags.cc.

4.2.4 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 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 3.1. 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 4.4 is a description of how this field is used to identify particles for on the fly data extraction.

An arbitrary 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 4.2.5.

```
<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>
```

4.2.5 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 4.2.4 for an example of their proper placement in the input file, namely, after all of the MPM materials have been described.

Null contact

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>
```

Single velocity 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 VAANGO -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.)

Friction contact

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.[1]. 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 improved implementation of the Bardenhagen contact model can be accessed with the alternative tag:

```
<contact>
   <type>friction_bard</type>
   <materials>[0,1,2]</materials>
   <mu> 0.5 </mu>
</contact>
```

A more recent friction contact algorithm [2] uses logistic regression to identify the interface between two (or more) materials in contact. This model can be activated using:

```
<contact>
   <type>friction_LR</type>
   <materials>[0,1,2]</materials>
   <mu> 0.5 </mu>
</contact>
```

For more detail, please see the section on Logistic regression friction contact in the theory manual.

> The logistic regression friction contact model is not activated if there is only one material but multiple geometry objects in the input file. Please associate objects that you would like to interact via friction contact with separate materials.

Approach 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>
```

Specified/Rigid 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>
```

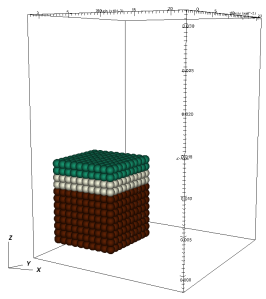
For reasons of backwards compatibility, the specified contact type is interchangable with rigid. By default, when either model is chosen, material 0 is the "rigid" material, although this can be overridden by the use of the master_material field. If no filename field is specified, then the particles 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:

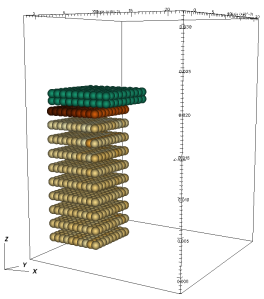
```
time1 velocity_x1 velocity_y1 velocity_z1
time2 velocity_x2 velocity_y2 velocity_z2
      .
      .
      .
```

The velocity of the rigid material particles will be set to these values, based on linear interpolation between times, until stop_time is reached.

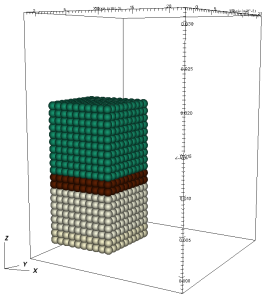
The figures below show how the specified contact algorithm can be used to simulate uniaxial tension and compression tests. The green block is the rigid master material that moves with a specified velocity.



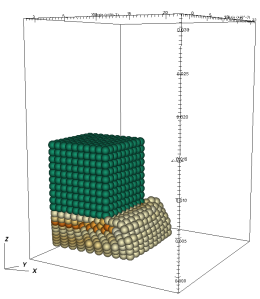
Initial state for tension test.



Deformed state for tension test.



Initial state for compression test.

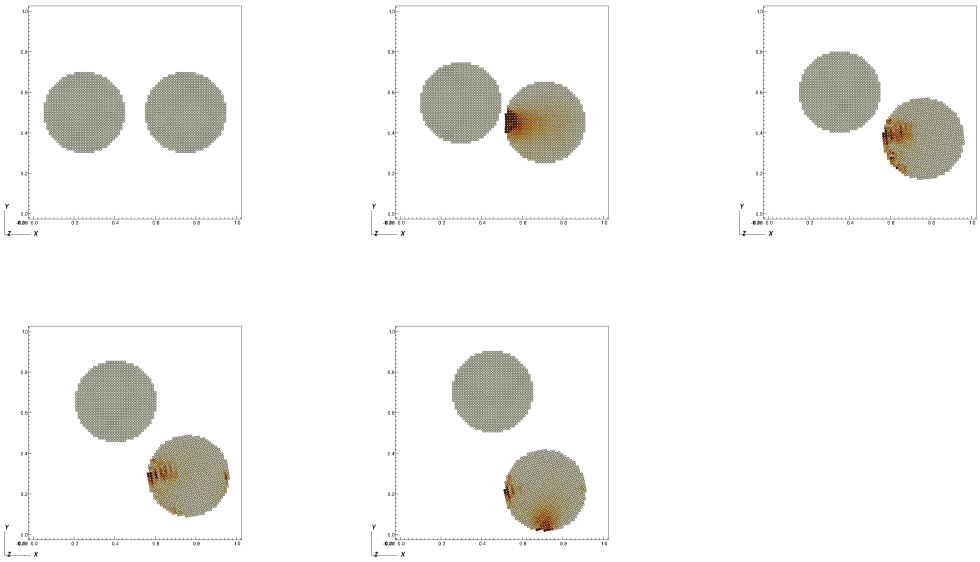


Deformed state for compression test.

An alternative way of performing this type of contact is via surface normals. In that case, the input has the form:

```
<contact>
  <type>specified</type>
  <master_material> 0 </master_material>
  <master_material_is_rigid> true </master_material_is_rigid>
  <normal_only> true </normal_only>
</contact>
```

This allows specified contact with arbitrarily moving geometries. In the figures below, note the motion of the rigid disk to the left and compare it to that of the the deformable disk which shows stress waves. The motion of the rigid disk is unaffected by the contact event.



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.

Composite contact

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 o 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.

4.3 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 Vaango 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.

4.3.1 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 Vaango , 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 4.2.3) of the input file by setting use_load_curves to true . The default value is false .

In VAANGO , 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/PressureBC.cc). 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.

4.4 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), Vaango 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 <with_color>true </with_color> tag in the MPM section of the input file. (See Section 4.2.3.)

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 6 for more information.

4.5 Prescribed Motion

The prescribed motion capability in Vaango 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>
    <use_prescribed_deformation>true</use_prescribed_deformation>
</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.

4.6 Cohesive Zones

A cohesive zone formulation is available in VAANGO based on the description by Daphalapurkar, et al. [3]. As in their implementation, that in VAANGO 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 VAANGO are those given in Eq. 15 of [3]. These require 4 input parameters. They are σ_{max}, τ_{max}, δ_n and δ_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 Vaango 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.

4.7 Examples

The following examples are meant to be illustrative of a variety of capabilities of Vaango -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 also be found in:

```
inputs/MPM
```

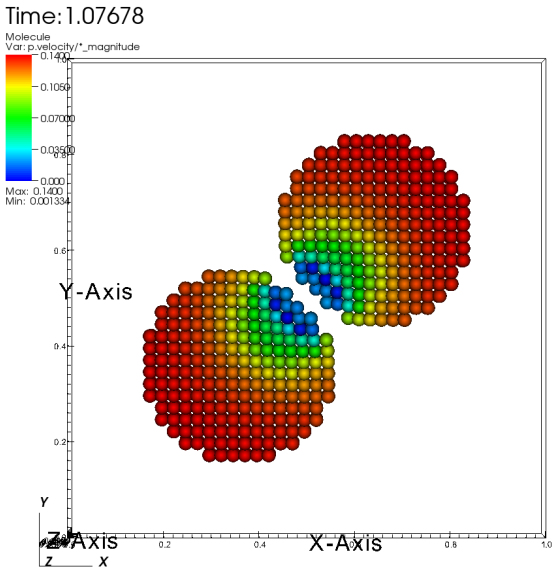


Figure 4.2: Colliding elastic disks. Particles colored according to velocity magnitude.

Colliding Disks

Problem Description

This is an implementation of an example calculation from [4] 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: vaango disks_sulsky.ups
Simulation Domain: 1.0 x 1.0 x 0.05 m
Cell Spacing:
.05 x .05 x .05 m (Level 0)
Example Runtimes:
4 seconds (1 processor, 3.16 GHz Xeon)

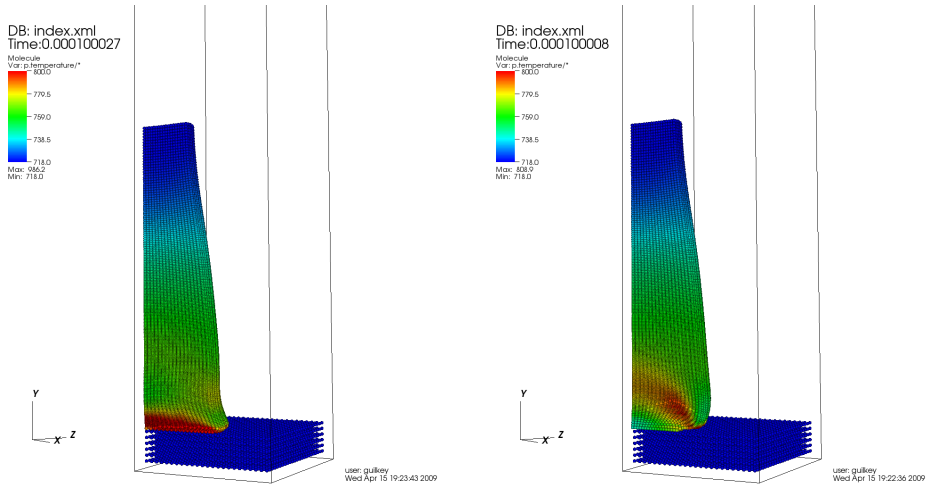
Physical time simulated: 3.0 seconds
Associate VisIt session: disks.session

Results

Figure 4.2 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 available and can be plotted to create a graph similar to that in Fig. 5a of [4]. e.g. using gnuplot:

```
cd disks.uda.000
gnuplot
gnuplot> plot "StrainEnergy.dat", "KineticEnergy.dat"
gnuplot> quit
```



(a) Sliding allowed (b) Sliding prohibited

Figure 4.3: Taylor impact simulation with (a) sliding (b) no sliding, between cylinder and target. Particles colored according to temperature.

Taylor Impact Test

Problem Description

This is a simulation of an Taylor impact experiment calculation from [5] 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 μs.

Simulation Specifics

Component used: MPM
Input file name: taylorImpact.ups
Command used to run input file: vaango inputs/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 μseconds
Associate VisIt session: taylorImpact.session

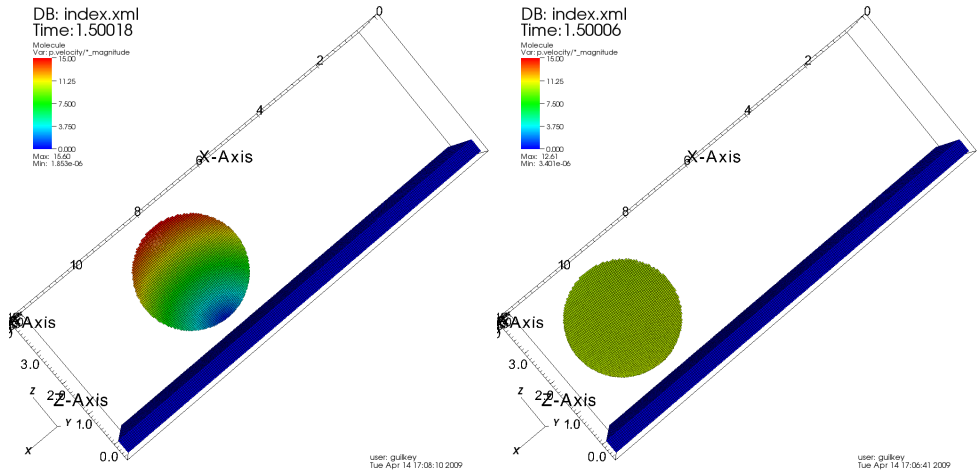
Results

Figure 4.3(a) 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 4.3(b) 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>
```



(a) Friction coefficient μ = 0.495. (b) Friction coefficient μ = 0.

Figure 4.4: 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. 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.

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 [1]. Frictional contact, using a friction coefficient of μ = 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 μ = 0.0. The difference in the results is shown below.

Simulation Specifics

Component used: MPM
Input file name: inclinedPlaneSphere.ups
Command used to run input file: vaango inputs/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 4.4(a) and Figure 4.4(b) show snapshots of the simulation, as the sphere is about halfway down the plane.

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 [6]. 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 receives 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 vaango. 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:

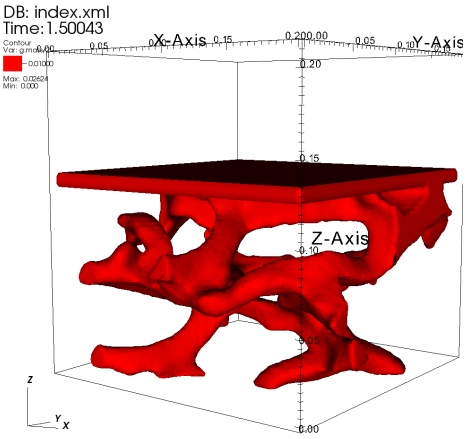
```
mpirun -np NP vaango foam.ups
```

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)

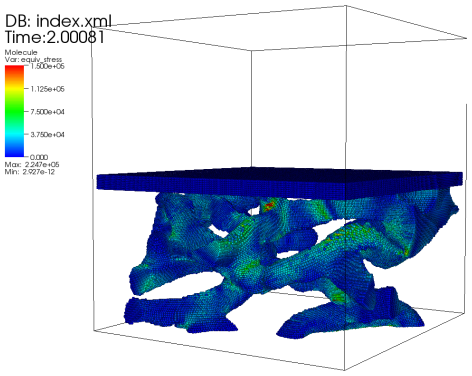
Physical time simulated: 3.75 seconds
Associated VisIt session 1: foam.iso.session
Associated VisIt session 2: foam.part.session

Results

Figure 4.5(a) shows a snapshot of the simulation via isosurfacing, as the foam is at about 50% compaction.



(a) Rendered via isosurfacing.



(b) Rendered with particles colored by equivalent stress.

Figure 4.5: Compaction of a foam microstructure.

Figure 4.5(b) 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.

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
Input file name: holePlate.ups
Command used to run input file: vaango inputs/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 4.6 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.

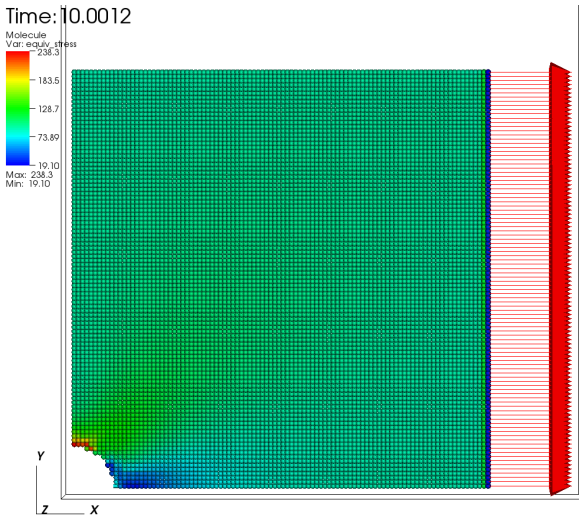


Figure 4.6: Elastic plate with a hole loaded in tension. Particles are colored by equivalent stress, vectors indicate applied load.

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 submodels 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: vaango inputs/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 μseconds
Associate VisIt session: WSphereSteel.session

Results

Figure 4.7 shows the initial configuration for this simulation, with particles colored by the magnitude of their velocity. Figure 4.8 shows the state of the simulation after 4μseconds this simulation, with particles still colored by velocity magnitude.

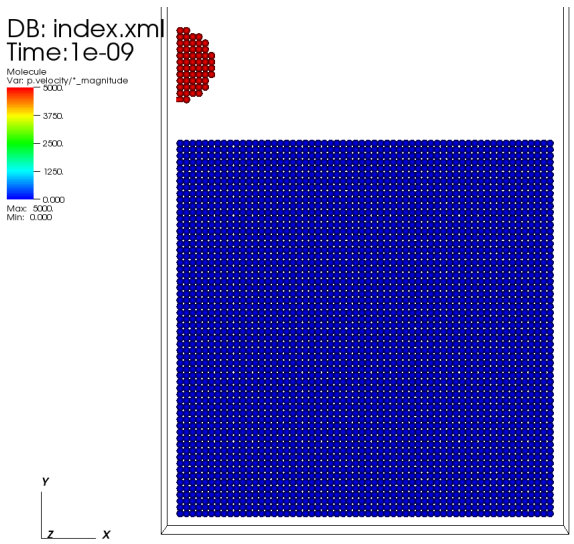


Figure 4.7: Initial configuration of hypervelocity impact of tungsten sphere into a steel target. Particles are colored by velocity magnitude.

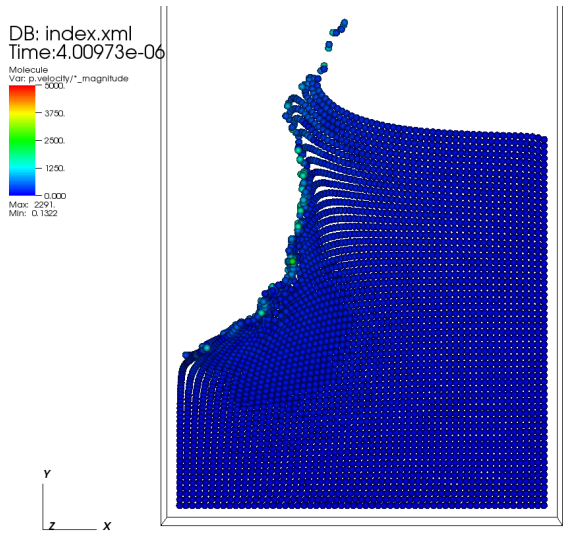


Figure 4.8: State of the tunsgsten and steel after 4μseconds. Particles are colored by velocity magnitude.

4.8 Method Of Manufactured Solutions (MMS)

There are three manufactured solutions available in Vaango 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
```

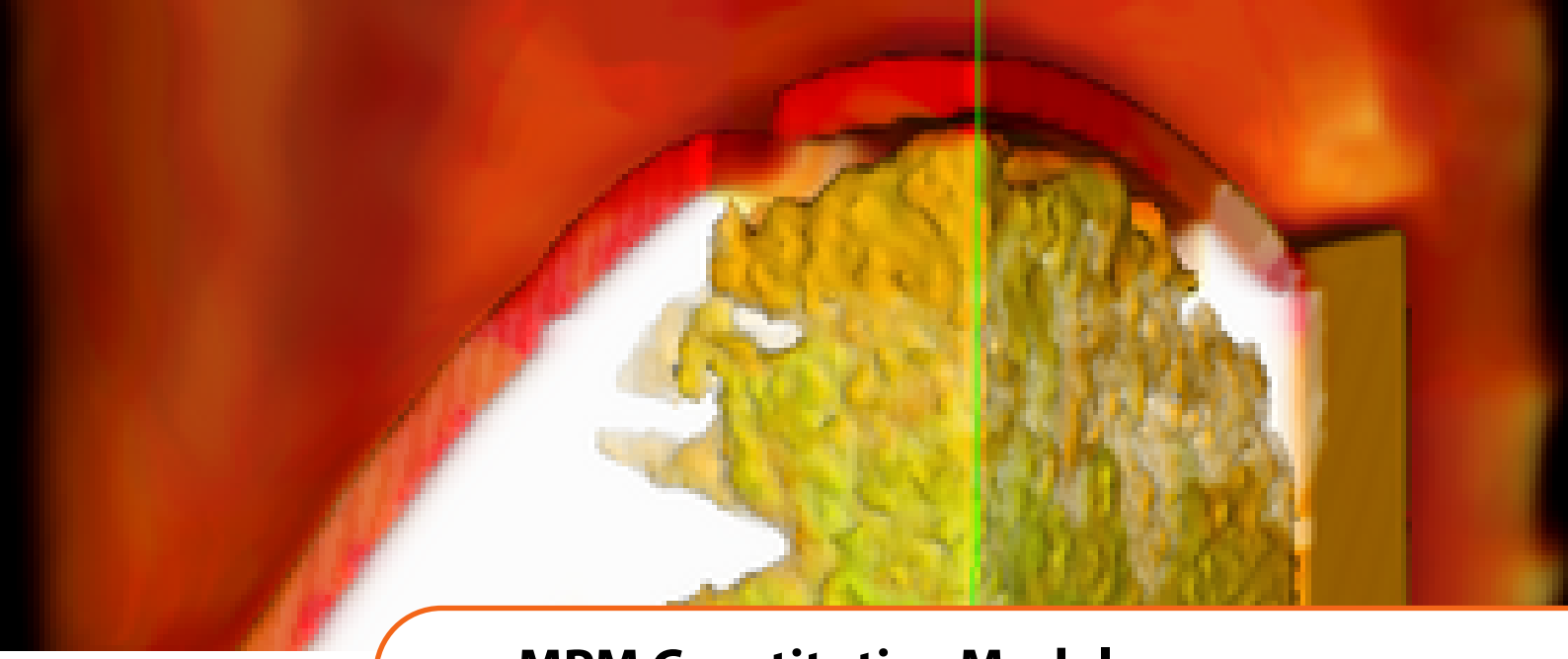
Following are the sequential steps to add a new MMS to the existing framework.
1) In MPMFlags.cc, add another if condition for the new MMS string in the following loop

```
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 existing 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).



5 — MPM 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 in this section.

5.1 Special materials

5.1.1 Rigid Material

This model was designed for use with the specified contact model described in Section 4.2.5. It is designed to compute zero stress and identity deformation of the material, and is basically a fast place-holder for materials that should not develop any stress.

```
<constitutive_model type="rigid">
</constitutive_model>
```

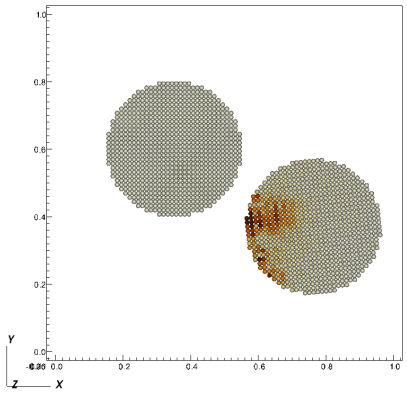


Figure 5.1: A rigid disk (left) interacting with a deformable disk (right).

5.1.2 Ideal Gas

The ideal gas material provides an equation of state capability that allows compression but no shear or tension. Usage is:

```
<constitutive_model type="ideal_gas">
  <gamma> 1.4 </gamma>
  <specific_heat> 800.0 </specific_heat>
  <reference_pressure> 101325.0 </reference_pressure>
</constitutive_model>
```

5.1.3 Water

This is a model for water, reported in [7]. The P-V relationship is given by:

$$p = \kappa \left[\left(\frac{\rho}{\rho_0} \right)^{\gamma} - 1 \right] \tag{5.1}$$

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>
```

5.2 Elastic materials

Please refer to the VAANGO Theory Manual for brief descriptions of the theory used in these material models.

5.2.1 Hypoelastic material

A hypoelastic material model can be specified in the UPS input file using:

```
<constitutive_model type="hypo_elastic">
  <K> 32.0e6 </K>
  <G> 12.0e6 </G>
  <alpha> 1.0e-4 </alpha>
</constitutive_model>
```

Here K is the bulk modulus, G is the shear modulus, and α is the coefficient of thermal expansion. If the Young's modulus (E) and Poisson's ratio (ν) of the material are known, the bulk and shear modulus can be computed using

$$K = \frac{E}{3(1-2\nu)} \quad \text{and} \quad G = \frac{E}{2(1+\nu)}. \tag{5.2}$$

5.2.2 Compressible Mooney-Rivlin Model

This model is generally parameterized for rubber type materials. Usage is as follows:

```
<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, and he_PR is the Poisson's ratio (ν). The initial shear modulus, G, is related to the two Mooney-Rivlin constants and the initial bulk modulus can be computed from G and ν using

$$G = 2(C_1 + C_2) \quad \text{and} \quad K = \frac{2G(1+\nu)}{3(1-2\nu)}. \tag{5.3}$$

5.2.3 Compressible Neo-Hookean Model

There are implementations of several hyperelastic-plastic model described by Simo and Hughes[8] (pp. 307 – 321). The model is dubbed "Unified Compressible Neo-Hookean 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 underlying 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 does not exist yet.

Purely elastic

For purely-elastic compressible neo-Hookean material simulations, the input has the form:

```
<constitutive_model type="UCNH">
  <bulk_modulus>   8.9e9  </bulk_modulus>
  <shear_modulus>  3.52e9 </shear_modulus>
  <useModifiedEOS> true   </useModifiedEOS>
</constitutive_model>
```

Alternatively, this model can be invoked usig the <comp_neo_hook > tag:

```
<constitutive_model type="comp_neo_hook">
  <bulk_modulus>   8.9e9  </bulk_modulus>
  <shear_modulus>  3.52e9 </shear_modulus>
  <useModifiedEOS> true   </useModifiedEOS>
</constitutive_model>
```

Elastic with brittle damage

The cnh_damage tag or the <useDamage > tag tells VAANGO 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 is:

```
<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 as discussed below.

Elastic-Plastic (J_2-plasticity)

For simulations with plasticity enabled, use

```
<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>
```

This model includes J_2-plasticity with isotropic linear hardening, and can be alternatively invoked using the comp_neo_hook_plastic tag:

```
<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>
```

Elastic-Plastic with damage

The UCNH model with damage can alternatively be invoked with the cnhp_damage tag. This constitutive model is an extension of the hyperelastic-plastic neo-Hookean model to failure based on a stress or strain, thus yielding an elastic-plastic model with failure. Note that the post-failure behaviour of simulations is not always robust.

When the cnhp_damage tag is used instead of UCNH, the input section for damage and plasticity is similar to that for UCNH without <useDamage > and <usePlasticity >.

A fairly sophisticated means of seeding explicit material heterogeneity is also provided for. To use these features the following four steps are required:

1. Erosion algorithm: To allow for failure (by material point erosion), 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. Failure criterion: The failure criterion 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 MohrCoulomb failure criterion is given by

$$\frac{\sigma_3 - \sigma_1}{2} = c\cos(\phi) - \frac{\sigma_3 + \sigma_1}{2}\sin(\phi) \tag{5.4}$$

where σ_i are the ordered principal stresses, positive in tension ($\sigma_3 > \sigma_2 > \sigma_1$). Note, the MohrCoulomb failure surface requires a friction angle, ϕ, (in degrees):

```
<friction_angle> friction angle </friction_angle>
```

and the cohesion (c) which 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).

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: 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: 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

$$\left(\frac{\overline{V}}{V}\right)^{1/n} \tag{5.5}$$

where V is the particle volume, a function of numerical resolution. The reference volume, $\overline{V}$ 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, generally consistent with the observation that smaller quantities of material contain fewer critical flaws.

Post-failure behavior

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.

Brittle damage

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 τ_b, related to the maximum principal strain ϵ_{max}, has exceeded a threshold value r_o^b

$$\sigma_{kk} > 0, \qquad \tau_b = \sqrt{E\epsilon_{max}^2} \geq r_o^b \tag{5.6}$$

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 Vaango but should be used with caution since stiffening is more prone to instability. The softening function for brittle damage is assumed to be

$$d(\tau_b) = \frac{0.999}{D}\left(\frac{1+D}{1+D\exp^{-C(\tau_b - r_o^b)}}\right) \tag{5.7}$$

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_o (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.

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.

Tag	Symbol	Description
brittle_damage_initial_threshold	r_o^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)

When a particle is damaged, the value of the particle variable p.damage can be output in the DataArchiver section of the input file.

5.3 Elastic modulus models

For selected hypoelasticity-based material models, the model used to compute the bulk and shear modulus can be chosen separately. Some of these elastic moduli models are discussed below.

5.3.1 Support vector model

The tabular plasticity model discussed later in this manual can be combined with a support vector model for the bulk modulus. The support vector based elastic modulus model is specified using an input description of the following form.

```
<constitutive_model type=".....">
  <elastic_moduli_model type="support_vector">
    <filename>SVR_fit.json</filename>
    <G0>3500</G0>
    <nu>0.189</nu>
  </elastic_moduli_model>
  ......
</constitutive_model>
```

The JSON input file is required to contain a support vector regression fit to pressure data as a function of the total volumetric strain and the plastic volumetric strain. The format of the JSON file should be of the following form:

```
{
  "X_var": ["Total volumetric strain (\%)", "Plastic volumetric strain (\%)"],
  "y_var": "Pressure (MPa)",
  "X_conversion_factor": [0.01, 0.01],
  "y_conversion_factor": 1000000.0,
  "X_scale": [0.018115942028985508, 0.031249261081060988],
  "X_min": [-10.0, 0.0],
  "X_max": [45.2, 32.00075667088534],
  "y_scale": [0.00016356411948678396],
  "y_min": [-1112.690309606444],
  "y_max": [5001.12],
  "gamma": 4.7474110414223025,
  "support_vectors": [[0.9998641304347827, 1.0], [0.8832644927536233, 1.0],
    [0.9954365942028985, 1.0], [0.6534438405797102, 0.7060732743351001],
    [0.8108822463768116, 1.0], [0.7982101449275363, 0.943522446672424], ...],
  "dual_coeffs": [[10.0, 10.0, 10.0, -10.0, 10.0, 10.0, 2.5932059232147773, 10.0, ...],
  "intercept": [0.8428855972115061]
}
```

An initial value of Go is used if nu is less than -1.0 or greater than 0.5. Otherwise the shear modulus is computed from the bulk modulus model using the value of nu at the Poisson's ratio.

An example of fitting a support vector regression model in Python is shown below.

```
#!/usr/bin/env python
# coding: utf-8
#
# Load packages
#
import numpy as np
from sklearn.svm import SVR, NuSVR
from sklearn import preprocessing
from sklearn.model_selection import StratifiedShuffleSplit
from sklearn.model_selection import ShuffleSplit
from sklearn.model_selection import GridSearchCV
from sklearn.model_selection import train_test_split
from sklearn.model_selection import cross_val_score
import pandas as pd
import matplotlib.pyplot as plt
import json
from json import JSONEncoder
import sys
#
# Load input data from CSV files
#
unload_09 = pd.read_csv("./Sand_unload_09.csv", header=0, skiprows=1)
unload_18 = pd.read_csv("./Sand_unload_18.csv", header=0, skiprows=1)
unload_27 = pd.read_csv("./Sand_unload_27.csv", header=0, skiprows=1)
unload_36 = pd.read_csv("./Sand_unload_36.csv", header=0, skiprows=1)
unload_45 = pd.read_csv("./Sand_unload_45.csv", header=0, skiprows=1)
#
# Compute plastic strains from intersection with strain axis
#
t_09 = -unload_09.iloc[-1,1]/(unload_09.iloc[-4,1] - unload_09.iloc[-1,1])
t_18 = -unload_18.iloc[-1,1]/(unload_18.iloc[-4,1] - unload_18.iloc[-1,1])
t_27 = -unload_27.iloc[-1,1]/(unload_27.iloc[-4,1] - unload_27.iloc[-1,1])
t_36 = -unload_36.iloc[-1,1]/(unload_36.iloc[-4,1] - unload_36.iloc[-1,1])
t_45 = -unload_45.iloc[-1,1]/(unload_45.iloc[-4,1] - unload_45.iloc[-1,1])
eps_p_09 = (1 - t_09) * unload_09.iloc[-1,0] + t_09 * unload_09.iloc[-4,0]
eps_p_18 = (1 - t_18) * unload_18.iloc[-1,0] + t_18 * unload_18.iloc[-4,0]
eps_p_27 = (1 - t_27) * unload_27.iloc[-1,0] + t_27 * unload_27.iloc[-4,0]
eps_p_36 = (1 - t_36) * unload_36.iloc[-1,0] + t_36 * unload_36.iloc[-4,0]
eps_p_45 = (1 - t_45) * unload_45.iloc[-1,0] + t_45 * unload_45.iloc[-4,0]
#
# Add intersection point to data
#
unload_09.loc[unload_09.index.max()+1] = (eps_p_09, 0.0)
unload_18.loc[unload_18.index.max()+1] = (eps_p_18, 0.0)
```

```
unload_27.loc[unload_27.index.max()+1] = (eps_p_27, 0.0)
unload_36.loc[unload_36.index.max()+1] = (eps_p_36, 0.0)
unload_45.loc[unload_45.index.max()+1] = (eps_p_45, 0.0)
#
# Separate out strain and pressure data
#
eps_09 = unload_09.iloc[:,0]
eps_18 = unload_18.iloc[:,0]
eps_27 = unload_27.iloc[:,0]
eps_36 = unload_36.iloc[:,0]
eps_45 = unload_45.iloc[:,0]
p_09 = unload_09.iloc[:,1]
p_18 = unload_18.iloc[:,1]
p_27 = unload_27.iloc[:,1]
p_36 = unload_36.iloc[:,1]
p_45 = unload_45.iloc[:,1]
eps_p = (eps_p_09, eps_p_18, eps_p_27, eps_p_36, eps_p_45, 0.0)
#
# Create data for 0% plastic strain
#
eps_p_00_data = pd.DataFrame({'TotalStrainVol' : eps_09 - eps_p_09, 'Pressure' : p_09})
eps_00 = eps_p_00_data.iloc[:,0]
p_00 = eps_p_00_data.iloc[:,1]
print(eps_00)
#
# Define functions for creating extended data in compression and tension
#
def create_extra_compression_data(eps, p, eps_max):
    x0_com = eps.values[1]
    x1_com = eps.values[0]
    y0_com = p.values[1]
    y1_com = p.values[0]
    com_strain = eps_max
    t_com_strain = (com_strain - x0_com)/(x1_com - x0_com)
    dx_com = (x1_com - x0_com)*10
    nx_com = ((com_strain - x1_com)/dx_com).astype(int)
    t_com = np.linspace(1.001, t_com_strain, nx_com)
    x_com = list(map(lambda t : (1 - t)*x0_com + t*x1_com, t_com))
    y_com = list(map(lambda t : (1 - t)*y0_com + t*y1_com, t_com))
    com_data = pd.DataFrame({'TotalStrainVol' : x_com, 'Pressure' : y_com})
    return com_data
#
def create_extra_tension_data(eps, p, eps_min):
    x0_ten = eps.values[-2]
    x1_ten = eps.values[-1]
    y0_ten = p.values[-2]
    y1_ten = p.values[-1]
    ten_strain = eps_min
    t_ten_strain = (ten_strain - x0_ten)/(x1_ten - x0_ten)
    dx_ten = (x0_ten - x1_ten)*10
    nx_ten = ((x1_ten - ten_strain)/dx_ten).astype(int)
    t_ten = np.linspace(1.001, t_ten_strain, nx_ten)
    x_ten = list(map(lambda t : (1 - t)*x0_ten + t*x1_ten, t_ten))
    y_ten = list(map(lambda t : (1 - t)*y0_ten + t*y1_ten, t_ten))
    ten_data = pd.DataFrame({'TotalStrainVol' : x_ten, 'Pressure' : y_ten})
    return ten_data
#
# Create extra data in compression and tension
#
data_00_com_extra = create_extra_compression_data(eps_00, p_00, 15)
data_09_com_extra = create_extra_compression_data(eps_09, p_09, 25)
data_18_com_extra = create_extra_compression_data(eps_18, p_18, 30)
data_27_com_extra = create_extra_compression_data(eps_27, p_27, 35)
#
data_00_ten_extra = create_extra_tension_data(eps_00, p_00, -10)
data_09_ten_extra = create_extra_tension_data(eps_09, p_09, -5)
data_18_ten_extra = create_extra_tension_data(eps_18, p_18, 0)
data_27_ten_extra = create_extra_tension_data(eps_27, p_27, 5)
data_36_ten_extra = create_extra_tension_data(eps_36, p_36, 20)
data_45_ten_extra = create_extra_tension_data(eps_45, p_45, 30)
#
# Save the input unloading data in a data frame
#
```

```
data_00_orig = pd.DataFrame({'TotalStrainVol' : eps_00, 'Pressure' : p_00})
data_09_orig = pd.DataFrame({'TotalStrainVol' : eps_09, 'Pressure' : p_09})
data_18_orig = pd.DataFrame({'TotalStrainVol' : eps_18, 'Pressure' : p_18})
data_27_orig = pd.DataFrame({'TotalStrainVol' : eps_27, 'Pressure' : p_27})
data_36_orig = pd.DataFrame({'TotalStrainVol' : eps_36, 'Pressure' : p_36})
data_45_orig = pd.DataFrame({'TotalStrainVol' : eps_45, 'Pressure' : p_45})
#
# Convert into loading form (increasing compressive strains)
#
data_00_asc = data_00_orig.sort_index(ascending=False)
data_09_asc = data_09_orig.sort_index(ascending=False)
data_18_asc = data_18_orig.sort_index(ascending=False)
data_27_asc = data_27_orig.sort_index(ascending=False)
data_36_asc = data_36_orig.sort_index(ascending=False)
data_45_asc = data_45_orig.sort_index(ascending=False)
#
# Convert the extra tensile data into compressive loading form
#
data_00_ten_asc = data_00_ten_extra.sort_index(ascending=False)
data_09_ten_asc = data_09_ten_extra.sort_index(ascending=False)
data_18_ten_asc = data_18_ten_extra.sort_index(ascending=False)
data_27_ten_asc = data_27_ten_extra.sort_index(ascending=False)
data_36_ten_asc = data_36_ten_extra.sort_index(ascending=False)
data_45_ten_asc = data_45_ten_extra.sort_index(ascending=False)
#
# Merge the data frames together
#
data_00 = data_00_ten_asc.append(data_00_asc, ignore_index = True)
data_00 = data_00.append(data_00_com_extra, ignore_index = True)
data_09 = data_09_ten_asc.append(data_09_asc, ignore_index = True)
data_09 = data_09.append(data_09_com_extra, ignore_index = True)
data_18 = data_18_ten_asc.append(data_18_asc, ignore_index = True)
data_18 = data_18.append(data_18_com_extra, ignore_index = True)
data_27 = data_27_ten_asc.append(data_27_asc, ignore_index = True)
data_27 = data_27.append(data_27_com_extra, ignore_index = True)
data_36 = data_36_ten_asc.append(data_36_asc, ignore_index = True)
data_45 = data_45_ten_asc.append(data_45_asc, ignore_index = True)
#
# Extract back into separate strain and pressure variables
#
eps_00_extra = data_00.iloc[:,0]
eps_09_extra = data_09.iloc[:,0]
eps_18_extra = data_18.iloc[:,0]
eps_27_extra = data_27.iloc[:,0]
eps_36_extra = data_36.iloc[:,0]
eps_45_extra = data_45.iloc[:,0]
p_00_extra = data_00.iloc[:,1]
p_09_extra = data_09.iloc[:,1]
p_18_extra = data_18.iloc[:,1]
p_27_extra = data_27.iloc[:,1]
p_36_extra = data_36.iloc[:,1]
p_45_extra = data_45.iloc[:,1]
#
# Set up strains for training
#
eps_p_00 = 0
strains_00 = np.column_stack((eps_00_extra, np.repeat(eps_p_00, eps_00_extra.shape[0])))
strains_09 = np.column_stack((eps_09_extra, np.repeat(eps_p_09, eps_09_extra.shape[0])))
strains_18 = np.column_stack((eps_18_extra, np.repeat(eps_p_18, eps_18_extra.shape[0])))
strains_27 = np.column_stack((eps_27_extra, np.repeat(eps_p_27, eps_27_extra.shape[0])))
strains_36 = np.column_stack((eps_36_extra, np.repeat(eps_p_36, eps_36_extra.shape[0])))
strains_45 = np.column_stack((eps_45_extra, np.repeat(eps_p_45, eps_45_extra.shape[0])))
#
# Collect strains and pressures together
#
strains = np.concatenate((strains_00, strains_09, strains_18, strains_27, strains_36,
    strains_45), axis=0)
pressures = np.concatenate((p_00_extra, p_09_extra, p_18_extra, p_27_extra, p_36_extra,
    p_45_extra), axis=0)
#
# Do bootstrapped data generation
#
data_00_3d = np.column_stack((strains_00, p_00_extra))
```

```
data_09_3d = np.column_stack((strains_09, p_09_extra))
data_18_3d = np.column_stack((strains_18, p_18_extra))
data_27_3d = np.column_stack((strains_27, p_27_extra))
data_36_3d = np.column_stack((strains_36, p_36_extra))
data_45_3d = np.column_stack((strains_45, p_45_extra))
data_all =  np.concatenate((data_00_3d, data_00_3d, data_00_3d, data_00_3d,
                            data_09_3d, data_09_3d, data_09_3d,
                            data_18_3d, data_18_3d, data_18_3d,
                            data_27_3d, data_27_3d,
                            data_36_3d, data_36_3d,
                            data_45_3d), axis=0)
#
# Shuffle the data
#
np.random.shuffle(data_all)
strains_shuffle = data_all[:,(0,1)]
pressures_shuffle = data_all[:,2]
#
# Separate into train and test sets
#
eps_train, eps_test, p_train, p_test = train_test_split(strains, pressures,
                                                        test_size=0.4, random_state=0)
#
# Set up data scaling functions
#
strain_scaler = preprocessing.MinMaxScaler()
strain_scaler_min_max = strain_scaler.fit(strains)
scaled_strains_train = strain_scaler_min_max.transform(eps_train)
scaled_strains_test = strain_scaler_min_max.transform(eps_test)
scaled_strains = strain_scaler_min_max.transform(strains_shuffle)
#
pressure_scaler = preprocessing.MinMaxScaler()
pressure_scaler_min_max = pressure_scaler.fit(pressures.reshape(-1, 1))
scaled_pressures_train = pressure_scaler_min_max.transform(p_train.reshape(-1, 1))
scaled_pressures_test = pressure_scaler_min_max.transform(p_test.reshape(-1, 1))
scaled_pressures = pressure_scaler_min_max.transform(pressures_shuffle.reshape(-1, 1))
#
# Do SVR fit
#
curve_fitter_1_01 = SVR(kernel='rbf', C=1.0, epsilon=0.01)
fit1_01 = curve_fitter_1_01.fit(scaled_strains_train, np.ravel(scaled_pressures_train))
#
# Compute error norm
#
score1_01 = fit1_01.score(scaled_strains_test, scaled_pressures_test)
#
# Compute cross validation scores to search the parameter space
#
cv = ShuffleSplit(n_splits=10, test_size=0.5, random_state=0)
scores_1_01 = cross_val_score(curve_fitter_1_01, scaled_strains, np.ravel(
    scaled_pressures), cv=cv)
#
# Define function for computing predicted bulk modulus adn plotting
#
def computeAndPlotBulkElastic(svgfile, curve_fitter, C, epsilon):
    s_pressures_pred_00 = curve_fitter.predict(strain_scaler_min_max.transform(
        strains_00))
    s_pressures_pred_09 = curve_fitter.predict(strain_scaler_min_max.transform(
        strains_09))
    s_pressures_pred_18 = curve_fitter.predict(strain_scaler_min_max.transform(
        strains_18))
    s_pressures_pred_27 = curve_fitter.predict(strain_scaler_min_max.transform(
        strains_27))
    s_pressures_pred_36 = curve_fitter.predict(strain_scaler_min_max.transform(
        strains_36))
    s_pressures_pred_45 = curve_fitter.predict(strain_scaler_min_max.transform(
        strains_45))

    pressures_pred_00 = pressure_scaler.inverse_transform(s_pressures_pred_00.reshape
        (-1,1))
    pressures_pred_09 = pressure_scaler.inverse_transform(s_pressures_pred_09.reshape
        (-1,1))
    pressures_pred_18 = pressure_scaler.inverse_transform(s_pressures_pred_18.reshape
```

```
    (-1,1))
pressures_pred_27 = pressure_scaler.inverse_transform(s_pressures_pred_27.reshape
    (-1,1))
pressures_pred_36 = pressure_scaler.inverse_transform(s_pressures_pred_36.reshape
    (-1,1))
pressures_pred_45 = pressure_scaler.inverse_transform(s_pressures_pred_45.reshape
    (-1,1))

# Compute tangent moduli for input data
K_00 = np.gradient(p_00_extra*1.0e6, eps_00_extra*0.01)
K_09 = np.gradient(p_09_extra*1.0e6, eps_09_extra*0.01)
K_18 = np.gradient(p_18_extra*1.0e6, eps_18_extra*0.01)
K_27 = np.gradient(p_27_extra*1.0e6, eps_27_extra*0.01)
K_36 = np.gradient(p_36_extra*1.0e6, eps_36_extra*0.01)
K_45 = np.gradient(p_45_extra*1.0e6, eps_45_extra*0.01)

# Compute tangent moduli for predicted data
#print(pressures_pred_09.shape, strains_09.shape)
K_pred_00 = np.gradient(np.ravel(pressures_pred_00)*1.0e6, strains_00[:,0]*0.01)
K_pred_09 = np.gradient(np.ravel(pressures_pred_09)*1.0e6, strains_09[:,0]*0.01)
K_pred_18 = np.gradient(np.ravel(pressures_pred_18)*1.0e6, strains_18[:,0]*0.01)
K_pred_27 = np.gradient(np.ravel(pressures_pred_27)*1.0e6, strains_27[:,0]*0.01)
K_pred_36 = np.gradient(np.ravel(pressures_pred_36)*1.0e6, strains_36[:,0]*0.01)
K_pred_45 = np.gradient(np.ravel(pressures_pred_45)*1.0e6, strains_45[:,0]*0.01)

# Compute error
K_error_00 = list(map(lambda x, y: 1 if x == 0 else (x - y)/x * 100, K_00, K_pred_00
    ))
K_error_09 = list(map(lambda x, y: 1 if x == 0 else (x - y)/x * 100, K_09, K_pred_09
    ))
K_error_18 = list(map(lambda x, y: 1 if x == 0 else (x - y)/x * 100, K_18, K_pred_18
    ))
K_error_27 = list(map(lambda x, y: 1 if x == 0 else (x - y)/x * 100, K_27, K_pred_27
    ))
K_error_36 = list(map(lambda x, y: 1 if x == 0 else (x - y)/x * 100, K_36, K_pred_36
    ))
K_error_45 = list(map(lambda x, y: 1 if x == 0 else (x - y)/x * 100, K_45, K_pred_45
    ))

plot_label = "SVR (C=" + str(C) + ", $\epsilon$=" + str(epsilon) + ")"
fig = plt.figure(figsize=(14,6))
ax = fig.add_subplot(121)
plt.plot(eps_00_extra - eps_p_00, K_00*1.0e-9, 'C5', label=labels[5])
plt.plot(eps_09_extra - eps_p_09, K_09*1.0e-9, 'k-', label=labels[0])
plt.plot(eps_18_extra - eps_p_18, K_18*1.0e-9, 'C0', label=labels[1])
plt.plot(eps_27_extra - eps_p_27, K_27*1.0e-9, 'C3', label=labels[2])
plt.plot(eps_36_extra - eps_p_36, K_36*1.0e-9, 'C1', label=labels[3])
plt.plot(eps_45_extra - eps_p_45, K_45*1.0e-9, 'C4', label=labels[4])
plt.plot(strains_00[:,0] - eps_p_00, K_pred_00*1.0e-9, 'C5--', label=plot_label,
    linewidth=2)
plt.plot(strains_09[:,0] - eps_p_09, K_pred_09*1.0e-9, 'k--', linewidth=2)
plt.plot(strains_18[:,0] - eps_p_18, K_pred_18*1.0e-9, 'C0--', linewidth=2)
plt.plot(strains_27[:,0] - eps_p_27, K_pred_27*1.0e-9, 'C3--', linewidth=2)
plt.plot(strains_36[:,0] - eps_p_36, K_pred_36*1.0e-9, 'C1--', linewidth=2)
plt.plot(strains_45[:,0] - eps_p_45, K_pred_45*1.0e-9, 'C4--', linewidth=2)

#plt.axis([-10, 45, -10, 50])
plt.axis([0, 10, 0, 50])
plt.xlabel('Elastic volumetric strain (%)', fontsize=16)
plt.ylabel('Tangent buk modulus (GPa)', fontsize=16)
ax.legend(loc='best', fontsize=10)

ax = fig.add_subplot(122)
plt.plot(eps_00_extra - eps_p_00, K_error_00, 'C5', label=labels[5])
plt.plot(eps_09_extra - eps_p_09, K_error_09, 'k', label=labels[0])
plt.plot(eps_18_extra - eps_p_18, K_error_18, 'C0', label=labels[1])
plt.plot(eps_27_extra - eps_p_27, K_error_27, 'C3', label=labels[2])
plt.plot(eps_36_extra - eps_p_36, K_error_36, 'C1', label=labels[3])
plt.plot(eps_45_extra - eps_p_45, K_error_45, 'C4', label=labels[4])
#plt.axis([0, 14, -3, 4])
plt.xlabel('Elastic volumetric strain (%)', fontsize=16)
plt.ylabel('Error in predicted modulus (%)', fontsize=16)
ax.legend(loc='best', fontsize=10)
```

```
    #fig.savefig(svgfile)
#
# Create plot of the fit
#
computeAndPlotBulkElastic('
    Fox_DrySand_BulkModulus_TotalElasticStrain_Tension_SVR_1_01_scaled_cv.svg',
    curve_fitter_1_01, 1, 0.01)
#
# Extract the SVR parameters for the fit
#
print(curve_fitter_1_01.support_.shape, curve_fitter_1_01.support_vectors_.shape,
      curve_fitter_1_01.dual_coef_.shape, curve_fitter_1_01.fit_status_,
      curve_fitter_1_01.intercept_)
#
# Compute gamma
#
gamma = 1/(2*scaled_strains_train.var())
#
# Get the scaling parameters
#
eps_scale = strain_scaler.scale_
eps_min = strain_scaler.data_min_
eps_max = strain_scaler.data_max_
p_scale = pressure_scaler.scale_
p_min = pressure_scaler.data_min_
p_max = pressure_scaler.data_max_
#
# Save as JSON
#
class NumpyEncoder(json.JSONEncoder):
    def default(self, obj):
        if isinstance(obj, np.integer):
            return int(obj)
        elif isinstance(obj, np.floating):
            return float(obj)
        elif isinstance(obj, np.ndarray):
            return obj.tolist()
        else:
            return super(NumpyArrayEncoder, self).default(obj)

# Save the scaling parameters, kernel parameters, support vectors, duals, and intercept
def save_svr_json(json_file, X_var, y_var, X_conv, y_conv, X_scaler, y_scaler,
    X_train_scaled, fitter):
    X_scale = X_scaler.scale_
    X_min = X_scaler.data_min_
    X_max = X_scaler.data_max_
    y_scale = y_scaler.scale_
    y_min = y_scaler.data_min_
    y_max = y_scaler.data_max_
    d = X_train_scaled.shape[1]
    sigma_sq = X_train_scaled.var()
    gamma = 1/(d*sigma_sq)
    X_sup_vec = fitter.support_vectors_
    X_dual = fitter.dual_coef_
    X_intercept = fitter.intercept_
    json_data = {}
    json_data['X_var'] = X_var
    json_data['y_var'] = y_var
    json_data['X_conversion_factor'] = X_conv
    json_data['y_conversion_factor'] = y_conv
    json_data['X_scale'] = X_scale
    json_data['X_min'] = X_min
    json_data['X_max'] = X_max
    json_data['y_scale'] = y_scale
    json_data['y_min'] = y_min
    json_data['y_max'] = y_max
    json_data['gamma'] = gamma
    json_data['support_vectors'] = X_sup_vec
    json_data['dual_coeffs'] = X_dual
    json_data['intercept'] = X_intercept
    # Set numpy's printoptions to display all the data with max precision
    np.set_printoptions(threshold=np.inf,
```

```
                        linewidth=sys.maxsize,
                        suppress=True,
                        nanstr='0.0',
                        infstr='0.0',
                        precision=np.finfo(np.longdouble).precision)
    with open(json_file, 'w') as outfile:
        json.dump(json_data, outfile, cls=NumpyEncoder, indent=2)
#
save_svr_json('ARL_Sand_SVR_fit_10_001.json',
              ['Total volumetric strain (%)', 'Plastic volumetric strain (%)'],
              pressure_scaler, scaled_strains_train, curve_fitter_10_001)
```

5.3.2 Neural-network model for the bulk modulus

Instead of a tabular or a support vector model, the tabular plasticity model can also use a neural network model. This model can be invoked using:

```
<constitutive_model type="tabular_plasticity_cap">
  <elastic_moduli_model type="neural_net_bulk">
    <filename>mlp_regression.h5</filename>
    <mean_elastic_strain>0.0</mean_elastic_strain>
    <mean_plastic_strain>0.5</mean_plastic_strain>
    <std_dev_elastic_strain>1.0</std_dev_elastic_strain>
    <std_dev_plastic_strain>0.5</std_dev_plastic_strain>
    <mean_bulk_modulus>0.0</mean_bulk_modulus>
    <std_dev_bulk_modulus>1.0e6</std_dev_bulk_modulus>
    <G0>3500</G0>
    <nu>0.189</nu>
  </elastic_moduli_model>
  ...
</constitutive_model>
```

The input file is required to contains a HDF5 representation of the fitted neural network produced by Tensorflow 2.0. The mean and standard deviations are required as inputs with the assumption that the input data were fitted after scaling the data such that the mean was zero and the standard deviation 1.

An example of fitting a neural network is given below.

```
# Load the required packages
import numpy as np
import pandas as pd
import PolylineIntersection as pl
from keras.models import Sequential
from keras.layers import Dense
from keras.utils import plot_model
import matplotlib.pyplot as plt
from sklearn import preprocessing
from keras.models import load_model
import h5py

# Load the CSV data into Pandas dataframes
hydrostat = pd.read_csv("./DrySand_Hydrostat.csv", header=0, skiprows=7)
data_09 = pd.read_csv("./DrySand_LoadUnload_09.csv", header=0, skiprows=4)
data_18 = pd.read_csv("./DrySand_LoadUnload_18.csv", header=0, skiprows=4)
data_27 = pd.read_csv("./DrySand_LoadUnload_27.csv", header=0, skiprows=4)
data_36 = pd.read_csv("./DrySand_LoadUnload_36.csv", header=0, skiprows=4)
data_45 = pd.read_csv("./DrySand_LoadUnload_45.csv", header=0, skiprows=4)

# Rename the columns of each dataframe
column_names = ["TotalStrainVol", "Pressure", "", "", "", ""]
hydrostat.columns = column_names
data_09.columns = column_names
data_18.columns = column_names
data_27.columns = column_names
data_36.columns = column_names
data_45.columns = column_names

# Convert percent into strain, MPa to Pa
strain_fac = 0.01
pressure_fac = 1.0e6
```

```
hydrostat.TotalStrainVol *= strain_fac
hydrostat.Pressure *= pressure_fac
data_09.TotalStrainVol *= strain_fac
data_09.Pressure *= pressure_fac
data_18.TotalStrainVol *= strain_fac
data_18.Pressure *= pressure_fac
data_27.TotalStrainVol *= strain_fac
data_27.Pressure *= pressure_fac
data_36.TotalStrainVol *= strain_fac
data_36.Pressure *= pressure_fac
data_45.TotalStrainVol *= strain_fac
data_45.Pressure *= pressure_fac

# Find the point at which unloading begins
p_max_09 = max(data_09.Pressure)
p_max_index_09 = data_09.Pressure.values.tolist().index(p_max_09)
p_max_18 = max(data_18.Pressure)
p_max_index_18 = data_18.Pressure.values.tolist().index(p_max_18)
p_max_27 = max(data_27.Pressure)
p_max_index_27 = data_27.Pressure.values.tolist().index(p_max_27)
p_max_36 = max(data_36.Pressure)
p_max_index_36 = data_36.Pressure.values.tolist().index(p_max_36)
p_max_45 = max(data_45.Pressure)
p_max_index_45 = data_45.Pressure.values.tolist().index(p_max_45)
p_max_index_00 = (np.abs(p_max_09 - hydrostat.Pressure.values)).argmin()
p_max_00 = hydrostat.Pressure.values[p_max_index_00]

# Create separate dataframes for the unload data
data_09_unload = data_09[p_max_index_09:].copy()
data_18_unload = data_18[p_max_index_18:].copy()
data_27_unload = data_27[p_max_index_27:].copy()
data_36_unload = data_36[p_max_index_36:].copy()
data_45_unload = data_45[p_max_index_45:].copy()

# Find plastic strains by intersecting the unload data with the pressure axis
pressure_axis = ((-1, 0),(1, 0))
poly_09_unload = list(data_09_unload[['TotalStrainVol', 'Pressure']].apply(tuple, axis
    =1))
line_09_unload = (poly_09_unload[-1], poly_09_unload[-2])
plastic_strain_09 = pl.line_intersection(pressure_axis, line_09_unload)[0]
poly_18_unload = list(data_18_unload[['TotalStrainVol', 'Pressure']].apply(tuple, axis
    =1))
line_18_unload = (poly_18_unload[-1], poly_18_unload[-2])
plastic_strain_18 = pl.line_intersection(pressure_axis, line_18_unload)[0]
poly_27_unload = list(data_27_unload[['TotalStrainVol', 'Pressure']].apply(tuple, axis
    =1))
line_27_unload = (poly_27_unload[-1], poly_27_unload[-2])
plastic_strain_27 = pl.line_intersection(pressure_axis, line_27_unload)[0]
poly_36_unload = list(data_36_unload[['TotalStrainVol', 'Pressure']].apply(tuple, axis
    =1))
line_36_unload = (poly_36_unload[-1], poly_36_unload[-2])
plastic_strain_36 = pl.line_intersection(pressure_axis, line_36_unload)[0]
poly_45_unload = list(data_45_unload[['TotalStrainVol', 'Pressure']].apply(tuple, axis
    =1))
line_45_unload = (poly_45_unload[-1], poly_45_unload[-2])
plastic_strain_45 = pl.line_intersection(pressure_axis, line_45_unload)[0]
print(plastic_strain_09, plastic_strain_18, plastic_strain_27, plastic_strain_36,
    plastic_strain_45)

# Reverse the order of the unload data to create elastic loading curves
data_00_load = hydrostat[:p_max_index_00]
data_09_load = data_09_unload.sort_index(ascending=False)
data_18_load = data_18_unload.sort_index(ascending=False)
data_27_load = data_27_unload.sort_index(ascending=False)
data_36_load = data_36_unload.sort_index(ascending=False)
data_45_load = data_45_unload.sort_index(ascending=False)

# Simplify the loading dataframes and remove duplicates (if any)
data_00_all = data_00_load[['TotalStrainVol', 'Pressure']].copy().drop_duplicates()
data_09_all = data_09_load[['TotalStrainVol', 'Pressure']].copy().drop_duplicates()
data_18_all = data_18_load[['TotalStrainVol', 'Pressure']].copy().drop_duplicates()
data_27_all = data_27_load[['TotalStrainVol', 'Pressure']].copy().drop_duplicates()
data_36_all = data_36_load[['TotalStrainVol', 'Pressure']].copy().drop_duplicates()
```

```
data_45_all = data_45_load[['TotalStrainVol', 'Pressure']].copy().drop_duplicates()

# Compute max strain and pressure
total_strain_max = data_45_all['TotalStrainVol'].max()
pressure_max = data_45_all['Pressure'].max()

# Rename strain variables
eps_p_09 = plastic_strain_09
eps_09 = data_09_all['TotalStrainVol'].values
eps_p_18 = plastic_strain_18
eps_18 = data_18_all['TotalStrainVol'].values
eps_p_27 = plastic_strain_27
eps_27 = data_27_all['TotalStrainVol'].values
eps_p_36 = plastic_strain_36
eps_36 = data_36_all['TotalStrainVol'].values
eps_p_45 = plastic_strain_45
eps_45 = data_45_all['TotalStrainVol'].values
eps_00 = np.linspace(0, total_strain_max)
eps_30 = np.linspace(0, total_strain_max)
eps_p_00 = 0
eps_p_30 = 0.5*(eps_p_27 + eps_p_36)

# Rename pressure variables
p_09 = data_09_all['Pressure'].values
p_18 = data_18_all['Pressure'].values
p_27 = data_27_all['Pressure'].values
p_36 = data_36_all['Pressure'].values
p_45 = data_45_all['Pressure'].values

# Scale the data
def scaled(x, min_x, max_x):
    return (x - min_x)/(max_x - min_x)

# Unscale the data
def unscaled(x, min_x, max_x):
    return min_x + x * (max_x - min_x)

# Convert strains to scaled values
eps_09_scaled = scaled(eps_09, 0, total_strain_max)
eps_18_scaled = scaled(eps_18, 0, total_strain_max)
eps_27_scaled = scaled(eps_27, 0, total_strain_max)
eps_36_scaled = scaled(eps_36, 0, total_strain_max)
eps_45_scaled = scaled(eps_45, 0, total_strain_max)
eps_00_scaled = scaled(eps_00, 0, total_strain_max)
eps_30_scaled = scaled(eps_30, 0, total_strain_max)
eps_p_09_scaled = scaled(eps_p_09, 0, total_strain_max)
eps_p_18_scaled = scaled(eps_p_18, 0, total_strain_max)
eps_p_27_scaled = scaled(eps_p_27, 0, total_strain_max)
eps_p_36_scaled = scaled(eps_p_36, 0, total_strain_max)
eps_p_45_scaled = scaled(eps_p_45, 0, total_strain_max)
eps_p_00_scaled = scaled(eps_p_00, 0, total_strain_max)
eps_p_30_scaled = scaled(eps_p_30, 0, total_strain_max)

# Convert pressures to scaled values
p_09_scaled = scaled(p_09, 0, 1.0e6)
p_18_scaled = scaled(p_18, 0, 1.0e6)
p_27_scaled = scaled(p_27, 0, 1.0e6)
p_36_scaled = scaled(p_36, 0, 1.0e6)
p_45_scaled = scaled(p_45, 0, 1.0e6)

# Set up data frames for regression
strains_09_scaled = np.column_stack((eps_09_scaled, np.repeat(eps_p_09_scaled,
    eps_09_scaled.shape[0])))
strains_18_scaled = np.column_stack((eps_18_scaled, np.repeat(eps_p_18_scaled,
    eps_18_scaled.shape[0])))
strains_27_scaled = np.column_stack((eps_27_scaled, np.repeat(eps_p_27_scaled,
    eps_27_scaled.shape[0])))
strains_36_scaled = np.column_stack((eps_36_scaled, np.repeat(eps_p_36_scaled,
    eps_36_scaled.shape[0])))
strains_45_scaled = np.column_stack((eps_45_scaled, np.repeat(eps_p_45_scaled,
    eps_45_scaled.shape[0])))
strains_30_scaled = np.column_stack((eps_30_scaled, np.repeat(eps_p_30_scaled,
    eps_30_scaled.shape[0])))
```

```
strains_00_scaled = np.column_stack((eps_00_scaled, np.repeat(eps_p_00_scaled,
    eps_00_scaled.shape[0])))
strains_scaled = np.concatenate((strains_09_scaled, strains_18_scaled, strains_27_scaled
    , strains_36_scaled, strains_45_scaled), axis=0)
pressures_scaled = np.concatenate((p_09_scaled, p_18_scaled, p_27_scaled, p_36_scaled,
    p_45_scaled), axis=0)

data_09_scaled = np.column_stack((strains_09_scaled, p_09_scaled))
data_18_scaled = np.column_stack((strains_18_scaled, p_18_scaled))
data_27_scaled = np.column_stack((strains_27_scaled, p_27_scaled))
data_36_scaled = np.column_stack((strains_36_scaled, p_36_scaled))
data_45_scaled = np.column_stack((strains_45_scaled, p_45_scaled))

# Bootstrapping step
data_all_scaled =  np.concatenate((data_09_scaled, data_18_scaled, data_27_scaled,
    data_36_scaled, data_45_scaled,
                            data_09_scaled, data_09_scaled, data_09_scaled,
                                data_09_scaled,
                            data_18_scaled, data_18_scaled, data_18_scaled,
                                data_18_scaled,
                            data_27_scaled, data_27_scaled, data_27_scaled,
                                data_27_scaled,
                            data_36_scaled, data_36_scaled, data_36_scaled,
                                data_36_scaled,
                            data_45_scaled, data_45_scaled, data_45_scaled,
                                data_45_scaled), axis=0)
#np.random.seed(12345)
np.random.shuffle(data_all_scaled)
strains_shuffle = data_all_scaled[:,(0,1)]
pressures_shuffle = data_all_scaled[:,2]

# Set up the neural network
def baseline2D_model():
    model = Sequential()
    model.add(Dense(64, input_dim=2, kernel_initializer='normal', activation='sigmoid'))
    model.add(Dense(32, kernel_initializer='normal', activation='sigmoid'))
    model.add(Dense(32, kernel_initializer='normal', activation='relu'))
    model.add(Dense(1, kernel_initializer='normal'))
    model.compile(loss='mean_squared_error', optimizer='adam')
    return model

model2D = baseline2D_model()

# Fit the model
model2D.fit(strains_shuffle, pressures_shuffle, batch_size=192, epochs=3000, verbose=1,
    validation_split=0.2, shuffle=True)

# Compute predicted values
pressures_pred_09_scaled = model2D.predict(strains_09_scaled)
pressures_pred_18_scaled = model2D.predict(strains_18_scaled)
pressures_pred_27_scaled = model2D.predict(strains_27_scaled)
pressures_pred_36_scaled = model2D.predict(strains_36_scaled)
pressures_pred_45_scaled = model2D.predict(strains_45_scaled)
pressures_pred_30_scaled = model2D.predict(strains_30_scaled)
pressures_pred_00_scaled = model2D.predict(strains_00_scaled)

# Unscale the data
strains_09 = unscaled(strains_09_scaled, 0, total_strain_max)
strains_18 = unscaled(strains_18_scaled, 0, total_strain_max)
strains_27 = unscaled(strains_27_scaled, 0, total_strain_max)
strains_36 = unscaled(strains_36_scaled, 0, total_strain_max)
strains_45 = unscaled(strains_45_scaled, 0, total_strain_max)
strains_30 = unscaled(strains_30_scaled, 0, total_strain_max)
strains_00 = unscaled(strains_00_scaled, 0, total_strain_max)

pressure_max = 1.0e6
pressures_pred_09 = unscaled(pressures_pred_09_scaled, 0, pressure_max)
pressures_pred_18 = unscaled(pressures_pred_18_scaled, 0, pressure_max)
pressures_pred_27 = unscaled(pressures_pred_27_scaled, 0, pressure_max)
pressures_pred_36 = unscaled(pressures_pred_36_scaled, 0, pressure_max)
pressures_pred_45 = unscaled(pressures_pred_45_scaled, 0, pressure_max)
pressures_pred_30 = unscaled(pressures_pred_30_scaled, 0, pressure_max)
pressures_pred_00 = unscaled(pressures_pred_00_scaled, 0, pressure_max)
```

```
# Set up rescale function for plotting
def rescale(data, VARIABLE_TYPE):
    result = {
        "e" : lambda data : data*100.0,
        "p" : lambda data : data*1.0e-6
    }[VARIABLE_TYPE](data)
    return result

# Plot the original vs predicted data
lab_09 = str("Plastic strain = {0:.3f}".format(eps_p_09))
lab_18 = str("Plastic strain = {0:.3f}".format(eps_p_18))
lab_27 = str("Plastic strain = {0:.3f}".format(eps_p_27))
lab_36 = str("Plastic strain = {0:.3f}".format(eps_p_36))
lab_45 = str("Plastic strain = {0:.3f}".format(eps_p_45))
lab_00 = str("MLP: Plastic strain = {0:.3f}".format(eps_p_00))
lab_30 = str("MLP: Plastic strain = {0:.3f}".format(eps_p_30))
type(lab_09)

fig = plt.figure(figsize=(6,6))
ax = fig.add_subplot(111)
plt.plot(rescale(eps_09, "e"), rescale(p_09, "p"), 'k--', label=lab_09)
plt.plot(rescale(eps_18, "e"), rescale(p_18, "p"), 'C0--', label=lab_18)
plt.plot(rescale(eps_27, "e"), rescale(p_27, "p"), 'C3--', label=lab_27)
plt.plot(rescale(eps_36, "e"), rescale(p_36, "p"), 'C1--', label=lab_36)
plt.plot(rescale(eps_45, "e"), rescale(p_45, "p"), 'C4--', label=lab_45)
plt.plot(rescale(strains_09[:,0], "e"), rescale(pressures_pred_09, "p"), 'k', linewidth
    =2, label='MLP')
plt.plot(rescale(strains_18[:,0], "e"), rescale(pressures_pred_18, "p"), 'C0', linewidth
    =2)
plt.plot(rescale(strains_27[:,0], "e"), rescale(pressures_pred_27, "p"), 'C3', linewidth
    =2)
plt.plot(rescale(strains_36[:,0], "e"), rescale(pressures_pred_36, "p"), 'C1', linewidth
    =2)
plt.plot(rescale(strains_45[:,0], "e"), rescale(pressures_pred_45, "p"), 'C4', linewidth
    =2)
plt.plot(rescale(strains_30[:,0], "e"), rescale(pressures_pred_30, "p"), 'C7', linewidth
    =2, label=lab_30)
plt.plot(rescale(strains_00[:,0], "e"), rescale(pressures_pred_00, "p"), 'C8', linewidth
    =2, label=lab_00)
plt.axis([0, 50, 0, 3000])
plt.xlabel('Total volumetric strain (%)', fontsize=16)
plt.ylabel('Pressure (MPa)', fontsize=16)
ax.legend(loc='best', fontsize=14)
fig.savefig('Fox_DrySand_ElasticStrain_MLP_Total.svg')

# Save the model and fit in binary HDF5 format for reading into Vaango
model2D.save('mlp_regression_keras_total_scaled.h5')
```

5.4 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.

Example input file for the HypoElasticPlastic 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.

5.4.1 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 [8], 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 ($\mathbf{d}$) is additively decomposed into an elastic part, $\mathbf{d}^e$, and a plastic part, $\mathbf{d}^p$,

$$\mathbf{d} = \mathbf{d}^e + \mathbf{d}^p \tag{5.8}$$

It is convenient to work with the deviatoric parts of $\mathbf{d}$, $\mathbf{d}^e$, and $\mathbf{d}^p$, denoted $\boldsymbol{\eta}$, $\boldsymbol{\eta}^e$, and $\boldsymbol{\eta}^p$ respectively. The same additive decomposition obtains

$$\boldsymbol{\eta} = \boldsymbol{\eta}^e + \boldsymbol{\eta}^p \tag{5.9}$$

Presently these models are limited to the case of plastic incompressibility ($\mathrm{tr}(\mathbf{d}^p) = 0$), so that $\mathbf{d}^p = \boldsymbol{\eta}^p$.

The radial return algorithm assumes a yield condition of the form

$$f(\mathbf{s}, \varepsilon_p^{\mathrm{eq}}, \dot{\varepsilon}_p^{\mathrm{eq}}) = \sqrt{3J_2} - \sigma_y(\varepsilon_p^{\mathrm{eq}}, \dot{\varepsilon}_p^{\mathrm{eq}}) \tag{5.10}$$

where σ_y is the flow stress, $J_2 = \frac{1}{2}\mathbf{s} : \mathbf{s}$ is the second invariant of the deviatoric part of the Cauchy stress, $\mathbf{s}$, and the equivalent plastic strain is defined as

$$\varepsilon_p^{\mathrm{eq}} = \int_0^t \sqrt{\tfrac{2}{3}\mathbf{d}^p : \mathbf{d}^p}\, dt = \int_0^t \sqrt{\tfrac{2}{3}\boldsymbol{\eta}^p : \boldsymbol{\eta}^p}\, dt \tag{5.11}$$

Assuming a state at the end of the previous time step, time t^n, satisfying $f(\mathbf{s}^n, (\varepsilon_p^{\mathrm{eq}})^n, \dot{\varepsilon}_p^n) \le 0$, a new state satisfying Eqn. 5.10 at time $t^{n+1} = t^n + \Delta t$, where Δt is the time step size, is sought.

Attention is further restricted to plastic flow associated with the yield condition, Eqn. 5.10, i.e.

$$\mathbf{d}^p = \boldsymbol{\eta}^p \propto \frac{\partial f}{\partial \boldsymbol{\sigma}} = \dot{\lambda}\frac{\mathbf{s}}{\|\mathbf{s}\|} = \dot{\lambda}\mathbf{n} \tag{5.12}$$

where $\boldsymbol{\sigma}$ is the Cauchy stress , $\mathbf{n} = \mathbf{s}/\|\mathbf{s}\|$ 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

$$\dot{\mathbf{s}} = 2\mu(\boldsymbol{\eta} - \boldsymbol{\eta}^p) \tag{5.13}$$

where μ 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.

$$\mathbf{s}^{\mathrm{trial}} = \mathbf{s}^n + 2\mu\boldsymbol{\eta}\Delta t \tag{5.14}$$

If $f(\mathbf{s}^{\text{trial}}, (\varepsilon_p^{\text{eq}})^n, 0) \leq 0$, the deformation is purely elastic and the solution at time t^{n+1} is $\mathbf{s}^{n+1} = \mathbf{s}^{\text{trial}}$, $(\varepsilon_p^{\text{eq}})^{n+1} = (\varepsilon_p^{\text{eq}})^n$. If $f(\mathbf{s}^{\text{trial}}, (\varepsilon_p^{\text{eq}})^n, 0) > 0$, the deformation is at least partially plastic. In this case

$$\mathbf{s}^{n+1} = \mathbf{s}^n + \dot{\mathbf{s}}\Delta t \tag{5.15}$$

where $\dot{\mathbf{s}}$ is given by Eqn. 5.13. Eqn. 5.15 may be rewritten in terms of the trial stress

$$\mathbf{s}^{n+1} = \mathbf{s}^{\text{trial}} - 2\mu\boldsymbol{\eta}^p\Delta t = \mathbf{s}^{\text{trial}} - 2\mu\dot{\lambda}\Delta t\frac{\mathbf{s}^{n+1}}{\|\mathbf{s}^{n+1}\|} \tag{5.16}$$

using Eqn.s 5.14, 5.13 and 5.12. This equation may be rearranged to give

$$\mathbf{s}^{\text{trial}} = \mathbf{s}^{n+1}\left[1 + \frac{2\mu\dot{\lambda}\Delta t}{\|\mathbf{s}^{n+1}\|}\right] \tag{5.17}$$

which gives the key result that $\mathbf{s}^{\text{trial}} \propto \mathbf{s}^{n+1}$, i.e. the trial stress and the updated stress are in the same direction. Consequently the flow direction may be written

$$\mathbf{n} = \frac{\mathbf{s}^{n+1}}{\|\mathbf{s}^{n+1}\|} = \frac{\mathbf{s}^{\text{trial}}}{\|\mathbf{s}^{\text{trial}}\|} \tag{5.18}$$

and Eqn. 5.16 may be rewritten

$$\mathbf{s}^{n+1} = \mathbf{s}^{\text{trial}} - 2\mu\dot{\lambda}\Delta t\mathbf{n} \tag{5.19}$$

or, contracting both sides with $\mathbf{n}$, and using Eqn. 5.18,

$$\|\mathbf{s}^{n+1}\| = \|\mathbf{s}^{\text{trial}}\| - 2\mu\dot{\lambda}\Delta t \tag{5.20}$$

which is a scalar equation for the proportionality constant $\dot{\lambda}$. Using the yield condition $f(\mathbf{s}^{n+1}, (\varepsilon_p^{\text{eq}})^{n+1}, \dot{\varepsilon}_p^{n+1}) = 0$ (Eqn. 5.10), this equation may be written in terms of $\dot{\lambda}$

$$\sqrt{\tfrac{2}{3}}\sigma_y((\varepsilon_p^{\text{eq}})^{n+1}, \dot{\varepsilon}_p^{n+1}) = \|\mathbf{s}^{\text{trial}}\| - 2\mu\dot{\lambda}\Delta t \tag{5.21}$$

where from Eqn.s 5.11 and 5.12, $(\varepsilon_p^{\text{eq}})^{n+1} = (\varepsilon_p^{\text{eq}})^n + \sqrt{\frac{2}{3}}\dot{\lambda}\Delta t$ and $\dot{\varepsilon}_p^{\text{eq}^{n+1}} = \sqrt{\frac{2}{3}}\dot{\lambda}$.

For the special case of linear isotropic hardening, $\sigma_y(\varepsilon_p^{\text{eq}}, \dot{\varepsilon}_p) = \sigma_{y0} + k\varepsilon_p^{\text{eq}}$, where σ_{y0} is the initial yield stress and k is the hardening modulus, Eqn. 5.21 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

$$g(\Delta\lambda_j) = \|\mathbf{s}^{\text{trial}}\| - 2\mu\Delta\lambda_j - \sqrt{\tfrac{2}{3}}\sigma_y(\epsilon_{p,j}^{n+1}, \dot{\epsilon}_{p,j}^{n+1}) \tag{5.22}$$

and, using the chain rule, the derivative may be calculated

$$\frac{dg}{d\Delta\lambda}(\Delta\lambda_j) = -2\mu - \frac{2}{3}\left[\frac{\partial\sigma_y}{\partial\varepsilon_p^{\text{eq}}}(\epsilon_{p,j}^{n+1}, \dot{\epsilon}_{p,j}^{n+1}) + \frac{\partial\sigma_y}{\partial\dot{\epsilon}_{p,j}^{n+1}}(\epsilon_{p,j}^{n+1}, \dot{\epsilon}_{p,j}^{n+1})\frac{1}{\Delta t}\right] \tag{5.23}$$

Then until $|g(\Delta\lambda_{j+1})| < \text{TOL}$, calculate

$$\Delta\lambda_{j+1} = \Delta\lambda_j - \frac{g(\Delta\lambda_j)}{\frac{dg}{d\Delta\lambda}(\Delta\lambda_j)} \tag{5.24}$$

where $\Delta\lambda_0 = 0$, $\epsilon_{p,0}^{n+1} = (\varepsilon_p^{\text{eq}})^n$, $\dot{\epsilon}_{p,0}^{n+1} = 0$, and, once $\Delta\lambda$ has been determined to a specified accuracy, the final values of plastic strain and strain rate are given by

$$(\varepsilon_p^{\text{eq}})^{n+1} = (\varepsilon_p^{\text{eq}})^n + \sqrt{\tfrac{2}{3}}\Delta\lambda_{j+1} \tag{5.25}$$

$$\dot{\varepsilon_p^{\text{eq}}}^{n+1} = \sqrt{\frac{2}{3}}\frac{\Delta\lambda_{j+1}}{\Delta t} \tag{5.26}$$

and the final value of $\mathbf{s}^{n+1}$ is calculated from Eqn. 5.19.

While several of the allowed flow stresses are of the form $\sigma_y(\varepsilon_p^{\text{eq}}, \dot{\varepsilon_p^{\text{eq}}})$ as given in Eqn. 5.10, 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. 5.12 (which is non-associated because the pressure dependence of the flow stress has been neglected, i.e. Eqn. 5.12 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 $\sigma_y((\varepsilon_p^{\text{eq}})^{n+1}, \dot{\varepsilon_p^{\text{eq}}}^{n+1}, p^n, T^n)$ 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.

$$\dot{\mathbf{s}} = 2\mu(\boldsymbol{\eta} - \boldsymbol{\eta}^p) - \sum_{i=1}^{N}\frac{\mathbf{s}_i}{\tau_i} \tag{5.27}$$

rather than Eqn. 5.13. Eqn. 5.27 is the constitutive equation for N linear Maxwell elements in parallel, each with shear modulus μ_i, time constant τ_i, and deviatoric stress $\mathbf{s}_i$, and

$$\mathbf{s} = \sum_{i=1}^{N}\mathbf{s}_i \qquad \mu = \sum_{i=1}^{N}\mu_i \tag{5.28}$$

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) [9, 10], Wang (1994) [11], Maudlin (1996) [12], and Zocher et al. (2000) [13]. 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!!!

5.4.2 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).

5.4.3 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 (ΔT) due to an increment in plastic strain ($\Delta\epsilon_p$) is given by the equation

$$\Delta T = \frac{\chi\sigma_y}{\rho C_p}\Delta\epsilon_p \tag{5.29}$$

where χ 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 [14] for more details on the variation of χ 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 [15]:

$$C_v = \frac{\tilde{T}^3}{c_3\tilde{T}^3 + c_2\tilde{T}^2 + c_1\tilde{T} + c_0} \tag{5.30}$$

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:

$$\theta(V) = \theta_0\left(\frac{V_0}{V}\right)^a e^{b(V_0-V)/V} \tag{5.31}$$

where V is the specific volume and θ_0 is the reference Debye temperature and a and b are fit parameters. The constant pressure specific heat is calculated via:

$$C_p = C_v + \beta^2 T V K_T \tag{5.32}$$

where β 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 β, a and b are required.

Specific heat model for copper

The specific heat model for copper is of the form

$$C_p = \begin{cases} A_0\, T^3 - B_0\, T^2 + C_0\, T - D_0 & \text{if } T < T_0 \\ A_1\, T + B_1 & \text{if } T \geq T_0\,. \end{cases} \tag{5.33}$$

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 ([16]).

$$C_p = \begin{cases} A_1 + B_1\, t + C_1\, |t|^{-\alpha} & \text{if } T < T_c \\ A_2 + B_2\, t + C_2\, t^{-\alpha'} & \text{if } T > T_c \end{cases} \tag{5.34}$$

$$t = \frac{T}{T_c} - 1 \tag{5.35}$$

where T_c is the critical temperature at which the phase transformation from the α to the γ 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.

5.4.4 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

$$\dot{\boldsymbol{\sigma}} = \lambda \, \mathrm{tr}(\mathbf{d}^e) \, \boldsymbol{1} + 2 \, \mu \, \mathbf{d}^e \tag{5.36}$$

where $\boldsymbol{\sigma}$ is the Cauchy stress, $\mathbf{d}^e$ is the elastic part of the rate of deformation, and λ, μ are constants.

If $\boldsymbol{\eta}^e$ is the deviatoric part of $\mathbf{d}^e$ then we can write

$$\dot{\boldsymbol{\sigma}} = \left(\lambda + \frac{2}{3}\,\mu\right) \mathrm{tr}(\mathbf{d}^e) \, \boldsymbol{1} + 2 \, \mu \, \boldsymbol{\eta}^e = \kappa \, \mathrm{tr}(\mathbf{d}^e) \, \boldsymbol{1} + 2 \, \mu \, \boldsymbol{\eta}^e \, . \tag{5.37}$$

If we split $\boldsymbol{\sigma}$ into a volumetric and a deviatoric part, i.e., $\boldsymbol{\sigma} = p \, \boldsymbol{1} + \mathbf{s}$ and take the time derivative to get $\dot{\boldsymbol{\sigma}} = \dot{p} \, \boldsymbol{1} + \dot{\mathbf{s}}$ then

$$\dot{p} = \kappa \, \mathrm{tr}(\mathbf{d}^e) \, . \tag{5.38}$$

In addition we assume that $\mathbf{d} = \mathbf{d}^e + \mathbf{d}^p$. If we also assume that the plastic volume change is negligible, we can then write that

$$\dot{p} = \kappa \, \mathrm{tr}(\mathbf{d}) \, . \tag{5.39}$$

This is the equation that is used to calculate the pressure p in the default hypoelastic equation of state, i.e.,

$$\boxed{p_{n+1} = p_n + \kappa \, \mathrm{tr}(\mathbf{d}_{n+1}) \, \Delta t \, .} \tag{5.40}$$

To get the derivative of p with respect to J, where $J = \det(\boldsymbol{F})$, we note that

$$\dot{p} = \frac{\partial p}{\partial J} \, \dot{J} = \frac{\partial p}{\partial J} \, J \, \mathrm{tr}(\mathbf{d}) \, . \tag{5.41}$$

Therefore,

$$\boxed{\frac{\partial p}{\partial J} = \frac{\kappa}{J} \, .} \tag{5.42}$$

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

$$p = \tfrac{1}{2}\,\kappa\left(J^e - \frac{1}{J^e}\right) \tag{5.43}$$

where κ is the bulk modulus and J^e is determinant of the elastic part of the deformation gradient.

We can also compute

$$\frac{dp}{dJ} = \tfrac{1}{2}\,\kappa\left(1 + \frac{1}{(J^e)^2}\right). \tag{5.44}$$

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

$$p = p_{ref} + \rho\Gamma(e - e_{ref}) \tag{5.45}$$

where ρ is the mass density, Γ 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,

$$U_s = C_0 + s_\alpha U_p + s_2\frac{U_p^2}{U_s} + s_3\frac{U_p^3}{U_s^2} \tag{5.46}$$

where C_0 is the bulk speed of sound, and the s's are dimensionless coefficients. This form is due to Steinberg ([17]), 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. ([13, 18]), 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

$$p_H = \frac{\rho_0 C_0^2\eta}{1 - s_\alpha\eta - s_2\eta^2 - s_3\eta^3} \tag{5.47}$$

$$e_H = \frac{p_H\eta}{2\rho_0} \tag{5.48}$$

where ρ_0 is the initial density (pre-shock) and

$$\eta = 1 - \frac{\rho_0}{\rho} \tag{5.49}$$

is a measure of volumetric deformation. Using these relationships and the additional assumption that $\rho\Gamma = \rho_o\Gamma_o$ the Mie-Grüneisen equation of state may be written

$$\boxed{p = \frac{\rho_o\, C_o^2\, \eta\left(1 - \frac{\Gamma_o \eta}{2}\right)}{(1 - s_\alpha \eta - s_2\eta^2 - s_3\eta^3)^2} + \rho_o\Gamma_o\, e} \tag{5.50}$$

To extend the Mie Grüneisen EOS into tensile stress regimes, for $\eta < o$ the pressure is evaluated as

$$\boxed{p = \rho_o\, C_o^2\, \eta + \rho_o\Gamma_o\, e} \tag{5.51}$$

This equation is integrated explicitly, using beginning timestep values for energy and the current value of density to update the pressure. For isochoric plasticity,

$$J^e = J = \det(\boldsymbol{F}) = \frac{\rho_o}{\rho}.$$

where $\boldsymbol{F}^e$ is the elastic part of the deformation gradient. The increment in specific internal energy is computed using

$$\rho^* \Delta e = (\sigma_{ij}^* D_{ij} - q D_{kk})\Delta t \tag{5.52}$$

where σ_{ij}^* and ρ^* 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

$$dT = -\rho\Gamma T dv + \frac{TdS}{C_v} \tag{5.53}$$

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

$$\Delta T_{isentropic} = -T\Gamma_o \frac{\rho_o}{\rho} D_{kk}\Delta t \tag{5.54}$$

where the same assumption, $\rho\Gamma = \rho_o\Gamma_o$ 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

$$\boxed{\frac{\partial p}{\partial J^e} = \frac{\rho_o\, C_o^2 \left[1 + (S_\alpha - \Gamma_o)\,(1 - J^e)\right]}{\left[1 - S_\alpha\,(1 - J^e)\right]^3} - \Gamma_o\, \frac{\partial e}{\partial J^e}.} \tag{5.55}$$

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 ([18]). Wilkins expands the pressure and energy as polynomials in η and uses Hugoniot data (and a linear U_s, U_p relationship) to determine the coefficients. Using the additional thermodynamic relationship

$$de = -pdv + TdS \tag{5.56}$$

and substituting for TdS from 5.53, assuming C_v constant and $\rho\Gamma = \rho_o\Gamma_o$ (as in Wilkins), the following relationship may be derived

$$e = -\int_{v_o}^{v}(p - \rho_o\Gamma_o C_v T)dv + C_v(T - T_o) \tag{5.57}$$

Since the integral is for $dS = 0$, it may be integrated to give an alternate form of 5.54,

$$T_{isentropic} = T_o \exp(\Gamma_o(1 - \frac{\rho_o}{\rho})) \tag{5.58}$$

which may be substituted into 5.57 along with the expansion for $p(\eta)$ (here for $s_2 = s_3 = 0$)

$$p = \rho_o\Gamma_o e + \rho_o C_o^2(\eta + (2s_\alpha - \frac{\Gamma_o}{2})\eta^2 + s_\alpha(3s_\alpha - \Gamma_o)\eta^3) + O(\eta^4) \tag{5.59}$$

to give the equation

$$e + \int_{v_o}^{v}\rho_o\Gamma_o e dv = C_v(T - T_o) + \rho_o\Gamma_o C_v T_o\int_{v_o}^{v}\exp(\Gamma_o\eta)dv - \rho_o\Gamma_o\int_{v_o}^{v}(\eta + (2s_\alpha - \frac{\Gamma_o}{2})\eta^2 + s_\alpha(3s_\alpha - \Gamma_o)\eta^3)dv \tag{5.60}$$

Finally, using

$$e = e_o(\eta) + \int_{T_o}^{T} C_v dT \tag{5.61}$$

with a polynomial expansion for $e(\eta)$ in powers of η, 5.60 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.

5.4.5 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

$$\dot{\mathbf{s}} = 2\mu(\boldsymbol{\eta} - \boldsymbol{\eta}^p) \tag{5.62}$$

where μ 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 [19]. 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

$$\dot{\mathbf{s}} = 2\mu(\boldsymbol{\eta} - \boldsymbol{\eta}^p) - \sum_{i=1}^{N} \frac{\mathbf{s}_i}{\tau_i} \tag{5.63}$$

where μ is the shear modulus and $\mathbf{s}_i$ are maxwell element stresses which are tracked internally. Also

$$\mathbf{s} = \sum_{i=1}^{N} \mathbf{s}_i \qquad \mu = \sum_{i=1}^{N} \mu_i \tag{5.64}$$

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.

5.4.6 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_mo is required, and either a or Gamma_o along with K_T, or b. T_mo is the initial melting temperature in Kelvin, a is the Kraut-Kennedy coefficient, Gamma_o 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

$$T_m = T_{mo} + bP \tag{5.65}$$

or

$$T_m = T_{mo}\left(1 + a\frac{\rho_o}{\rho}\right) \tag{5.66}$$

The constants in these equations are linked together via the following equation

$$b = \frac{aT_{mo}}{K_T} \tag{5.67}$$

and a is related to the Gruneisen Gamma by the Lindemann Law [20]:

$$a = 2\left(\Gamma_o - \frac{1}{3}\right) \tag{5.68}$$

SCG melt model

We use a pressure dependent relation to determine the melting temperature (T_m). The Steinberg-Cochran-Guinan (SCG) melt model ([21]) has been used for our simulations of copper. This model is based on a modified Lindemann law and has the form

$$T_m(\rho) = T_{mo}\exp\left[2a\left(1 - \frac{1}{\eta}\right)\right]\eta^{2(\Gamma_o - a - 1/3)}; \quad \eta = \frac{\rho}{\rho_o} \tag{5.69}$$

where T_{mo} is the melt temperature at $\eta = 1$, a is the coefficient of the first order volume correction to Grüneisen's gamma (Γ_o).

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 ([22]) can also be used. This model has been used to determine the melt temperature for 4340 steel. The BPS model has the form

$$T_m(p) = T_m(0)\left[\frac{1}{\eta} + \frac{1}{\eta^{4/3}}\frac{\mu_0'}{\mu_0}p\right]; \quad \eta = \left(1 + \frac{K_0'}{K_0}p\right)^{1/K_0'} \tag{5.70}$$

$$T_m(0) = \frac{\kappa\lambda\mu_0 v_{WS}}{8\pi\ln(z-1)\,k_b}\ln\left(\frac{\alpha^2}{4\,b^2\rho_c(T_m)}\right) \tag{5.71}$$

where p is the pressure, $\eta = \rho/\rho_0$ is the compression, μ_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, κ 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, α 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>
```

5.4.7 Shear Modulus

Three models for the shear modulus (μ) 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 [23] ([24])

$$\mu(T) = \mu_o - \frac{D}{exp(T_o/T) - 1} \tag{5.72}$$

where μ_o is the shear modulus at oK, and D, T_o 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 ([13, 21]) is pressure dependent and has the form

$$\mu(p, T) = \mu_o + \frac{\partial \mu}{\partial p}\frac{p}{\eta^{1/3}} + \frac{\partial \mu}{\partial T}(T - 300); \quad \eta = \rho/\rho_o \tag{5.73}$$

where, μ_o is the shear modulus at the reference state(T = 300 K, p = 0, η = 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 [25] 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

$$\mu(p, T) = \frac{1}{\mathcal{J}(\hat{T})}\left[\left(\mu_o + \frac{\partial \mu}{\partial p}\frac{p}{\eta^{1/3}}\right)(1 - \hat{T}) + \frac{\rho}{Cm}\, k_b\, T\right]; \quad C := \frac{(6\pi^2)^{2/3}}{3} f^2 \tag{5.74}$$

where

$$\mathcal{J}(\hat{T}) := 1 + \exp\left[-\frac{1+1/\zeta}{1+\zeta/(1-\hat{T})}\right] \quad \text{for} \quad \hat{T} := \frac{T}{T_m} \in [0, 1+\zeta], \tag{5.75}$$

μ_0 is the shear modulus at 0 K and ambient pressure, ζ 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.

5.4.8 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

$$\Phi = \sqrt{3J_2} - \sigma_y \tag{5.76}$$

where J_2 is the second invariant of the deviatoric stress tensor ($J_2 = \frac{1}{2}\mathbf{s} : \mathbf{s}$) and σ_y is the flow stress. Currently the return algorithms are restricted to plastic flow in the direction of the deviatoric stress (Eqn. 5.12). 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 [26, 27] depends on porosity. *This model is for experts only!!!* Here are some caveats: Formally, you can replace the flow stress in Gursonfhs 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

$$\Phi = \left(\frac{\sigma_{eq}}{\sigma_f}\right)^2 + 2q_1 f_* \cosh\left(q_2 \frac{Tr(\sigma)}{2\sigma_f}\right) - (1 + q_3 f_*^2) = 0 \tag{5.77}$$

where q_1, q_2, q_3 are material constants and f_* is the porosity (damage) function given by

$$f* = \begin{cases} f & \text{for } f \le f_c, \\ f_c + k(f - f_c) & \text{for } f > f_c \end{cases} \tag{5.78}$$

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 [28]. The rate of growth of porosity and the void nucleation rate are given by the following equations [29]

$$\dot{f} = \dot{f}_{\text{nucl}} + \dot{f}_{\text{grow}} \tag{5.79}$$

$$\dot{f}_{\text{grow}} = (1 - f)\text{Tr}(\boldsymbol{D}_p) \tag{5.80}$$

$$\dot{f}_{\text{nucl}} = \frac{f_n}{(s_n\sqrt{2\pi})} \exp\left[-\tfrac{1}{2}\frac{(\epsilon_p - \epsilon_n)^2}{s_n^2}\right]\dot{\epsilon}_p \tag{5.81}$$

where $\boldsymbol{D}_p$ is the rate of plastic deformation tensor, f_n is the volume fraction of void nucleating particles , ϵ_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>
```

5.4.9 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

$$\sigma_y(\varepsilon_p^{\text{eq}}) = \sigma_y + K(\varepsilon_p^{\text{eq}}) \tag{5.82}$$

where $\varepsilon_p^{\text{eq}}$ is the equivalent plastic strain, σ_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 ([30]) is purely empirical and gives the following relation for the flow stress (σ_y)

$$\sigma_y(\varepsilon_p^{\text{eq}}, \dot{\varepsilon}_p, T) = \left[A + B(\varepsilon_p^{\text{eq}})^n\right]\left[1 + C\ln(\dot{\varepsilon}_p^*)\right]\left[1 - (T^*)^m\right] \tag{5.83}$$

where $\varepsilon_p^{\text{eq}}$ is the equivalent plastic strain, $\dot{\varepsilon}_p$ is the plastic strain rate, A, B, C, n, m are material constants,

$$\dot{\varepsilon}_p^* = \frac{\dot{\varepsilon}_p}{\dot{\varepsilon}_{p0}}; \quad T^* = \frac{(T - T_0)}{(T_m - T_0)}, \tag{5.84}$$

$\dot{\varepsilon}_{p0}$ 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 [21] for high strain rate situations and extended to low strain rates and bcc materials by [31]. The flow stress

in this model is given by

$$\sigma_y(\varepsilon_p^{\text{eq}}, \dot{\varepsilon}_p, T) = \left[\sigma_a f(\varepsilon_p^{\text{eq}}) + \sigma_t(\dot{\varepsilon}_p, T)\right] \frac{\mu(p,T)}{\mu_o} \tag{5.85}$$

where σ_a is the athermal component of the flow stress, $f(\varepsilon_p^{\text{eq}})$ is a function that represents strain hardening, σ_t is the thermally activated component of the flow stress, $\mu(p,T)$ is the shear modulus, and μ_o is the shear modulus at standard temperature and pressure. The strain hardening function has the form

$$f(\varepsilon_p^{\text{eq}}) = \left[1 + \beta(\varepsilon_p^{\text{eq}} + \varepsilon_{pi})\right]^n; \quad \sigma_a f(\varepsilon_p^{\text{eq}}) \le \sigma_{\max} \tag{5.86}$$

where β, n are work hardening parameters, and ε_{pi} is the initial equivalent plastic strain. The thermal component σ_t is computed using a bisection algorithm from the following equation (based on the work of [32])

$$\dot{\varepsilon}_p = \left[\frac{1}{C_1}\exp\left[\frac{2U_k}{k_b\,T}\left(1 - \frac{\sigma_t}{\sigma_p}\right)^2\right] + \frac{C_2}{\sigma_t}\right]^{-1}; \quad \sigma_t \le \sigma_p \tag{5.87}$$

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, σ_p is the Peierls stress. The constants C_1, C_2 are given by the relations

$$C_1 := \frac{\rho_d L_d a b^2 \nu}{2w^2}; \quad C_2 := \frac{D}{\rho_d b^2} \tag{5.88}$$

where ρ_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, ν 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 ([33–35]) is based on simplified dislocation mechanics. The general form of the equation for the flow stress is

$$\sigma_y(\varepsilon_p^{\text{eq}}, \dot{\varepsilon}_p, T) = \sigma_a + B\exp(-\beta(\dot{\varepsilon}_p)T) + B_o\sqrt{\varepsilon_p^{\text{eq}}}\exp(-\alpha(\dot{\varepsilon}_p)T) \tag{5.89}$$

where σ_a is the athermal component of the flow stress given by

$$\sigma_a := \sigma_g + \frac{k_h}{\sqrt{l}} + K(\varepsilon_p^{\text{eq}})^n, \tag{5.90}$$

σ_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_o are material constants. The functional forms of the exponents α and β are

$$\alpha = \alpha_o - \alpha_1 \ln(\dot{\varepsilon}_p); \quad \beta = \beta_o - \beta_1 \ln(\dot{\varepsilon}_p); \tag{5.91}$$

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 [36] 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([19]) 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

$$\sigma_y(\varepsilon_p^{eq}, \dot{\varepsilon}_p, p, T) = \sigma_g + B\exp(-\beta(T - T_0)) + B_0\sqrt{\omega\varepsilon_p^{eq}}\exp(-\alpha(T - T_0)) \tag{5.92}$$

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, σ_g, which is a constant. The other terms are specified via

$$B = B_{pa}(1 + B_{pb}\sqrt{p})^{B_{pn}}; \quad B_0 = B_{0pa}(1 + B_{0pb}\sqrt{p})^{B_{0pn}}; \quad \omega = \omega_a + \omega_b\ln(\dot{\varepsilon}_p) + \omega_p\sqrt{p} \tag{5.93}$$

where B_{pa}, B_{pb}, B_{pn}, B_{0pa}, B_{0pb}, B_{0pn}, ω_a, ω_b, and ω_p are material parameters. The functional forms of the exponents α and β are (as in the original)

$$\alpha = \alpha_0 - \alpha_1\ln(\dot{\varepsilon}_p); \quad \beta = \beta_0 - \beta_1\ln(\dot{\varepsilon}_p); \tag{5.94}$$

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 ([37–39]) gives the following form for the flow stress

$$\sigma_y(\varepsilon_p^{eq}, \dot{\varepsilon}_p, T) = \sigma_a + (S_i\sigma_i + S_e\sigma_e)\frac{\mu(p,T)}{\mu_o} \tag{5.95}$$

where σ_a is the athermal component of mechanical threshold stress, μ_o is the shear modulus at o K and ambient pressure, σ_i is the component of the flow stress due to intrinsic barriers to thermally activated dislocation motion and dislocation-dislocation interactions, σ_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

$$S_i = \left[1 - \left(\frac{k_b\ T}{g_{oi}b^3\mu(p,T)}\ln\frac{\dot{\varepsilon}_{poi}}{\dot{\varepsilon}_p}\right)^{1/q_i}\right]^{1/p_i} \tag{5.96}$$

$$S_e = \left[1 - \left(\frac{k_b\ T}{g_{oe}b^3\mu(p,T)}\ln\frac{\dot{\varepsilon}_{poe}}{\dot{\varepsilon}_p}\right)^{1/q_e}\right]^{1/p_e} \tag{5.97}$$

where k_b is the Boltzmann constant, b is the magnitude of the Burgers' vector, (g_{oi}, g_{oe}) are normalized activation energies, $(\dot{\varepsilon}_{poi}, \dot{\varepsilon}_{poe})$ 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 (σ_e) is given by a modified Voce law

$$\frac{d\sigma_e}{d\varepsilon_p^{eq}} = \theta(\sigma_e) \tag{5.98}$$

where

$$\theta(\sigma_e) = \theta_o[1 - F(\sigma_e)] + \theta_{IV}F(\sigma_e) \tag{5.99}$$

$$\theta_o = a_o + a_1\ln\dot{\varepsilon}_p + a_2\sqrt{\dot{\varepsilon}_p} - a_3T \tag{5.100}$$

$$F(\sigma_e) = \frac{\tanh\left(\alpha\frac{\sigma_e}{\sigma_{es}}\right)}{\tanh(\alpha)} \tag{5.101}$$

$$\ln\left(\frac{\sigma_{es}}{\sigma_{oes}}\right) = \left(\frac{kT}{g_{oes}b^3\mu(p,T)}\right)\ln\left(\frac{\dot{\varepsilon}_p}{\dot{\varepsilon}_{poes}}\right) \tag{5.102}$$

and θ_o is the hardening due to dislocation accumulation, θ_{IV} is the contribution due to stage-IV hardening, $(a_o, a_1, a_2, a_3, \alpha)$ are constants, σ_{es} is the stress at zero strain hardening rate, σ_{oes} is the saturation threshold stress for deformation at o K, g_{oes} is a constant, and $\dot{\varepsilon}_{poes}$ 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 ([40]) 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

$$\sigma_y(\varepsilon_p^{\text{eq}}, \dot{\varepsilon}_p, T) = \begin{cases} 2\left[\tau_s + \alpha \ln\left[1 - \varphi \exp\left(-\beta - \dfrac{\theta \varepsilon_p^{\text{eq}}}{\alpha\varphi}\right)\right]\right]\mu(p,T) & \text{thermal regime} \\ 2\tau_s \mu(p,T) & \text{shock regime} \end{cases} \tag{5.103}$$

with

$$\alpha := \frac{s_0 - \tau_y}{d}; \quad \beta := \frac{\tau_s - \tau_y}{\alpha}; \quad \varphi := \exp(\beta) - 1 \tag{5.104}$$

where τ_s is a normalized work-hardening saturation stress, s_0 is the value of τ_s at 0K, τ_y is a normalized yield stress, θ 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

$$\tau_s = \max\left\{s_0 - (s_0 - s_\infty)\text{erf}\left[\kappa\hat{T}\ln\left(\frac{\gamma\dot{\xi}}{\dot{\varepsilon}_p}\right)\right], s_0\left(\frac{\dot{\varepsilon}_p}{\gamma\dot{\xi}}\right)^{s_1}\right\} \tag{5.105}$$

$$\tau_y = \max\left\{y_0 - (y_0 - y_\infty)\text{erf}\left[\kappa\hat{T}\ln\left(\frac{\gamma\dot{\xi}}{\dot{\varepsilon}_p}\right)\right], \min\left\{y_1\left(\frac{\dot{\varepsilon}_p}{\gamma\dot{\xi}}\right)^{y_2}, s_0\left(\frac{\dot{\varepsilon}_p}{\gamma\dot{\xi}}\right)^{s_1}\right\}\right\} \tag{5.106}$$

where s_∞ is the value of τ_s close to the melt temperature, (y_0, y_∞) are the values of τ_y at 0K and close to melt, respectively, (κ, γ) are material constants, $\hat{T} = T/T_m$, (s_1, y_1, y_2) are material parameters for the high strain rate regime, and

$$\dot{\xi} = \frac{1}{2}\left(\frac{4\pi\rho}{3M}\right)^{1/3}\left(\frac{\mu(p,T)}{\rho}\right)^{1/2} \tag{5.107}$$

where ρ 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>
```

5.4.10 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 [41–43].

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 [44]. 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

$$\dot{D} = \frac{\dot{\epsilon_p}}{\epsilon_p^f}; \quad \epsilon_p^f = \left[D_1 + D_2 \exp\left(\frac{D_3}{3}\sigma^*\right)\right]\left[1 + D_4 \ln(\dot{\epsilon_p}^*)\right]\left[1 + D_5 T^*\right] \; ; \quad \sigma^* = \frac{\mathrm{Tr}(\boldsymbol{\sigma})}{\sigma_{eq}}; \tag{5.108}$$

where D is the damage variable which has a value of 0 for virgin material and a value of 1 at fracture, ϵ_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 [45] is satisfied. This condition can be written as

$$(f/f_c)^2 + (\epsilon_p/\epsilon_p^f)^2 = 1 \tag{5.109}$$

where f is the current porosity, f_c is the maximum allowable porosity, ϵ_p is the current plastic strain, and ϵ_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 [46] and the loss of hyperbolicity criterion (using the determinant of the acoustic tensor) [47, 48].

The simplest criterion that can be used is the Drucker stability postulate [46] 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

$$\dot{\boldsymbol{\sigma}} : \boldsymbol{D}^p \leq 0 \tag{5.110}$$

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 [47, 48]. 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 [49]. In this approach, we assume that the strain is localized in a band with normal **n**, and the magnitude of the velocity difference across the band is **g**. Then the bifurcation condition leads to the relation

$$R_{ij}g_j = 0\,; \quad R_{ij} = M_{ikjl}n_k n_l + M_{ilkj}n_k n_l - \sigma_{ik}n_j n_k \tag{5.111}$$

where M_{ijkl} are the components of the co-rotational tangent modulus tensor and σ_{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 [50–52]. Viscoplastic stress update models or nonlocal/gradient plasticity models [53, 54] 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$_{\text{m}}$odeltype = "elastic$_{\text{p}}$lastic" > 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>
```

5.5 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`.

5.6 Arenisca

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 fhcapfh 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 [55]. Usage is as follows:

```
<constitutive_model type="Arenisca">
    <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

$$\sqrt{J_2} + FSLOPE \times (I_1 - PEAKI1) = 0, \tag{5.112}$$

<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

$$p_3 - \bar{\varepsilon}_v^p = p_3 \exp -3p_1(\bar{p} - p_0) \tag{5.113}$$

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 Arenisca model, linear Zieglerfhs rule is used:

$$\dot{\alpha}_{ij} = \dot{\mu}(\boldsymbol{\sigma}_{ij} - \alpha_{ij}) \tag{5.114}$$

in which α_{ij} is the back stress tensor, $\dot{\alpha}_{ij}$ is time derivative of the back stress tensor, and

$$\dot{\mu} = c\dot{\xi}^p \tag{5.115}$$

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 Arenisca model to consider the fluid-filled porous effects:

$$\begin{aligned}\frac{\partial X}{\partial \varepsilon_v^p} = \quad & \frac{1}{p_1 p_3}\exp\left(-p_1 X - p_0\right) - \frac{3K_f\left(\exp(p_3+p_4)-1\right)\exp(p_3+p_4+\varepsilon_v^p)}{\left(\exp(p_3+p_4+\varepsilon_v^p)-1\right)^2} \\ & + \frac{3K_f\left(\exp(p_3+p_4)-1\right)\exp(p_3+\varepsilon_v^p)}{\left(\exp(p_3+\varepsilon_v^p)-1\right)^2}\end{aligned} \tag{5.116}$$

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>, ε_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

$$\alpha_{n+1}^{\text{iso}} = \alpha_n^{\text{iso}} + \frac{3K_f \exp(p_3)\left(\exp(p_4) - \exp(\varepsilon_v^p)\right)}{\left(\exp(p_3 + \varepsilon_v^p) - 1\right)} \dot{\varepsilon}_v^p \Delta t \, \boldsymbol{1} \tag{5.117}$$

in which $\boldsymbol{1}$ is the second-order identity tensor. Also, the effective bulk modulus is calculated as

$$K_e = B0 + \frac{K_f \left(\exp(p_3 + p_4) - 1\right) \exp(p_3 + p_4 + \varepsilon_v^e + \varepsilon_v^p)}{\left(\exp(p_3 + p_4 + \varepsilon_v^e + \varepsilon_v^p) - 1\right)^2} \tag{5.118}$$

in which ε_v^e is the volumetric part of the elastic strain.

5.7 Arena: Partially Saturated Soils

The Arena soil model is designed to be used to simulate high strain-rate compression and shear of partially saturated soils. For a detailed description, please see the manual in the TheoryManual/ArenaSoil directory.

5.7.1 A typical input file

The inputs for the Arena soil model are typically specified as follows.

```
<?xml version='1.0' encoding='ISO-8859-1' ?>

<Uintah_specification>

  <Meta>
      <title>Uniaxial_Strain_Compression fully saturated</title>
  </Meta>

  <SimulationComponent type="mpm" />

  <Time>
      <maxTime> 1.0 </maxTime>
      <initTime> 0.0 </initTime>
      <delt_min> 1.0e-6 </delt_min>
      <delt_max> 0.01 </delt_max>
      <timestep_multiplier> 0.3 </timestep_multiplier>
  </Time>

  <DataArchiver>
      <filebase>UniaxialStrainSaturated.uda</filebase>
      <outputInterval>0.04</outputInterval>
      <save label = "p.x"/>
      <save label = "p.color"/>
      <save label = "p.temperature"/>
      <save label = "p.velocity"/>
      <save label = "p.particleID"/>
      <save label = "p.stress"/>
      <save label = "g.mass"/>
      <save label = "p.deformationMeasure"/>
      <save label = "g.acceleration"/>
      <save label = "p.capX"/>
      <save label = "p.plasticStrain"/>
      <save label = "p.plasticCumEqStrain"/>
      <save label = "p.plasticVolStrain"/>
      <save label = "p.p3"/>
      <save label = "p.porePressure"/>
      <save label = "p.ArenaPEAKI1"/>
      <save label = "p.ArenaFSLOPE"/>
      <save label = "p.ArenaSTREN"/>
      <save label = "p.ArenaYSLOPE"/>
      <save label = "p.ArenaBETA"/>
      <save label = "p.ArenaCR"/>
      <save label = "p.ArenaT1"/>
      <save label = "p.ArenaT2"/>
      <save label = "p.porosity"/>
      <save label = "p.saturation"/>
      <save label = "p.elasticVolStrain"/>
      <save label = "p.stressQS"/>
      <save label = "p.COHER"/>
      <save label = "p.TGROW"/>
      <checkpoint cycle = "2" timestepInterval = "4000"/>
  </DataArchiver>

  <MPM>
    <time_integrator>              explicit   </time_integrator>
    <interpolator>                 linear     </interpolator>
    <use_load_curves>              false      </use_load_curves>
    <minimum_particle_mass>        1.0e-15    </minimum_particle_mass>
    <minimum_mass_for_acc>         1.0e-15    </minimum_mass_for_acc>
    <maximum_particle_velocity>    1.0e5      </maximum_particle_velocity>
    <artificial_damping_coeff>     0.0        </artificial_damping_coeff>
```

```
  <artificial_viscosity>          true       </artificial_viscosity>
  <artificial_viscosity_heating> false       </artificial_viscosity_heating>
  <do_contact_friction_heating>  false       </do_contact_friction_heating>
  <create_new_particles>          false      </create_new_particles>
  <UseMomentumForm>               false      </UseMomentumForm>
  <withColor>                     true       </withColor>
  <UsePrescribedDeformation>      true       </UsePrescribedDeformation>
  <PrescribedDeformationFile>     UniaxialStrain_PrescribedDeformation.inp    </
      PrescribedDeformationFile>
  <minimum_subcycles_for_F>        -2        </minimum_subcycles_for_F>
  <erosion algorithm = "none"/>
</MPM>

<PhysicalConstants>
    <gravity>[0,0,0]</gravity>
</PhysicalConstants>

<MaterialProperties>
  <MPM>
    <material name="MasonSand">
      <density>1800</density>
      <melt_temp>3695.0</melt_temp>
      <room_temp>294.0</room_temp>
      <thermal_conductivity>174.0e-7</thermal_conductivity>
      <specific_heat>134.0e-8</specific_heat>

      <constitutive_model type="arena">

        <reference_porosity> 0.42 </reference_porosity>
        <initial_porosity>   0.40 </initial_porosity>
        <initial_saturation> 0.5  </initial_saturation>
        <initial_fluid_pressure> 0.0 </initial_fluid_pressure>

        <p0>      0.1e4     </p0>
        <p1>      482.68e6  </p1>
        <p1_sat>  1.0       </p1_sat>
        <p1_density_scale_fac> 5.0 </p1_density_scale_fac>
        <p2>      0.719     </p2>
        <p3>      0.448     </p3>

        <elastic_moduli_model type="arena">
          <b0>  0.0029  </b0>
          <b1>  0.4731  </b1>
          <b2>  1.5057  </b2>
          <b3>  2.5728  </b3>
          <b4>  2.0799  </b4>
          <G0>  7.0e8   </G0>
          <nu1> 0.35    </nu1>
          <nu2> -0.05   </nu2>
        </elastic_moduli_model>

        <plastic_yield_condition type="arena">
          <PEAKI1> 1.0e3   </PEAKI1>
          <weibullDist_PEAKI1> weibull, 1.0e3, 4, 0.001, 1 </weibullDist_PEAKI1>
          <FSLOPE> 0.453   </FSLOPE>
          <weibullDist_FSLOPE> weibull, 0.453, 4, 0.001, 1 </weibullDist_FSLOPE>
          <STREN>  1.0e7   </STREN>
          <weibullDist_STREN> weibull, 1.0e7, 4, 0.001, 1 </weibullDist_STREN>
          <YSLOPE> 0.31    </YSLOPE>
          <weibullDist_YSLOPE> weibull, 0.31, 4, 0.001, 1 </weibullDist_YSLOPE>
          <BETA>   1.0     </BETA>
          <CR>     0.5     </CR>
          <T1>     5.0e-5  </T1>
          <T2>     0.5     </T2>
        </plastic_yield_condition>

        <use_disaggregation_algorithm> false </use_disaggregation_algorithm>
        <subcycling_characteristic_number>256</subcycling_characteristic_number>
        <consistency_bisection_tolerance>0.0001</consistency_bisection_tolerance>
        <yield_surface_radius_scaling_factor> 1000.0 </
            yield_surface_radius_scaling_factor>

        <do_damage> true </do_damage>
```

```
          <fspeed> 7 </fspeed>
          <time_at_failure> 800.0e-6 </time_at_failure>

        </constitutive_model>

        <geom_object>
          <box label = "Plate1">
            <min>[0.0,0.0,0.0]</min>
            <max>[1.0,1.0,1.0]</max>
          </box>
          <res>[1,1,1]</res>
          <velocity>[0.0,0.0,0.0]</velocity>
          <temperature>294</temperature>
          <color>0</color>
        </geom_object>

      </material>

      <contact>
        <type>null</type>
        <materials>[0]</materials>
        <mu>0.1</mu>
      </contact>

    </MPM>
  </MaterialProperties>

  <Grid>

      <BoundaryConditions>
        ......
      </BoundaryConditions>

      <Level>
        <Box label = "Domain">
          <lower>[-2.0, -2.0, -2.0]</lower>
          <upper>[3.0, 3.0, 3.0]</upper>
          <resolution>[5,5,5]</resolution>
          <extraCells>[0,0,0]</extraCells>
          <patches>[1,1,1]</patches>
        </Box>
      </Level>

  </Grid>

</Uintah_specification>
```

5.7.2 Model components

The convention used in Vaango is that tension is positive and compression is negative. To keep the notation simple we define, for any x,

$$\bar{x} := -x\,, \quad \dot{x} := \frac{\partial x}{\partial t}\,. \tag{5.119}$$

Summary 5.7.1 ***Bulk modulus model***

Drained soil:
The equation of state of the drained soil is

$$K_d = \frac{[K_s]^2}{[K_s - n_s \bar{p}^{\text{eff}}]} \left[b_0 + \frac{b_1 b_3 b_4 (\bar{\varepsilon}_v^e)^{b_4 - 1}}{\left[b_2 (\bar{\varepsilon}_v^e)^{b_4} + b_3 \right]^2} \right], \quad \bar{\varepsilon}_v^e \approx \left[\frac{b_3 \bar{p}^{\text{eff}}}{b_1 K_s - b_2 \bar{p}^{\text{eff}}} \right]^{1/b_4}.$$

Partially saturated soil:
The bulk modulus model is

$$K = K_d + \frac{\left(1 - \frac{K_d}{K_s}\right)^2}{\frac{1}{K_s}\left(1 - \frac{K_d}{K_s}\right) + \phi\left(\frac{S_w}{K_w} + \frac{1 - S_w}{K_a} - \frac{1}{K_s}\right)}$$

where

$$K_s(\bar{p}) = K_{s0} + n_s(\bar{p} - \bar{p}_{s0}), \quad K_w(\bar{p}) = K_{w0} + n_w(\bar{p} - \bar{p}_{w0}), \quad K_a(\bar{p}) = \gamma(\bar{p} + \bar{p}_r)$$

Summary 5.7.2 ***Shear modulus model***

The shear modulus is either a constant (G_0) or determined using a variable Poisson's ratio (ν)

$$\nu = \nu_1 + \nu_2 \exp\left[-\frac{K_d(\bar{p}^{\text{eff}}, \bar{\varepsilon}_v^p, \phi, S_w)}{K_s(\bar{p}^{\text{eff}})} \right]$$

$$G(\bar{p}^{\text{eff}}, \bar{\varepsilon}_v^p, \phi, S_w) = \frac{3K_d(\bar{p}^{\text{eff}}, \bar{\varepsilon}_v^p, \phi, S_w)(1 - 2\nu)}{2(1 + \nu)}.$$

Summary 5.7.3 ***Yield function***

The Arena yield function is

$$f = \sqrt{J_2} - F_f(\bar{I}_1, \zeta) F_c(\bar{I}_1, \bar{\zeta}, \bar{X}, \bar{\kappa}) = \sqrt{J_2} - F_f(\bar{p}^{\text{eff}}) F_c(\bar{p}^{\text{eff}}, \bar{X}, \bar{\kappa}) \tag{5.120}$$

where

$$F_f(\bar{p}^{\text{eff}}) = a_1 - a_3 \exp[-3a_2 \bar{p}^{\text{eff}})] + 3a_4 \bar{p}^{\text{eff}} \tag{5.121}$$

and

$$F_c(\bar{p}^{\text{eff}}, \bar{X}, \bar{\kappa}) = \begin{cases} 1 & \text{for} \quad 3\bar{p}^{\text{eff}} \le \bar{\kappa} \\ \sqrt{1 - \left(\frac{3\bar{p}^{\text{eff}} - \bar{\kappa}}{\bar{X} - \bar{\kappa}} \right)^2} & \text{for} \quad 3\bar{p}^{\text{eff}} > \bar{\kappa}. \end{cases} \tag{5.122}$$

Non-associativity is modeled using a parameter β that modifies $\sqrt{J_2}$.

Summary 5.7.4 ***Hydrostatic strength model***

Drained soil:

$$\overline{X}_d(\overline{\varepsilon_v^p}) - p_o = p_1 \left[\frac{1 - \exp(-p_3)}{1 - \exp(-p_3 + \overline{\varepsilon_v^p})} - 1 \right]^{1/p_2} \quad , \quad p_3 = -\ln(1 - \phi_o) \, .$$

Partially saturated soil:

$$\overline{X}(\overline{\varepsilon_v^p}) = 3K(\bar{I}_1, \overline{\varepsilon_v^p}, \phi, S_w) \, \bar{\varepsilon}_v^{e,\text{yield}}(\overline{\varepsilon_v^p})$$

where

$$\varepsilon_v^{e,\text{yield}}(\overline{\varepsilon_v^p}) = \frac{\overline{X}_d(\overline{\varepsilon_v^p})}{3\,K_d\left(\dfrac{\overline{X}_d(\overline{\varepsilon_v^p})}{6}, \overline{\varepsilon_v^p}\right)}$$

Summary 5.7.5 ***Pore pressure model***

Solve $g(\bar{\zeta}, \overline{\varepsilon_v^p}) = 0$ for $\bar{\zeta}$.

$$g(\bar{\zeta}, \overline{\varepsilon_v^p}) = -\exp(-\overline{\varepsilon_v^p}) + \phi_o\,(1 - S_o) \exp\left[-\frac{1}{\gamma} \ln\left(\frac{\bar{\zeta}}{\bar{p}_r} + 1\right)\right] + \phi_o\, S_o \exp\left(-\frac{\bar{\zeta} - \bar{p}_o}{K_w}\right) + (1 - \phi_o) \exp\left(-\frac{\bar{\zeta}}{K_s}\right) .$$

Alternatively, integrate

$$\bar{\zeta} = \int \frac{d\bar{\zeta}}{d\overline{\varepsilon_v^p}} \, d\overline{\varepsilon_v^p} \, .$$

where

$$\frac{d\bar{\zeta}}{d\overline{\varepsilon_v^p}} = \frac{\exp(-\overline{\varepsilon_v^p})}{\mathcal{B}} \, ,$$

and

$$\mathcal{B} := \left[\frac{\phi_o\,(1 - S_o)}{\gamma(\bar{p}_r + \bar{\zeta})}\right] \exp\left[-\frac{1}{\gamma} \ln\left(\frac{\bar{\zeta}}{\bar{p}_r} + 1\right)\right] + \frac{\phi_o\, S_o}{K_w} \exp\left(\frac{\bar{p}_o - \bar{\zeta}}{K_w}\right) + \frac{1 - \phi_o}{K_s} \exp\left(-\frac{\bar{\zeta}}{K_s}\right) .$$

Summary 5.7.6 ***Saturation and porosity evolution***

Saturation:

$$S_w(\varepsilon_v) = \frac{\mathcal{C}(\varepsilon_v)}{1+\mathcal{C}(\varepsilon_v)}\ ,\quad \mathcal{C}(\varepsilon_v) := \left(\frac{S_o}{1-S_o}\right)\exp(\varepsilon_v^w)\exp(-\varepsilon_v^a)\,.$$

where ϕ_o, S_o are the initial porosity and saturation, and

$$\varepsilon_v^w(\varepsilon_v) = -\frac{\bar{p}(\varepsilon_v)-\bar{p}_o}{K_w}\ ,\quad \varepsilon_v^a(\varepsilon_v) = -\frac{1}{\gamma}\ln\left[1+\frac{\bar{p}(\varepsilon_v)}{\bar{p}_r}\right]\ ,\quad \varepsilon_v^s(\varepsilon_v) = -\frac{\bar{p}(\varepsilon_v)}{K_s}\,.$$

Porosity:

$$\phi(\varepsilon_v) = \phi_o\left(\frac{1-S_o}{1-S_w(\varepsilon_v)}\right)\left[\frac{\exp(\varepsilon_v^a)}{\exp(\varepsilon_v)}\right]\,. \tag{5.123}$$

Note that

$$\exp(\varepsilon_v) = (1-S_o)\phi_o\exp(\varepsilon_v^a) + S_o\phi_o\exp(\varepsilon_v^w) + (1-\phi_o)\exp(\varepsilon_v^s)$$

5.8 Tabular plasticity

The tabular plasticity model is designed to simulate hypoelastic-plasticity using tabulated data for elastic moduli and the yield function for J_2 and $J_2 - I_1$ models of perfect plasticity.

The model has been designed with the high-rate deformation of soils in mind but can also be used for non-hardening metals. Sample input files for linear elastic materials and von Mises plasticity can be found in src/StandAlone/inputs/MPM/TabularModels/TabularPlasticity.

5.8.1 An input file

The inputs for the tabular plasticity model are typically specified as follows.

```
<?xml version='1.0' encoding='ISO-8859-1' ?>

<Uintah_specification>

  <Meta>
      <title>Tabular_Verification_Test_05_Uniaxial_Strain_Compression_DP_with_LoadUnload
          </title>
  </Meta>

  <SimulationComponent type="mpm" />

  <Time>
      <maxTime> 8.0 </maxTime>
      <initTime> 0.0 </initTime>
      <delt_min> 1.0e-8 </delt_min>
      <delt_max> 0.01 </delt_max>
      <timestep_multiplier> 0.3 </timestep_multiplier>
  </Time>

  <DataArchiver>
      <filebase>TabularTest_05_UniaxialStrainLoadUnloadNonLinDPNonLin.uda</filebase>
      <outputInterval>1.0e-3</outputInterval>
      <outputInitTimestep/>
      <save label = "p.x"/>
      <save label = "p.color"/>
      <save label = "p.temperature"/>
      <save label = "p.velocity"/>
      <save label = "p.particleID"/>
      <save label = "p.stress"/>
      <save label = "g.mass"/>
      <save label = "p.deformationGradient"/>
      <save label = "g.acceleration"/>
      <save label = "p.plasticVolStrain"/>
      <save label = "p.elasticVolStrain"/>
      <checkpoint cycle = "2" timestepInterval = "2000"/>
  </DataArchiver>

  <MPM>
    <time_integrator>              explicit    </time_integrator>
    <interpolator>                 linear      </interpolator>
    <use_load_curves>              false       </use_load_curves>
    <minimum_particle_mass>        1.0e-15     </minimum_particle_mass>
    <minimum_mass_for_acc>         1.0e-15     </minimum_mass_for_acc>
    <maximum_particle_velocity>    1.0e5       </maximum_particle_velocity>
    <artificial_damping_coeff>     0.0         </artificial_damping_coeff>
    <artificial_viscosity>         true        </artificial_viscosity>
    <artificial_viscosity_heating> false       </artificial_viscosity_heating>
    <do_contact_friction_heating>  false       </do_contact_friction_heating>
    <create_new_particles>         false       </create_new_particles>
    <use_momentum_form>            false       </use_momentum_form>
    <with_color>                   true        </with_color>
    <use_prescribed_deformation>   true        </use_prescribed_deformation>
    <prescribed_deformation_file>  TabularTest_05_PrescribedDeformation.inp    </
        prescribed_deformation_file>
    <erosion algorithm = "none"/>
  </MPM>
```

```
  <PhysicalConstants>
      <gravity>[0,0,0]</gravity>
  </PhysicalConstants>

  <MaterialProperties>
    <MPM>
      <material name="TabularPlastic">
        <density>1050</density>
        <melt_temp>3695.0</melt_temp>
        <room_temp>294.0</room_temp>
        <thermal_conductivity>174.0e-7</thermal_conductivity>
        <specific_heat>134.0e-8</specific_heat>
        <constitutive_model type="tabular_plasticity">
          <elastic_moduli_model type="tabular">
            <filename>TabularTest_05_Elastic.json</filename>
            <independent_variables>PlasticStrainVol, TotalStrainVol</
                independent_variables>
            <dependent_variables>Pressure</dependent_variables>
            <interpolation type="linear"/>
            <G0>3500</G0>
            <nu>0.35</nu>
          </elastic_moduli_model>
          <plastic_yield_condition type="tabular">
            <filename>TabularTest_05_Yield.json</filename>
            <independent_variables>Pressure</independent_variables>
            <dependent_variables>SqrtJ2</dependent_variables>
            <interpolation type="linear"/>
          </plastic_yield_condition>
        </constitutive_model>
        <geom_object>
          <box label = "Plate1">
            <min>[0.0,0.0,0.0]</min>
            <max>[1.0,1.0,1.0]</max>
          </box>
          <res>[1,1,1]</res>
          <velocity>[0.0,0.0,0.0]</velocity>
          <temperature>294</temperature>
          <color>0</color>
        </geom_object>
      </material>
      <contact>
        <type>null</type>
        <materials>[0]</materials>
        <mu>0.1</mu>
      </contact>
    </MPM>
  </MaterialProperties>

  <Grid>
      <BoundaryConditions>
      </BoundaryConditions>
      <Level>
        <Box label = "1">
            <lower>[-2.0, -2.0, -2.0]</lower>
            <upper>[3.0, 3.0, 3.0]</upper>
            <resolution>[5,5,5]</resolution>
            <extraCells>[0,0,0]</extraCells>
            <patches>[1,1,1]</patches>
        </Box>
      </Level>
  </Grid>

</Uintah_specification>
```

The prescribed deformation file

The input file listed above is for a single particle driven by a prescribed deformation gradient. The deformation file is TabularTest_05_PrescribedDeformation.inp which contains the following:

```
0  1.0  0  0  0  1  0  0  0  1  0  0  0  0
1  3.0  0  0  0  1  0  0  0  1  0  0  0  0
3  1.0  0  0  0  1  0  0  0  1  0  0  0  0
```

```
5   0.5  0  0  0  1  0  0  0  1  0  0  0  0
7   1.0  0  0  0  1  0  0  0  1  0  0  0  0
8   1.02 0  0  0  1  0  0  0  1  0  0  0  0
```

The first column is the time (seconds), the second column is deformation gradient F_{xx}. This deformation represents uniaxial strain in the x-direction.

The bulk-modulus model

The bulk modulus model is non-linear (tanh) and in input as JSON file containing data that covers the range expected during a simulation. A sample file (TabularTest_05_Elastic.json) is shown below:

```
{"Vaango_tabular_data": {
  "Meta" : {
    "title" : "Nonlinear elastic data"
  },
  "Data" : {
    "PlasticStrainVol" : [-2.5, 2.5],
    "Data" : [{
      "TotalStrainVol" : [-15.000000, -14.310345, -13.620690, -12.931034, -12.241379, -11.551724, -10.862069, -10.172414, -9.482759, -8.793103, -8.103448, -7.413793, -6.724138, -6.034483, -5.344828, -4.655172, -3.965517, -3.275862, -2.586207, -1.896552, -1.206897, -0.517241, 0.172414, 0.862069, 1.551724, 2.241379, 2.931034, 3.620690, 4.310345, 5.000000],
      "Pressure" : [-2959.842894, -2943.464146, -2920.494168, -2888.366995, -2843.600634, -2781.548344, -2696.156593, -2579.811517, -2423.426002, -2217.004505, -1950.975669, -1618.496214, -1218.556435, -759.014896, -257.981931, 257.981931, 759.014896, 1218.556435, 1618.496214, 1950.975669, 2217.004505, 2423.426002, 2579.811517, 2696.156593, 2781.548344, 2843.600634, 2888.366995, 2920.494168, 2943.464146, 2959.842894]
    }, {
      "TotalStrainVol" : [-5.000000, -4.310345, -3.620690, -2.931034, -2.241379, -1.551724, -0.862069, -0.172414, 0.517241, 1.206897, 1.896552, 2.586207, 3.275862, 3.965517, 4.655172, 5.344828, 6.034483, 6.724138, 7.413793, 8.103448, 8.793103, 9.482759, 10.172414, 10.862069, 11.551724, 12.241379, 12.931034, 13.620690, 14.310345, 15.000000],
      "Pressure" : [-2959.842894, -2943.464146, -2920.494168, -2888.366995, -2843.600634, -2781.548344, -2696.156593, -2579.811517, -2423.426002, -2217.004505, -1950.975669, -1618.496214, -1218.556435, -759.014896, -257.981931, 257.981931, 759.014896, 1218.556435, 1618.496214, 1950.975669, 2217.004505, 2423.426002, 2579.811517, 2696.156593, 2781.548344, 2843.600634, 2888.366995, 2920.494168, 2943.464146, 2959.842894]
    }]
  }
}}
```

The yield function

The yield function in this example is a nonlinear Drucker-Prager model. The input file (TabularTest_05_Yield.json) is shown below:

```
{"Vaango_tabular_data": {
  "Meta" : {
    "title" : "Quadratic Drucker-Prager yield data"
  },
  "Data" : {
    "Pressure" : [-333.333333, -298.850575, -264.367816, -229.885057, -195.402299, -160.919540, -126.436782, -91.954023, -57.471264, -22.988506, 11.494253, 45.977011, 80.459770, 114.942529, 149.425287, 183.908046, 218.390805, 252.873563, 287.356322, 321.839080, 356.321839, 390.804598, 425.287356, 459.770115, 494.252874, 528.735632, 563.218391, 597.701149, 632.183908, 666.666667],
    "SqrtJ2" :   [0.000000, 45.485883, 64.326752, 78.783860, 90.971765, 101.709526, 111.417203, 120.344334, 128.653504, 136.457648, 143.838990, 150.859606, 157.567719, 164.001682, 170.192589, 176.166066,181.943530, 187.543098, 192.980256, 198.268366, 203.419051, 208.442500, 213.347701, 218.142630, 222.834406, 227.429413, 231.933403, 236.351579, 240.688667, 244.948974]
  }
}}
```

5.8.2 Model components

Since the tabular plasticity model was designed for materials that have almost no tensile strength, the inputs are expected in the **compression positive** convention. Note that the general convention used in the Vaango code is that **tension is positive and compression is negative**. Conversions are done internally in the code to make sure that signs are consistent.

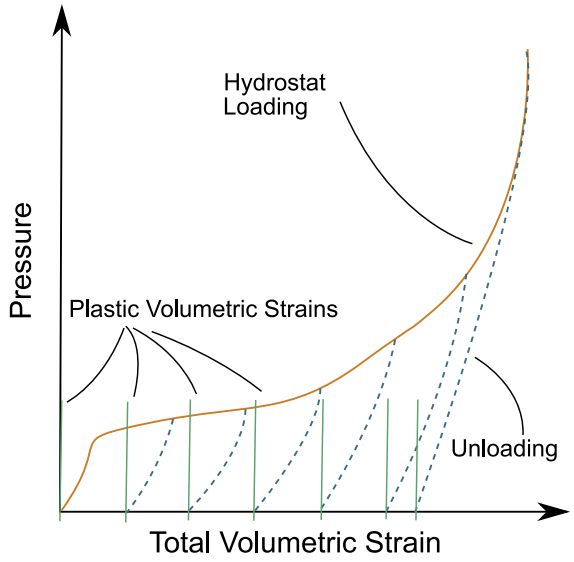
Summary 5.8.1 ***Shear modulus model***

The shear modulus is either a constant (G_o) or determined using a Poisson's ratio (ν) from the tabular bulk modulus, $K(p)$, using a linear elastic model:

$$G = \frac{3K(1-2\nu)}{2(1+\nu)} \tag{5.124}$$

This relation is activated if $\nu \in [-1.0, 0.5)$, otherwise the constant shear modulus is used.

Summary 5.8.2 ***Bulk modulus model***

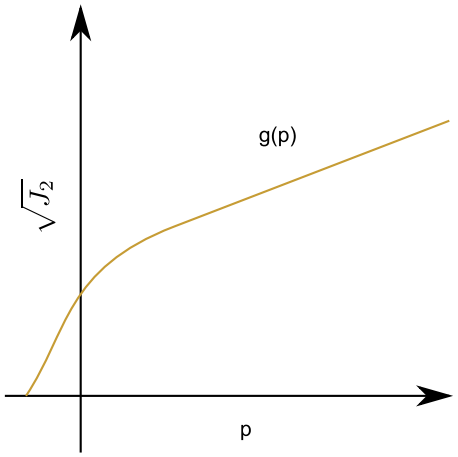


The bulk modulus is determined from a table of unloading curves. Each unloading curve is associated with a Hencky plastic volumetric strain ($\overline{\varepsilon_v^p}$). These strains are associated in the JSON file with the key PlasticStrainVol and added as the first independent variable in the Vaango ups input file.
For each plastic strain value, the JSON file has to contain an associated data set of TotalStrainVol (the total Hencky volumetric strain, $\overline{\varepsilon_v}$) and the Pressure (the mean stress, $\overline{p}$).

Summary 5.8.3 ***Yield function***

The tabular yield condition is

$$f = \sqrt{J_2} - g(\bar{p}) = 0 \tag{5.125}$$



The function $g(\bar{p})$ is provided in tabular form. Only one such function is allowed. The independent variable in the associated JSON file will have the name Pressure while the dependent variable will have the name SqrtJ2.

5.8.3 Examples

Several examples of input files for exercising various features of the TabularPlasticity model can be found in the directory StandAlone/inputs/MPM/TabularModels/TabularPlasticity.

von Mises plasticity with linear elasticity compression with rotation

The input file is TabularTest_01_UniaxialStrainRotateJ2Lin.ups. For this problem, $g(\bar{p}) = \sigma_y$ and K, G are constant. The input JSON file for the bulk modulus model is

```
{"Vaango_tabular_data": {
  "Meta" : {
    "title" : "Test linear elastic data"
  },
  "Data" : {
    "PlasticStrainVol" : [-10.0, 10.0],
    "Data" : [{
      "TotalStrainVol" : [-20.0, 20.0],
      "Pressure" : [-1.0e5,  1.0e5]
    }, {
      "TotalStrainVol" : [-20.0, 20.0],
      "Pressure" : [-1.0e5, 1.0e5]
    }]
  }
}}
```

and that for the yield function is

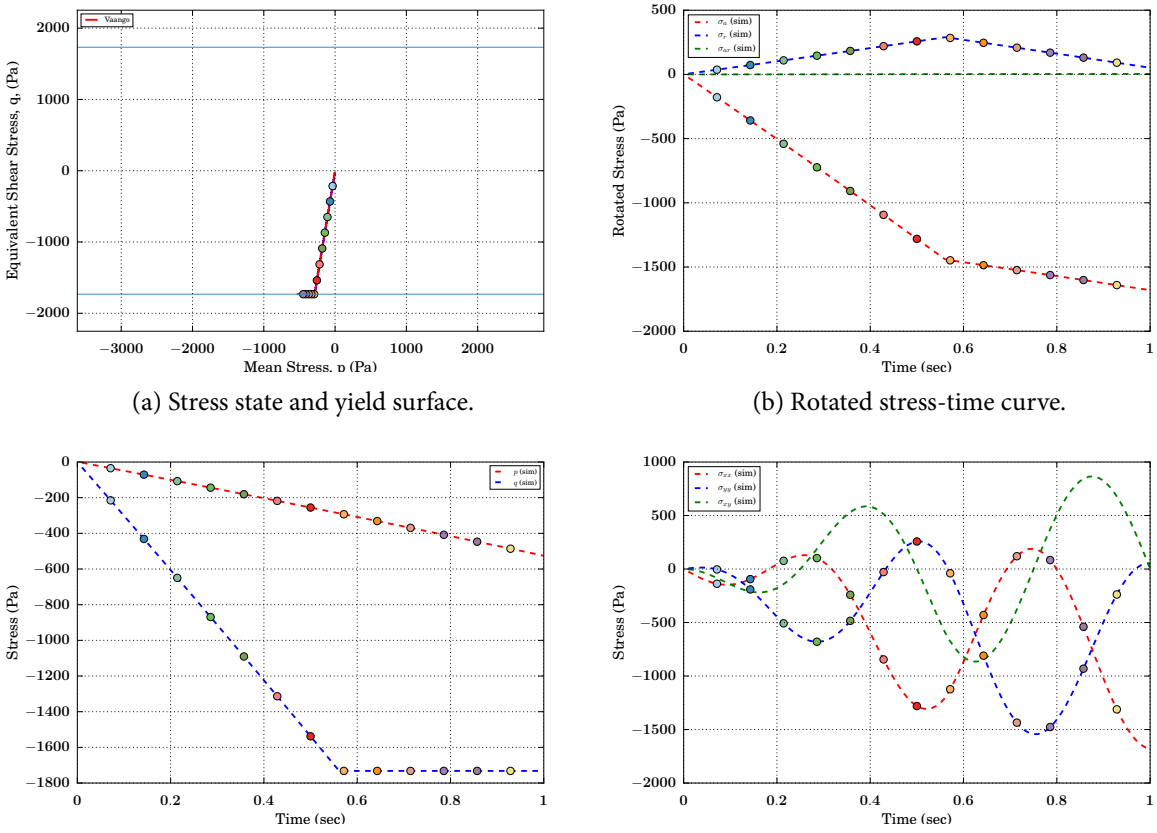
```
{"Vaango_tabular_data": {
  "Meta" : {
    "title" : "Test von Mises yield data"
  },
  "Data" : {
    "Pressure" : [-0.1e10,  0.1e10],
    "SqrtJ2" :   [  1.0e3,  1.0e3]
```

```
  }
}}
```

We drive the simulation using uniaxial strain with rotation using the prescribed deformation file

```
0.0    1.000   0.0   0.0    0.0   1.0   0.0    0.0   0.0   1.0     0     0 0 1
1.0    0.900   0.0   0.0    0.0   1.0   0.0    0.0   0.0   1.0   360     0 0 1
```

After running the simulation, we get the response shown in Figure 5.2.



(a) Stress state and yield surface.

(b) Rotated stress-time curve.

(c) Mean and deviatoric stress.

(d) Unrotated stress-time curve.

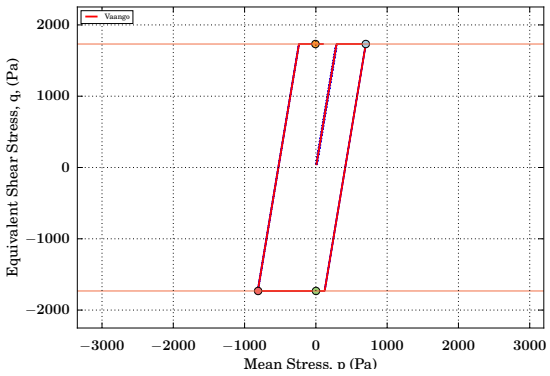
Figure 5.2: Stress evolution for J_2 plasticity with linear elasticity under uniaxial strain compression with rotation.

von Mises plasticity with linear elasticity loading-unloading

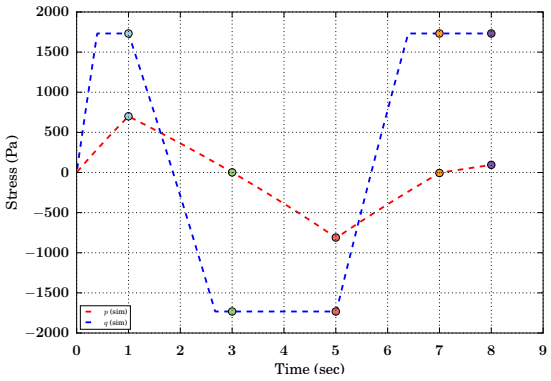
If we change the prescribed deformation to loading-unloading, using the input deformation

```
0  1.0  0  0  0  1  0  0  0  1  0  0  0  0
1  1.15  0  0  0  1  0  0  0  1  0  0  0  0
3  1.0  0  0  0  1  0  0  0  1  0  0  0  0
5  0.85  0  0  0  1  0  0  0  1  0  0  0  0
7  1.0  0  0  0  1  0  0  0  1  0  0  0  0
8  1.02 0  0  0  1  0  0  0  1  0  0  0  0
```

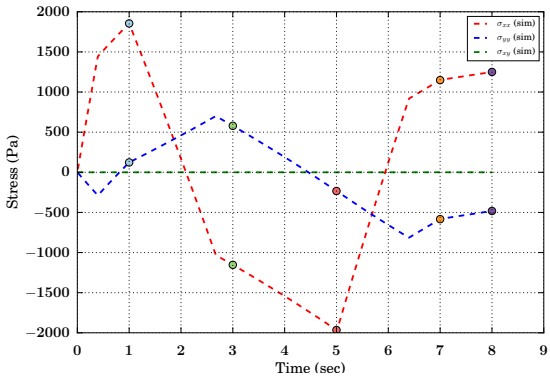
we get the response shown in Figure 5.3



(a) Stress state and yield surface.



(b) Mean and deviatoric stress vs. time.



(c) Stress versus time.

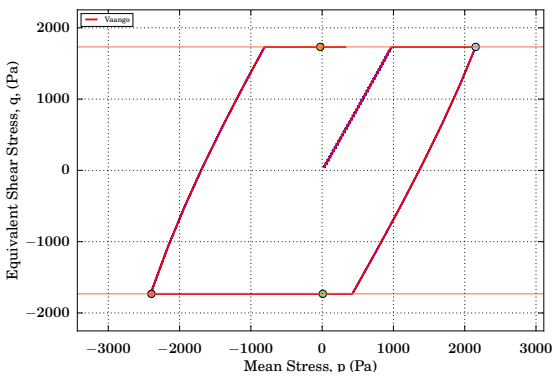
Figure 5.3: Stress evolution for J_2 plasticity with linear elasticity under uniaxial strain loading and unloading.

von Mises plasticity with nonlinear elasticity loading-unloading

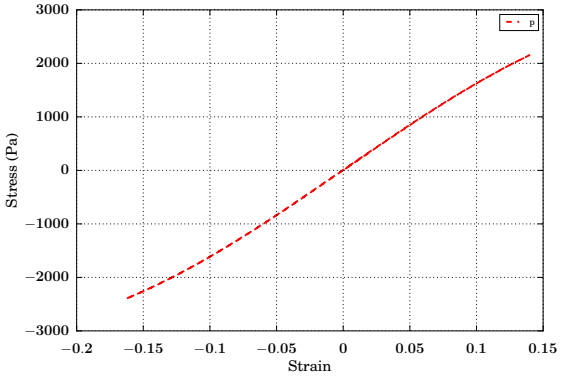
If we replace the linear elastic model with a nonlinear one:

```
{"Vaango_tabular_data": {
  "Meta" : {
    "title" : "Test nonlinear elastic data"
  },
  "Data" : {
    "PlasticStrainVol" : [-0.02, 0.02],
    "Data" : [{
      "TotalStrainVol" : [-0.295000, -0.276034, -0.257069, -0.238103, -0.219138, -0.200172, -0.181207, -0.162241, -0.143276, -0.124310, -0.105345, -0.086379, -0.067414, -0.048448, -0.029483, -0.010517, 0.008448,0.027414, 0.046379, 0.065345, 0.084310, 0.103276, 0.122241, 0.141207, 0.160172, 0.179138, 0.198103, 0.217069, 0.236034, 0.255000],
      "Pressure" : [-3000.000000, -2982.413871, -2929.861667, -2842.959514, -2722.726259, -2570.571529, -2388.279197, -2177.986476, -1942.158854, -1683.561196, -1405.225322, -1110.414466, -802.585016, -485.345990, -162.416726, 162.416726, 485.345990, 802.585016, 1110.414466, 1405.225322, 1683.561196, 1942.158854, 2177.986476, 2388.279197, 2570.571529, 2722.726259, 2842.959514, 2929.861667, 2982.413871, 3000.000000]
    }, {
      "TotalStrainVol" : [-0.255000, -0.236034, -0.217069, -0.198103, -0.179138, -0.160172, -0.141207, -0.122241, -0.103276, -0.084310, -0.065345, -0.046379, -0.027414, -0.008448, 0.010517, 0.029483, 0.048448, 0.067414, 0.086379, 0.105345, 0.124310, 0.143276, 0.162241, 0.181207, 0.200172, 0.219138, 0.238103, 0.257069, 0.276034, 0.295000],
      "Pressure" : [-3000.000000, -2982.413871, -2929.861667, -2842.959514, -2722.726259, -2570.571529, -2388.279197, -2177.986476, -1942.158854, -1683.561196, -1405.225322, -1110.414466, -802.585016, -485.345990, -162.416726, 162.416726, 485.345990, 802.585016, 1110.414466, 1405.225322, 1683.561196, 1942.158854, 2177.986476, 2388.279197, 2570.571529, 2722.726259, 2842.959514, 2929.861667, 2982.413871, 3000.000000]
    }]
  }
}}
```

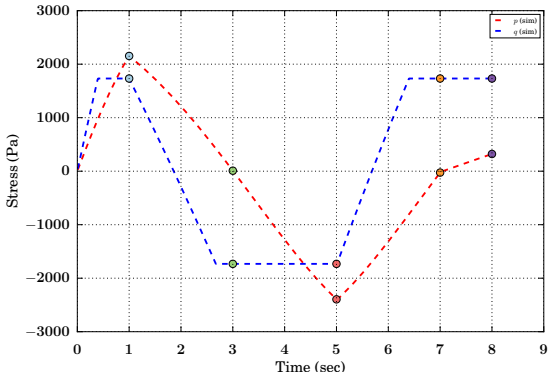
we get the response shown in Figure 5.4



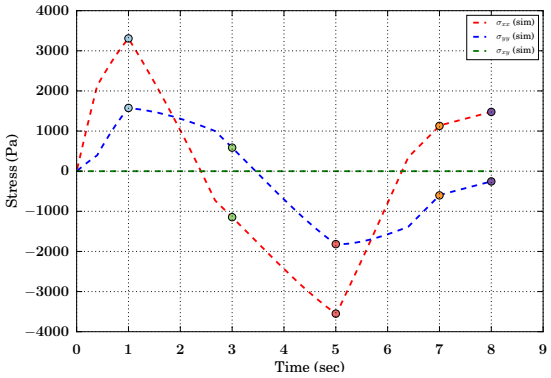
(a) Stress state and yield surface.



(b) Stress vs. strain.



(c) Mean and deviatoric stress vs. time.



(d) Stress versus time.

Figure 5.4: Stress evolution for J_2 plasticity with nonlinear elasticity under uniaxial strain loading and unloading.

Linear Drucker-Prager plasticity with nonlinear elasticity loading-unloading

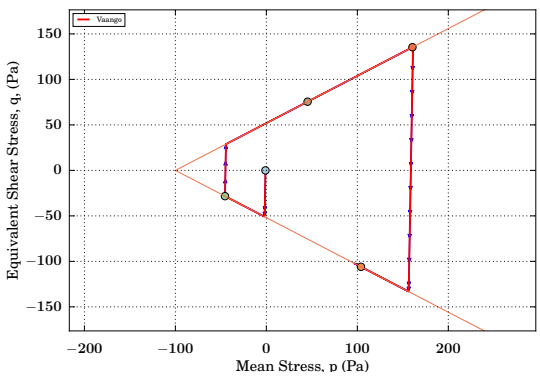
If we change the yield function to a linear Drucker-Prager type model:

```
{"Vaango_tabular_data": {
  "Meta" : {
    "title" : "Test linear Drucker-Prager yield data"
  },
  "Data" : {
    "Pressure" : [-0.1e3,  0.1e10],
    "SqrtJ2" :   [  0,  3.0e8]
  }
}}
```

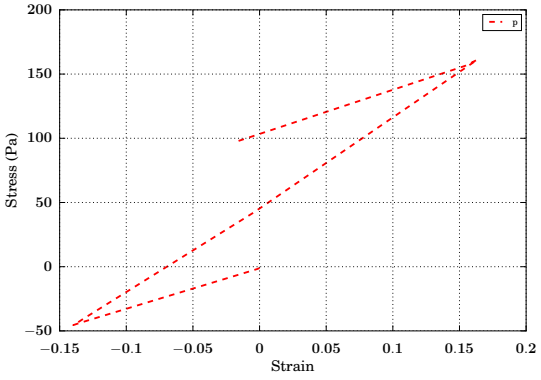
and use the nonlinear elastic model:

```
{"Vaango_tabular_data": {
  "Meta" : {
    "title" : "Test nonlinear elastic data"
  },
  "Data" : {
    "PlasticStrainVol" : [-2.5, 2.5],
    "Data" : [{
      "TotalStrainVol" : [-15.000000, -14.310345, -13.620690, -12.931034, -12.241379, -11.551724, -10.862069, -10.172414, -9.482759, -8.793103, -8.103448, -7.413793, -6.724138, -6.034483, -5.344828, -4.655172, -3.965517, -3.275862, -2.586207, -1.896552, -1.206897, -0.517241, 0.172414, 0.862069, 1.551724, 2.241379, 2.931034, 3.620690, 4.310345, 5.000000],
      "Pressure" : [-2959.842894, -2943.464146, -2920.494168, -2888.366995, -2843.600634, -2781.548344, -2696.156593, -2579.811517, -2423.426002, -2217.004505, -1950.975669, -1618.496214, -1218.556435, -759.014896, -257.981931, 257.981931, 759.014896, 1218.556435, 1618.496214, 1950.975669, 2217.004505, 2423.426002, 2579.811517, 2696.156593, 2781.548344, 2843.600634, 2888.366995, 2920.494168, 2943.464146, 2959.842894]
    }, {
      "TotalStrainVol" : [-5.000000, -4.310345, -3.620690, -2.931034, -2.241379, -1.551724, -0.862069, -0.172414, 0.517241, 1.206897, 1.896552, 2.586207, 3.275862, 3.965517, 4.655172, 5.344828, 6.034483, 6.724138, 7.413793, 8.103448, 8.793103, 9.482759, 10.172414, 10.862069, 11.551724, 12.241379, 12.931034, 13.620690, 14.310345, 15.000000],
      "Pressure" : [-2959.842894, -2943.464146, -2920.494168, -2888.366995, -2843.600634, -2781.548344, -2696.156593, -2579.811517, -2423.426002, -2217.004505, -1950.975669, -1618.496214, -1218.556435, -759.014896, -257.981931, 257.981931, 759.014896, 1218.556435, 1618.496214, 1950.975669, 2217.004505, 2423.426002, 2579.811517, 2696.156593, 2781.548344, 2843.600634, 2888.366995, 2920.494168, 2943.464146, 2959.842894]
    }]
  }
}}
```

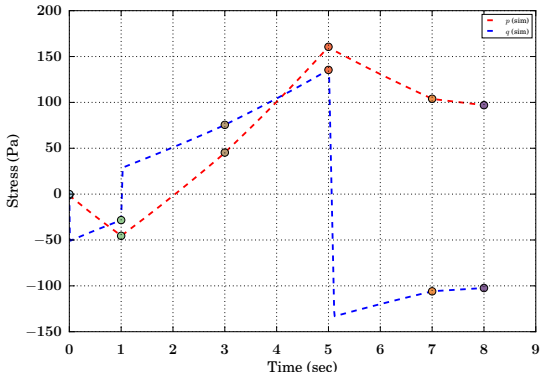
we get the response shown in Figure 5.5



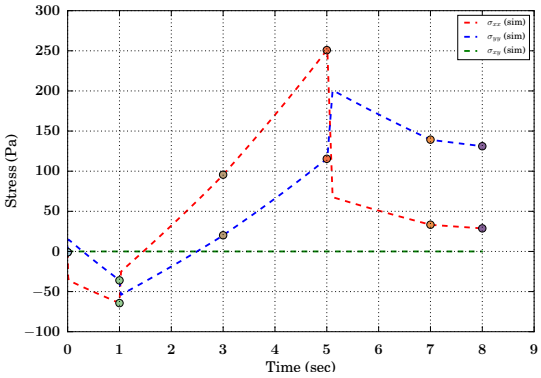
(a) Stress state and yield surface.



(b) Stress vs. strain.



(c) Mean and deviatoric stress vs. time.



(d) Stress versus time.

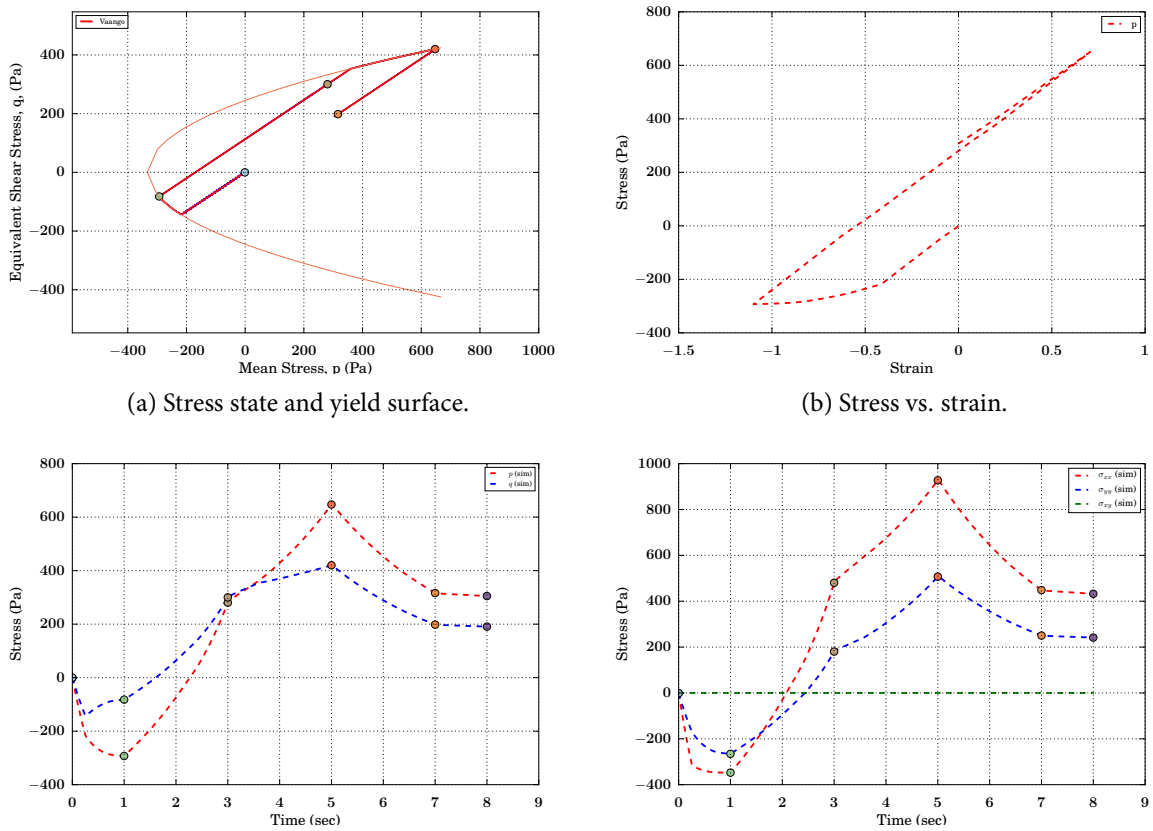
Figure 5.5: Stress evolution for Drucker-Prager plasticity with nonlinear elasticity under uniaxial strain loading and unloading.

Nonlinear Drucker-Prager plasticity with nonlinear elasticity loading-unloading

Finally, if we change the yield function to a nonlinear Drucker-Prager type model:

```
{"Vaango_tabular_data": {
  "Meta" : {
    "title" : "Test nonlinear Drucker-Prager yield data"
  },
  "Data" : {
    "Pressure" : [-333.333333, -298.850575, -264.367816, -229.885057, -195.402299, -160.
        919540, -126.436782, -91.954023, -57.471264, -22.988506, 11.494253, 45.977011, 8
        0.459770, 114.942529, 149.425287, 183.908046, 218.390805, 252.873563, 287.356322
        , 321.839080, 356.321839, 390.804598, 425.287356, 459.770115, 494.252874, 528.73
        5632, 563.218391, 597.701149, 632.183908, 666.666667],
    "SqrtJ2" :   [0.000000, 45.485883, 64.326752, 78.783860, 90.971765, 101.709526, 111.
        417203, 120.344334, 128.653504, 136.457648, 143.838990, 150.859606, 157.567719,
        164.001682, 170.192589, 176.166066,181.943530, 187.543098, 192.980256, 198.26836
        6, 203.419051, 208.442500, 213.347701, 218.142630, 222.834406, 227.429413, 231.9
        33403, 236.351579, 240.688667, 244.948974]
  }
}}
```

we get the response shown in Figure 5.6



(a) Stress state and yield surface.

(b) Stress vs. strain.

(c) Mean and deviatoric stress vs. time.

(d) Stress versus time.

Figure 5.6: Stress evolution for Drucker-Prager plasticity with nonlinear elasticity under uniaxial strain loading and unloading.

5.8.4 Python script for converting CSV to JSON

A JSON format is used for the input data in tabular models. The python script below shows an example that can be modified for converting data into the JSON format used by VAANGO . If the input data are in an

Excel spreadsheet, they have to be saved as CSV (comma separated values) before they can be translated into JSON.

```
#  Compression is positive
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import json
import PolylineIntersection as pl
import imp
pl = imp.reload(pl)

# Load the CSV data into Pandas dataframes
hydrostat = pd.read_csv("./DrySand_Hydrostat.csv", header=0, skiprows=7)
data_09 = pd.read_csv("./DrySand_LoadUnload_09.csv", header=0, skiprows=4)
data_18 = pd.read_csv("./DrySand_LoadUnload_18.csv", header=0, skiprows=4)
data_27 = pd.read_csv("./DrySand_LoadUnload_27.csv", header=0, skiprows=4)
data_36 = pd.read_csv("./DrySand_LoadUnload_36.csv", header=0, skiprows=4)
data_45 = pd.read_csv("./DrySand_LoadUnload_45.csv", header=0, skiprows=4)

# Rename the columns of each dataframe
column_names = ["TotalStrainVol", "Pressure", "", "", "", ""]
hydrostat.columns = column_names
data_09.columns = column_names
data_18.columns = column_names
data_27.columns = column_names
data_36.columns = column_names
data_45.columns = column_names

# Convert percent into strain, MPa to Pa
strain_fac = 0.01
pressure_fac = 1.0e6
hydrostat.TotalStrainVol *= strain_fac
hydrostat.Pressure *= pressure_fac
data_09.TotalStrainVol *= strain_fac
data_09.Pressure *= pressure_fac
data_18.TotalStrainVol *= strain_fac
data_18.Pressure *= pressure_fac
data_27.TotalStrainVol *= strain_fac
data_27.Pressure *= pressure_fac
data_36.TotalStrainVol *= strain_fac
data_36.Pressure *= pressure_fac
data_45.TotalStrainVol *= strain_fac
data_45.Pressure *= pressure_fac

# Find the point at which unloading begins
p_max_09 = max(data_09.Pressure)
p_max_index_09 = data_09.Pressure.values.tolist().index(p_max_09)
p_max_18 = max(data_18.Pressure)
p_max_index_18 = data_18.Pressure.values.tolist().index(p_max_18)
p_max_27 = max(data_27.Pressure)
p_max_index_27 = data_27.Pressure.values.tolist().index(p_max_27)
p_max_36 = max(data_36.Pressure)
p_max_index_36 = data_36.Pressure.values.tolist().index(p_max_36)
p_max_45 = max(data_45.Pressure)
p_max_index_45 = data_45.Pressure.values.tolist().index(p_max_45)
p_max_index_00 = (np.abs(p_max_09 - hydrostat.Pressure.values)).argmin()
p_max_00 = hydrostat.Pressure.values[p_max_index_00]

# Create separate dataframes for the unload data
data_09_unload = data_09[p_max_index_09:].copy()
data_18_unload = data_18[p_max_index_18:].copy()
data_27_unload = data_27[p_max_index_27:].copy()
data_36_unload = data_36[p_max_index_36:].copy()
data_45_unload = data_45[p_max_index_45:].copy()

# Find plastic strains by intersecting the unload data with the pressure axis
pressure_axis = ((-1, 0),(1, 0))
poly_09_unload = list(data_09_unload[['TotalStrainVol', 'Pressure']].apply(tuple, axis=1
    ))
line_09_unload = (poly_09_unload[-1], poly_09_unload[-2])
plastic_strain_09 = pl.line_intersection(pressure_axis, line_09_unload)[0]
poly_18_unload = list(data_18_unload[['TotalStrainVol', 'Pressure']].apply(tuple, axis=1
```

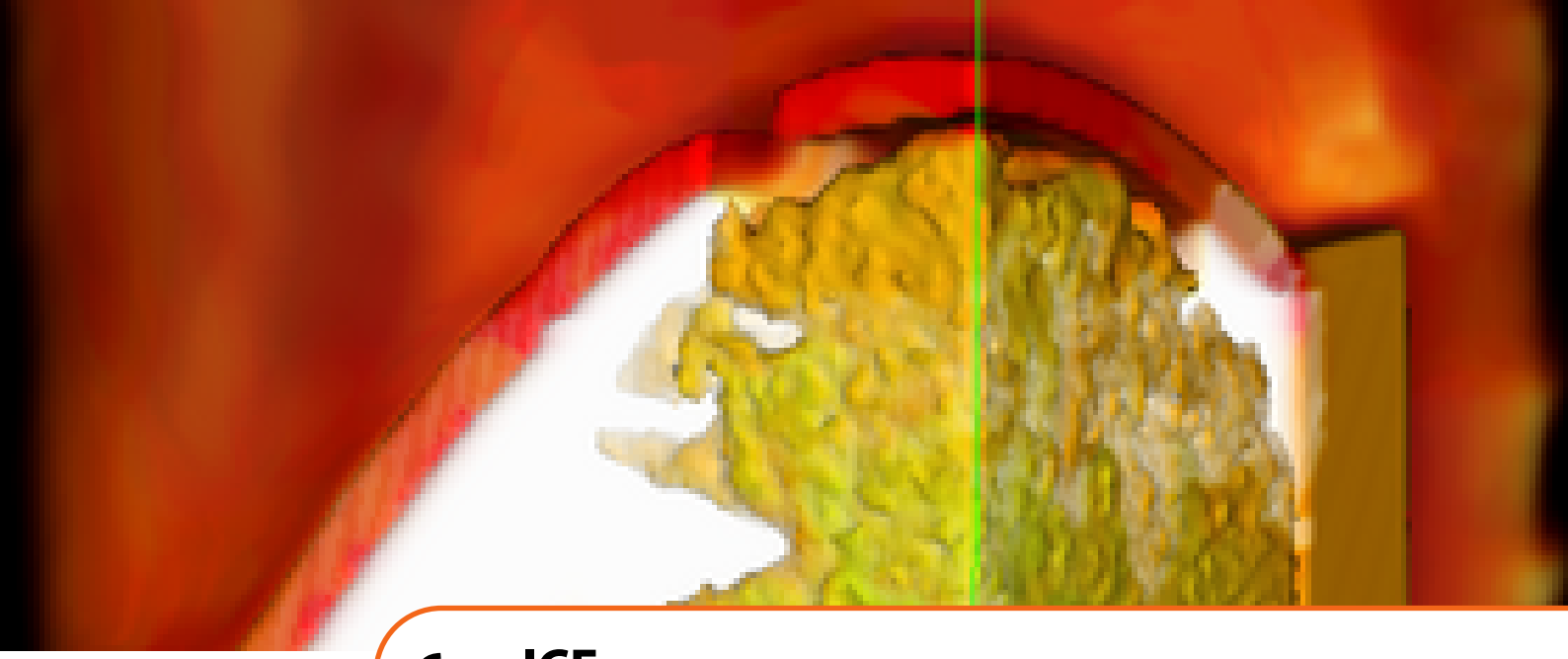
```
    ))
line_18_unload = (poly_18_unload[-1], poly_18_unload[-2])
plastic_strain_18 = pl.line_intersection(pressure_axis, line_18_unload)[0]
poly_27_unload = list(data_27_unload[['TotalStrainVol', 'Pressure']].apply(tuple, axis=1
    ))
line_27_unload = (poly_27_unload[-1], poly_27_unload[-2])
plastic_strain_27 = pl.line_intersection(pressure_axis, line_27_unload)[0]
poly_36_unload = list(data_36_unload[['TotalStrainVol', 'Pressure']].apply(tuple, axis=1
    ))
line_36_unload = (poly_36_unload[-1], poly_36_unload[-2])
plastic_strain_36 = pl.line_intersection(pressure_axis, line_36_unload)[0]
poly_45_unload = list(data_45_unload[['TotalStrainVol', 'Pressure']].apply(tuple, axis=1
    ))
line_45_unload = (poly_45_unload[-1], poly_45_unload[-2])
plastic_strain_45 = pl.line_intersection(pressure_axis, line_45_unload)[0]

# Compute the volumetric elastic strains by subtracting plastic strains from total
    strains
data_09_unload.TotalStrainVol -= plastic_strain_09
data_18_unload.TotalStrainVol -= plastic_strain_18
data_27_unload.TotalStrainVol -= plastic_strain_27
data_36_unload.TotalStrainVol -= plastic_strain_36
data_45_unload.TotalStrainVol -= plastic_strain_45

# Reverse the order of the unload data to create elastic loading curves
data_00_load = hydrostat[:p_max_index_00]
data_09_load = data_09_unload.sort_index(ascending=False)
data_18_load = data_18_unload.sort_index(ascending=False)
data_27_load = data_27_unload.sort_index(ascending=False)
data_36_load = data_36_unload.sort_index(ascending=False)
data_45_load = data_45_unload.sort_index(ascending=False)

# Simplify the loading dataframes and remove duplicates (if any)
data_00_all = data_00_load[['TotalStrainVol', 'Pressure']].copy().drop_duplicates()
data_09_all = data_09_load[['TotalStrainVol', 'Pressure']].copy().drop_duplicates()
data_18_all = data_18_load[['TotalStrainVol', 'Pressure']].copy().drop_duplicates()
data_27_all = data_27_load[['TotalStrainVol', 'Pressure']].copy().drop_duplicates()
data_36_all = data_36_load[['TotalStrainVol', 'Pressure']].copy().drop_duplicates()
data_45_all = data_45_load[['TotalStrainVol', 'Pressure']].copy().drop_duplicates()

# Compute the total volumetric strain
data_09_all.TotalStrainVol += plastic_strain_09
data_18_all.TotalStrainVol += plastic_strain_18
data_27_all.TotalStrainVol += plastic_strain_27
data_36_all.TotalStrainVol += plastic_strain_36
data_45_all.TotalStrainVol += plastic_strain_45

# Create an extra data set for tension interpolations
data_09_tension_all = data_00_all.copy()
data_09_tension_all.TotalStrainVol -= plastic_strain_09

# Visual check of data set
fig = plt.figure(figsize=(6,6))
ax = fig.add_subplot(111)
plt.plot(data_09_tension_all.TotalStrainVol, data_09_tension_all.Pressure, 'C7')
plt.plot(data_00_all.TotalStrainVol, data_00_all.Pressure, 'C0')
plt.plot(data_09_all.TotalStrainVol, data_09_all.Pressure, 'C1')
plt.plot(data_18_all.TotalStrainVol, data_18_all.Pressure, 'C2')
plt.plot(data_27_all.TotalStrainVol, data_27_all.Pressure, 'C3')
plt.plot(data_36_all.TotalStrainVol, data_36_all.Pressure, 'C4')
plt.plot(data_45_all.TotalStrainVol, data_45_all.Pressure, 'C5')
plt.grid(True)
plt.axis([-0.1, 0.4, 0, 2e9])
plt.xlabel('Engineering volumetric strain', fontsize=16)
plt.ylabel('Pressure (Pa)', fontsize=16)
fig.savefig('Fox_DrySand_ElasticData.png')

# Create data that can be written as JSON
elastic_data_json_09_t = {}
elastic_data_json_09_t["TotalStrainVol"] = data_09_tension_all.TotalStrainVol.values.
    tolist()
elastic_data_json_09_t["Pressure"] = data_09_tension_all.Pressure.values.tolist()
#json.dumps(elastic_data_json_09_t)
```

```
elastic_data_json_00 = {}
elastic_data_json_00["TotalStrainVol"] = data_00_all.TotalStrainVol.values.tolist()
elastic_data_json_00["Pressure"] = data_00_all.Pressure.values.tolist()
#json.dumps(elastic_data_json_00)
elastic_data_json_09 = {}
elastic_data_json_09["TotalStrainVol"] = data_09_all.TotalStrainVol.values.tolist()
elastic_data_json_09["Pressure"] = data_09_all.Pressure.values.tolist()
#json.dumps(elastic_data_json_09)
elastic_data_json_18 = {}
elastic_data_json_18["TotalStrainVol"] = data_18_all.TotalStrainVol.values.tolist()
elastic_data_json_18["Pressure"] = data_18_all.Pressure.values.tolist()
#json.dumps(elastic_data_json_18)
elastic_data_json_27 = {}
elastic_data_json_27["TotalStrainVol"] = data_27_all.TotalStrainVol.values.tolist()
elastic_data_json_27["Pressure"] = data_27_all.Pressure.values.tolist()
#json.dumps(elastic_data_json_27)
elastic_data_json_36 = {}
elastic_data_json_36["TotalStrainVol"] = data_36_all.TotalStrainVol.values.tolist()
elastic_data_json_36["Pressure"] = data_36_all.Pressure.values.tolist()
#json.dumps(elastic_data_json_36)
elastic_data_json_45 = {}
elastic_data_json_45["TotalStrainVol"] = data_45_all.TotalStrainVol.values.tolist()
elastic_data_json_45["Pressure"] = data_45_all.Pressure.values.tolist()
#json.dumps(elastic_data_json_45)

# Set up plastic strain list
plastic_strain_data = [-plastic_strain_09, 0, plastic_strain_09, plastic_strain_18,
    plastic_strain_27, plastic_strain_36, plastic_strain_45]

# Set up elastic data list
vaango_elastic_data = {}
vaango_elastic_data["PlasticStrainVol"] = plastic_strain_data
vaango_elastic_data["Data"] = [elastic_data_json_09_t, elastic_data_json_00,
    elastic_data_json_09, elastic_data_json_18, elastic_data_json_27, elastic_data_json_
    36, elastic_data_json_45]

# Write JSON
meta_data_json = {}
meta_data_json["title"] = "Dry sand nonlinear elasticity data"

vaango_data = {}
vaango_data["Meta"] = meta_data_json
vaango_data["Data"] = vaango_elastic_data
json.dumps(vaango_data)

vaango_input_data = {}
vaango_input_data["Vaango_tabular_data"] = vaango_data
json.dumps(vaango_input_data, sort_keys=True)

with open('DrySand_ElasticData.json', 'w') as outputFile:
    json.dump(vaango_input_data, outputFile, sort_keys=True, indent=2)
```

The above script uses an intersection algorithm called "PolylineIntersection.py" which is listed below.

```
def line_intersection(line1, line2):
    xdiff = (line1[0][0] - line1[1][0], line2[0][0] - line2[1][0])
    ydiff = (line1[0][1] - line1[1][1], line2[0][1] - line2[1][1])

    def det(a, b):
        return a[0] * b[1] - a[1] * b[0]

    div = det(xdiff, ydiff)
    if div == 0:
       return None

    d = (det(*line1), det(*line2))
    x = det(d, xdiff) / div
    y = det(d, ydiff) / div
    return (x, y)

def poly_intersection(poly1, poly2):

    for i, p1_first_point in enumerate(poly1[:-1]):
```

```
        p1_second_point = poly1[i + 1]

        for j, p2_first_point in enumerate(poly2[:-1]):
            p2_second_point = poly2[j + 1]

            pt = line_intersection((p1_first_point, p1_second_point), (p2_first_point, p
                2_second_point))
            t = (p2_second_point[0] - pt[0])/(p2_second_point[0] - p2_first_point[0])
            print(j, p1_first_point, p2_first_point, p2_second_point, pt, t)
            if pt and t >= 0 and t <= 1:
                return [True, pt]

    return False
```



6 — ICE

The theory of the ICE-CFD approach can be found in the VAANGO Theory Manual.

6.1 Uintah Specification

6.1.1 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.

6.1.2 Semi-Implicit Pressure Solve

The equation for the change in the pressure field ΔP during a given timestep is given by

$$\frac{dP}{dt} = \frac{\sum_{m=1}^{N} \frac{\dot{m}}{V\rho_m^o} - \sum_{m=1}^{N} \nabla \cdot \hat{\theta_m} \vec{U_m}^{*^f}}{\sum_{m=1}^{N} \frac{\theta_m}{\rho_m^o c_m^2}} \tag{6.1}$$

which can be written in matrix form $Ax = b$ and solved with a linear solver. Details on the notation, discretization of Eq. 6.1 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 [56]. 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

6.1.3 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>
```

6.1.4 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>
```

6.1.5 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

$$p = (\gamma - 1) c_v \rho T \tag{6.2}$$

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 [57, 58]

```
<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 [59] 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

$$p(v - b) = RT \tag{6.3}$$

where b corresponds to the volume occupied by the molecules themselves [60]. 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

$$p = \frac{(\gamma - 1)c_v T}{v - b} - \frac{a}{(v + 3.0b)(v - 0.5b)} \tag{6.4}$$

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 [61] 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>
```

6.1.6 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:

$$C_v(T) = \frac{T^n}{\sum_{i=0}^{n} a_i * T^i}$$

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>
```

6.1.7 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>
```

6.1.8 BoundaryConditions

Boundary conditions must be specified on each face of the computational domain $(x^-, x^+, y^-, y^+, z^-, z^+)$ for the variables P, $\mathbf{u}$, $\mathbf{T}$, ρ, $\mathbf{v}$ 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,

$$q[\text{boundary cell}] = q[\text{interior cell}] - value * dn; \tag{6.5}$$

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

$$q[\text{boundary cell}] = value; \tag{6.6}$$

```
<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

$$p = p_{reference} + Asin(\omega t) \tag{6.7}$$

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 [62]

```
<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.

6.1.9 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.

6.1.10 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

```
1  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	L_3/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_frac_CC`, 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.

6.1.11 XML tag description

XML tag	Type	Dimensions	Description
cfl	double		Courant Number.
gravity	Vector	$[L/t^2]$	gravitational acceleration, $\vec{g}$.
global material properties			
dynamic_viscosity	double	$[M/Lt]$	viscosity, μ.
thermal_conditucivity	double	$[ML/t^3T]$	thermal conductivity, k
specific_heat	double	$[L^2/t^2T]$	c_p
gamma	double		ratio of specific heats, γ.
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, ρ.
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.

6.2 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` – `np1sus` – `solverhypreinputs/UintahRelease/ICE/CouettePoiseuille.ups`

Postprocessing command:

`inputs/UintahRelease/ICE/compare_CouettePoiseuille.m` – `udaCouette` – `Poiseuille.uda`

You must edit `compare_CouettePoiseuille.m` and set `wallVel` = 0. This will generate a postscript file Couette-Poiseuille.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 6.1 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 $\|u - u_{exact}\|$.

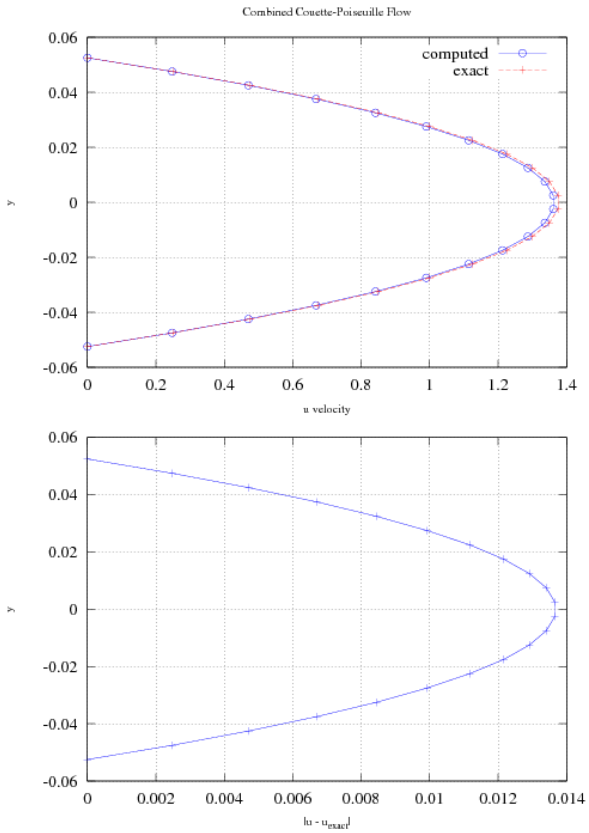


Figure 6.1: Comparison of u velocity $t = 15sec$

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

$$\mu \frac{d^2u}{dy^2} = \frac{dp}{dx} = \text{constant} \tag{6.8}$$

subject to the no slip boundary condition $u(\pm h)$ = wall velocity, where h is half the height of the channel [63].

Simulation Specifics

Component used: ICE
Input file name: `CouettePoiseuille.ups`
Command used to run input file:
`mpirun – np1sus – solverhypreinputs/UintahRelease/ICE/CouettePoiseuille.ups`
Postprocessing command:
`inputs/UintahRelease/ICE/compare_CouettePoiseuille.m – udaCouette – 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)

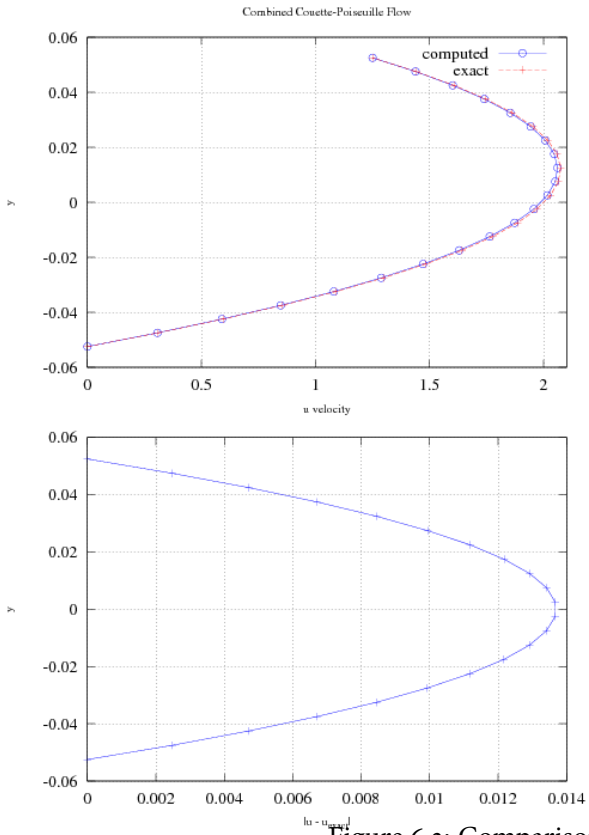


Figure 6.2: Comparison of u velocity $t = 15sec$

Example Runtimes:
8ish minutes (1 processor, 2.66 GHz Xeon)
Physical time simulated: 15 sec.

Results

Figure 6.2 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 $\|u - u_{exact}\|$.

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 [64–66]. 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: `susinputs/UintahRelease/ICE/shockTube.ups`
Postprocessing command:
`scripts/ICE/plot_shockTube_1LshockTube.uday`
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 6.3 shows a comparison of the exact versus simulated results at time $t = 5msec$.

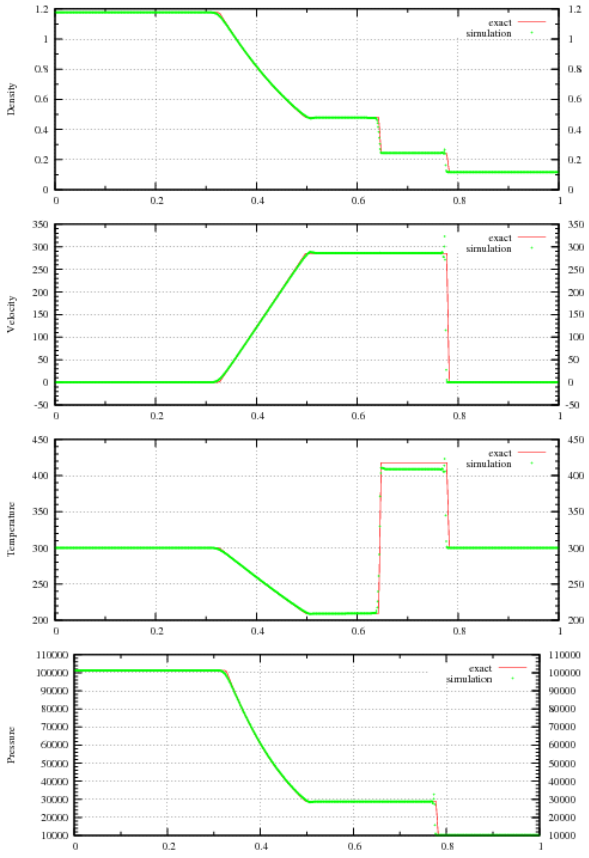


Figure 6.3: Shock tube results at time $t = 5msec$

Shock Tube with Adaptive Mesh Refinement

Simulation Specifics

Component used: ICE
Input file name: `shockTube_AMR.ups`
Command used to run input file:
`susinputs/UintahRelease/ICE/shockTube_AMR.ups`
Postprocessing command:
`../../src/scripts/ICE/plot_shockTube_AMRshockTube_AMR.uday`
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 6.4 shows a comparison of the exact versus simulated results at time $t = 5msec$.

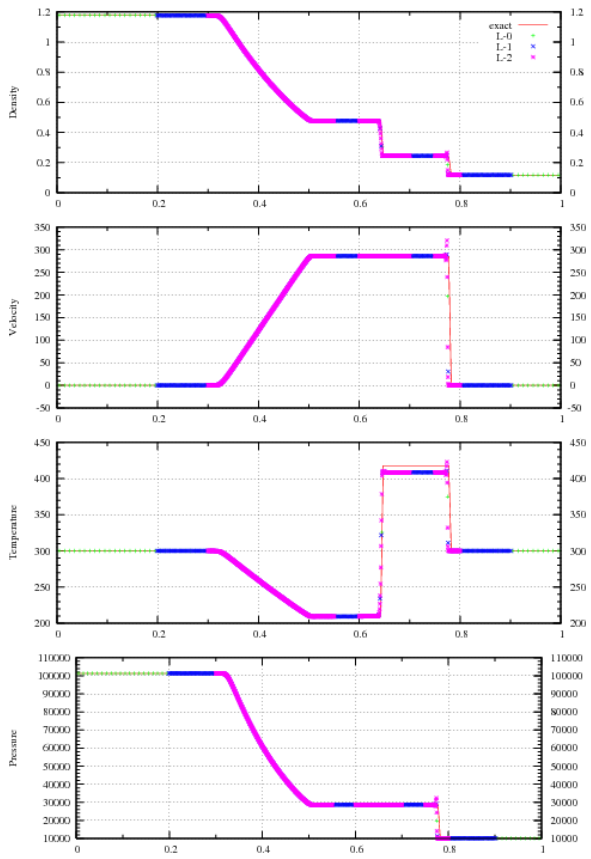


Figure 6.4: Shock tube results at time $t = 5msec$

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 [67, 68]. 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, ρ 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 – np5susinputs/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.

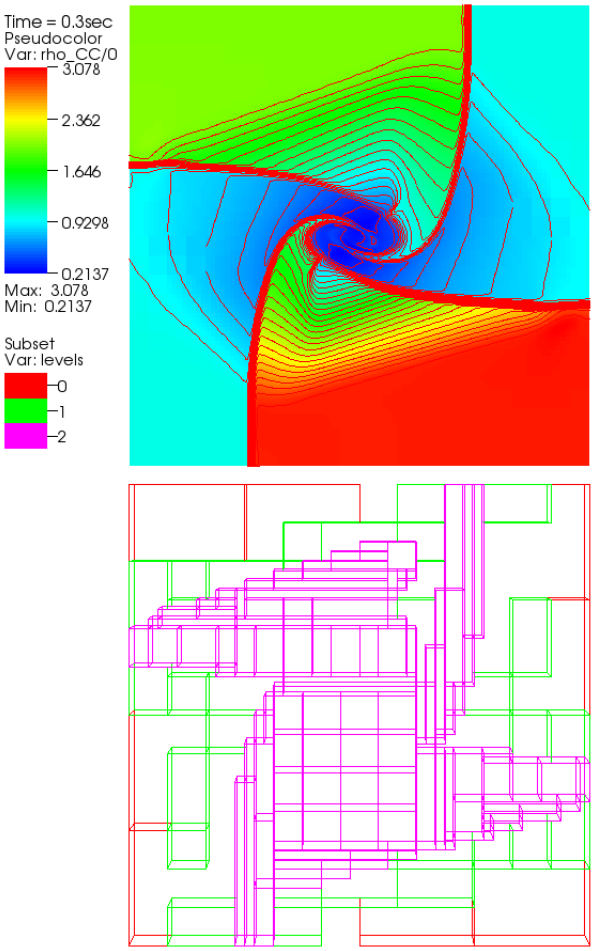


Figure 6.5: Contour plot of density for the 2D Riemann problem at time $t = 0.3sec$. Bottom plot shows the outline of the patches on the 3 levels.

Results

Figure 6.5 shows a flood and line contour plot(s) of the density of the gas at 0.03sec.

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 [66]. 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 – np4susinputs/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 6.6 and 6.7 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

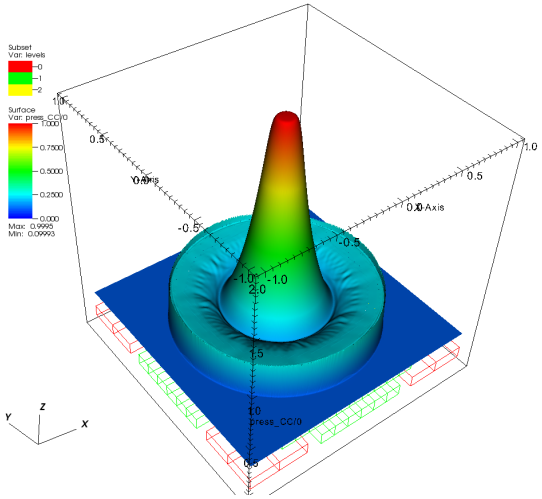


Figure 6.6: Pressure field at $t = 0.25$

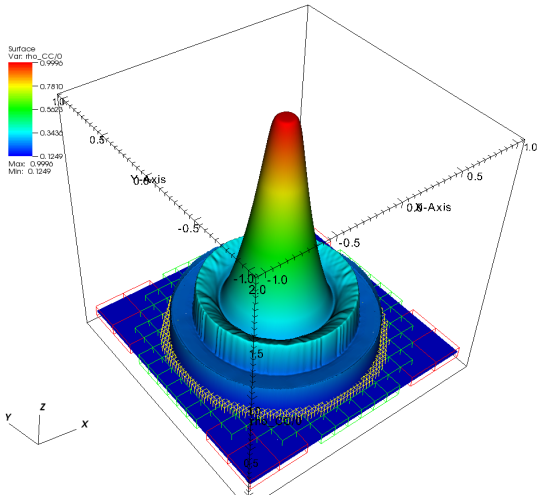


Figure 6.7: Density field at time $t = 0.25$

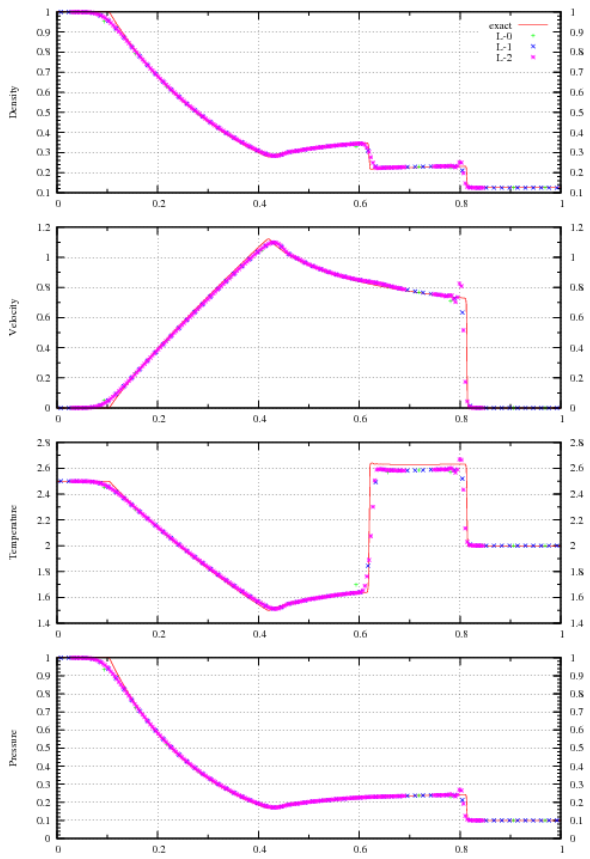


Figure 6.8: t = 0.25

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++ [59] 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 – np4 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:

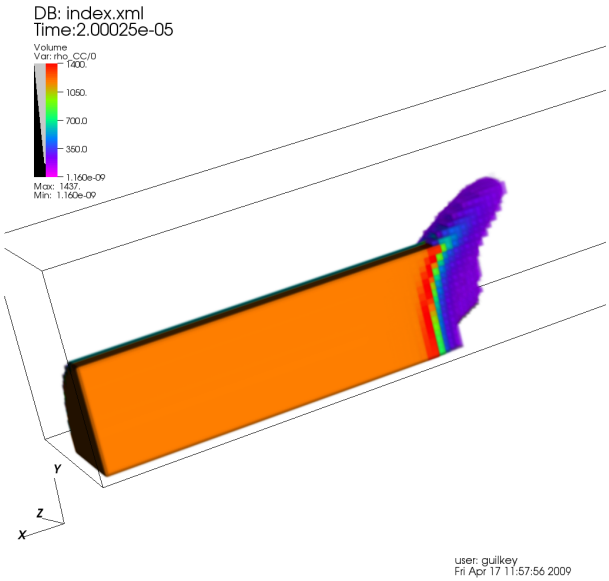


Figure 6.9: Density of reactant material.

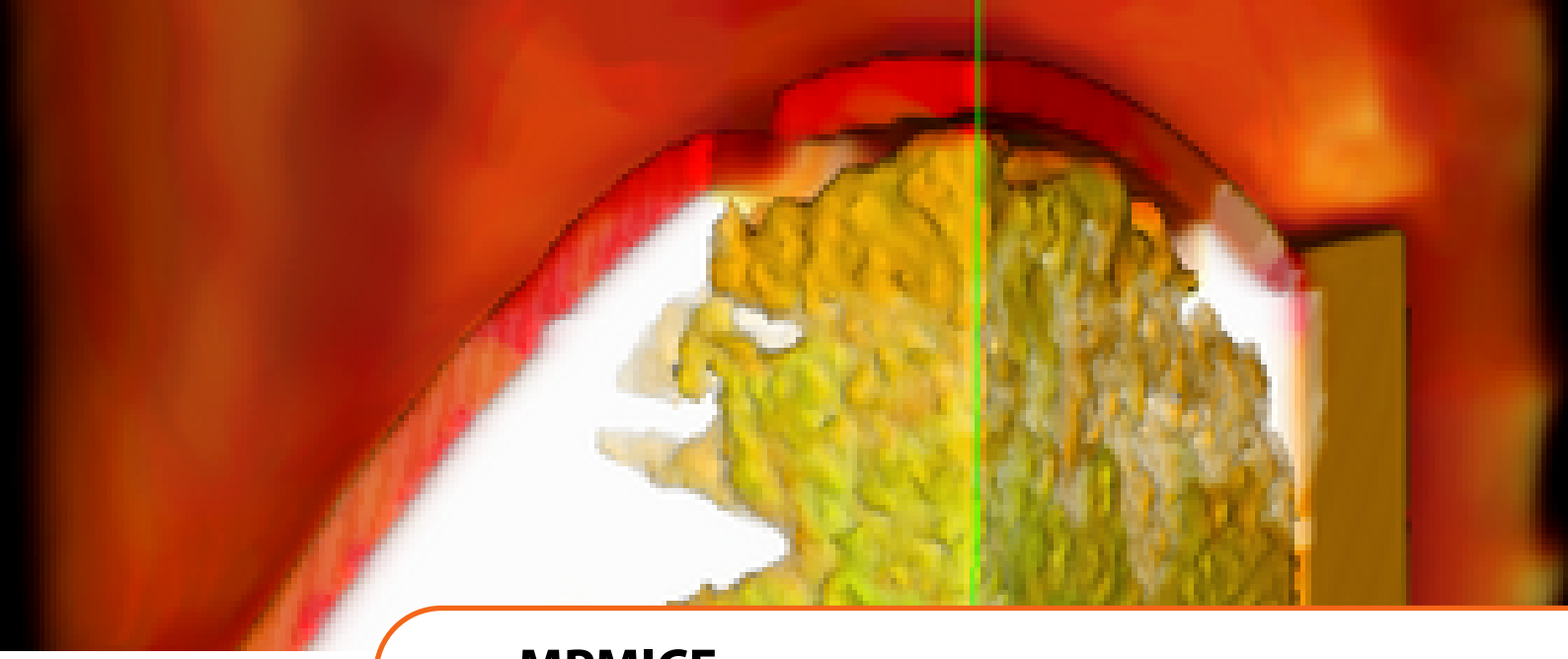
0.0005 x 0.0005 x 0.0005 (Level 0)

Example Runtimes:
1.5 hours (4 processor, 3.16 GHz Xeon)

Physical time simulated: 20.0 μseconds

Results

Figure 6.9 shows a volume rendering of the density of the reactant. Note the curvature of the reaction zone.



7 — MPMICE

7.1 Introduction

MPMICE is a marriage of the multi-material ICE method, described in Section 6 and MPM, described in Section 4. 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.

7.2 Theory - Algorithm Description

At this time, the reader is directed to the manuscript by Guilkey, Harman and Banerjee [69] for the theoretical and algorithmic description of the method.

7.3 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:

$$\frac{d\alpha}{dt} = k(T)f(\alpha) \tag{7.1}$$

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:

$$k(T) = Ae^{\frac{-E_a}{RT}} \tag{7.2}$$

and:

$$k(T) = AT^b e^{\frac{-Ea}{RT}} \tag{7.3}$$

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 [70] and [71].

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.

7.4 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.

7.4.1 Simple Burn

Simple Burn, as the name implies, is a simple model of combustion of HMX based on the rate equation:

$$\dot{m} = AP^{0.778} \tag{7.4}$$

Where $\dot{m}$ is the mass flux, P is the pressure and n is the pressure dependence coefficient. The pressure coefficient in Equation (7.4) 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 6.1.4 and 4.2.4 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.

7.4.2 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 [72]. 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:

$$A(solid) \rightarrow B(gas) \tag{7.5}$$

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 $E_g = 0$:

$$B(gas) + M(gas) \rightarrow C(gas) + M(gas) \tag{7.6}$$

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:

$$\dot{m}\left(T_s\right) = \sqrt{\frac{\kappa_c \rho_c A_c R T_s^2 \exp\left(\frac{-E_c}{RT_s}\right)}{C_p E_c \left(T_s - T_0 - \frac{Q_c}{2C_p}\right)}} \tag{7.7}$$

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:

$$T_s(\dot{m}, P) = T_0 + \frac{Q_c}{C_p} + \frac{Q_g}{Cp\left(1 + \frac{x_g\left(\dot{m}, P\right)}{x_{cd}(\dot{m})}\right)} \tag{7.8}$$

x_g in the third term of Equation (7.8) is the flame standoff distance, computed from:

$$x_g\left(\dot{m}, P\right) = \frac{2x_{cd}\left(\dot{m}\right)}{\sqrt{1 + D_a\left(\dot{m}, P\right)} - 1} \tag{7.9}$$

where x_{cd} and D_a are the convective-diffusive length and Damkohler number, respectively:

$$x_{cd}\left(\dot{m}\right) = \frac{\kappa_g}{\dot{m} C_p} \tag{7.10}$$

$$D_a\left(\dot{m}, P\right) = \frac{4 B_g M C_p P^2}{R^2 \kappa_g} x_{cd}\left(\dot{m}\right)^2 \tag{7.11}$$

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 $\dot{m}$, Δt of the timestep and the calculated surface area, found using:

$$A = \frac{\delta x \delta y \delta z}{\delta x |g_x| + \delta y |g_y| + \delta z |g_z|} \tag{7.12}$$

where δx, δy, and δz are the dimensions of the cell and components of $\vec{g}$ are the normalized density gradients of the particle mass in a cell. A more thorough examination of Steady Burn can be read about in [73].

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 (κ_g)	$0.07 W/(m \times K)$
<HeatConductCondPh>	double	Thermal conductivity of condensed phase (κ_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 ≥ to burn mass	$50000 Pa$
<IgnitionTemp>	double	Temperature cell must have ≥ to be burning	$550 K$

7.4.3 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 (7.7) and (7.8). An expression for the temperature gradient of the solid:

$$\beta = (T_s - T_o)\frac{mc_p}{\kappa_c} \tag{7.13}$$

is reaarranged for $(T_s - T_o)$ and substituted in Equation (7.7) leading to the quadradic equation:

$$\dot{m}^2 - \frac{2\beta\kappa_c}{Q_c}\dot{m} + \frac{2A_c R T_s^2 \kappa_c \rho_c}{E_c Q_C}\exp\left(\frac{-E_c}{RT_s}\right) = 0 \tag{7.14}$$

which allows independent tracking of temperature gradient β 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 β as it approaches the steady burning value. A slow relaxation time for β 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 [73].

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 (κ_g)
<HeatConductCondPh>	double	Thermal conductivity of condensed phase (κ_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 ≠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 ≥ to burn mass
<IgnitionTemp>	double	Temperature cell must be at ≥ to be burning

7.4.4 Ignition & Growth

The Ignition & Growth model created by Lee and Tarver [74] to simulate shock-to-detonation transitions in condensed explosives. The rate equation takes the form:

$$\frac{dF}{dt} = I(1-F)^b \left(\frac{\rho}{\rho_o} - 1 - a\right)^x + G_1(1-F)^c F^d P^y + G_2(1-F)^e F^g P^z \tag{7.15}$$

where F is the extent of reaction in a cell, P is the pressure, ρ and ρ_o 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
<rhoo>	double	The initial density of the explosive
<Eo>	double	The energy of detonation
<ThresholdPressure>	double	Reaction is allowed to occur above this pressure

7.4.5 JWL++

The JWL++ model by Souers et al. [59] is related to the Ignition and Growth model (see Section 7.4.4), but much simplified in its form for ease of rate constant fit. The rate is described by:

$$\frac{dF}{dt} = G(1-F)P^b \tag{7.16}$$

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
<rhoo>	double	The initial density of the explosive
<Eo>	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

7.4.6 DDTo

Deflagration-to-detonation model o (DDTo) 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 7.4.1). Detonation is accounted for by the JWL++ model (see Section 7.4.5) presented by P. Clark Souers [59]. 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. [75] The parameters are presented in the following table.

DDTo 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)
<Eo>	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

7.4.7 DDT1

Deflagration-to-detonation model 1 (DDT1) [76] 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 7.4.2). Detonation is accounted for by the JWL++ model (see Section 7.4.5) presented by P. Clark Souers [59]. 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. [75] 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)
<Eo>	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 (κ_g)
<HeatConductCondPh>	double	Thermal conductivity of condensed phase (κ_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 [72]. 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 7.17 [77], where P is the dimensionless pressure (p/p_o) and S_f is the flame propagation in cm/s. Equation 7.17 is used in determining the induction time as seen by equation 7.18 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.

$$S_f = 0.259P^{0.538} \tag{7.17}$$

$$\tau = \frac{\Delta x A}{S_f} \tag{7.18}$$

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_o)
<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. [77] for experimentally determined values for the explosive PBX9501.

7.5 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. [78] (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

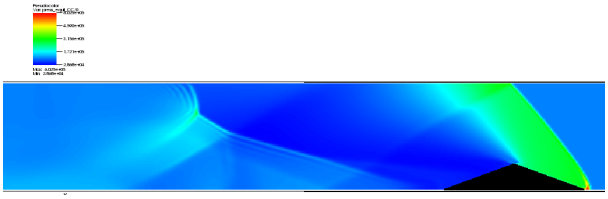


Figure 7.1: 20^o wedge moving at Mach 2.0 through initially stationary air. Contour plot depicts pressure.

Results

Figure 7.1 shows a snapshot of the simulation. Contour plot depicts pressure and reflects the presence of a leading shock and an expansion fan.

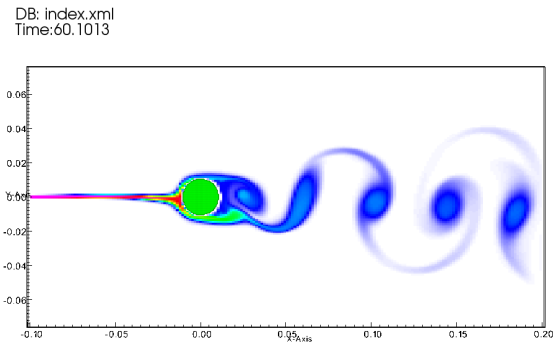


Figure 7.2: Flow over a stationary cylinder, $Re = 700$, a passive scalar is used as a flow marker

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 – np6susinputs/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 7.2 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

```
1  movies/cyl_crossFlow.mpg
```

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:
`susinputs/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 μseconds
Associated visit session: QM100.session

Results

Figure 7.3 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.

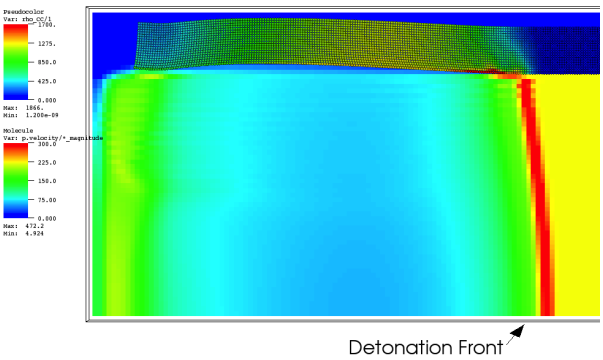


Figure 7.3: Detonation in a copper cylinder (2-D). Particles are colored by velocity magnitude, contours indicate density of unreacted explosive.

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 7.4.1.

Simulation Specifics

Component used: mpmice (MPM-ICE)
Input file name: guni2dRT.ups
Preprocessing on input file:

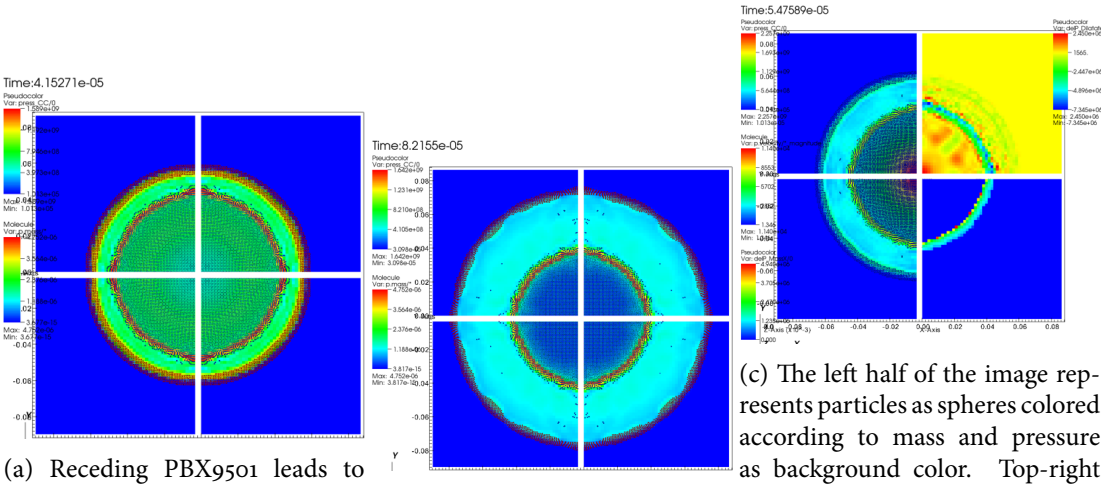
1) Comment out or remove <max_Timesteps> on line 21
2) Comment out <outputTimestepInterval> on line 96
3) add <outputInterval>5e-5<outputInterval> on line 97

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 7.4

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 7.4a. At this time pressure has increased to three-fold its initial value. A later snapshot Figure 7.4b shows the response of the steel cylinder to increased pressure. Note that mass flux will scale according to 7.4. Another interesting view of the simulation can be seen in Figure 7.4c. 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 6.1.10 for description of these variables.

Exploding Cylinder Using Steady Burn

Problem Description

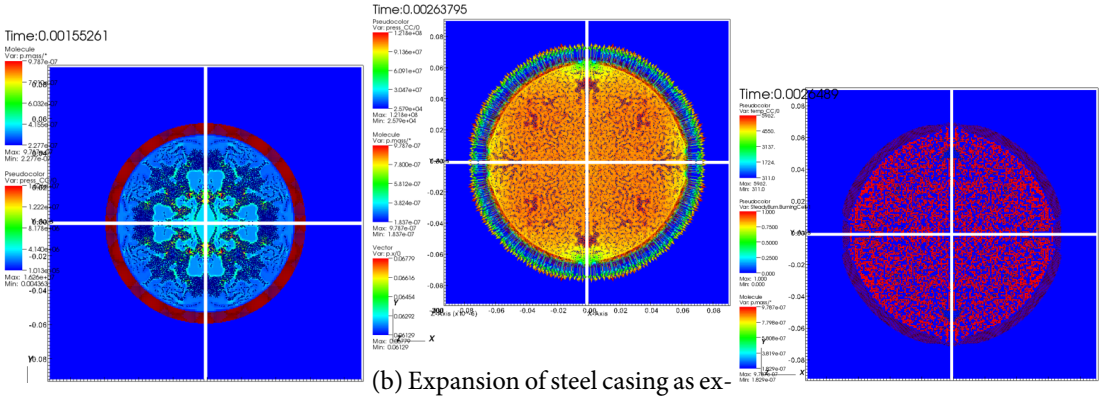
This problem consistes of a cylinder initially at 600 K causing burning. Steady Burn acts as the model for burning of HE material. More information on Steady Burn can be found in 7.4.2. The cylinder is build from an outer shell of steel covering a hollow bored cylinder of PBX9501. The simulation demonstrates 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 [73].

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 7.5

Results

Figure 7.5a 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 7.5b 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 7.5c 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 [79]. 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 [79]. This simulation demonstrates the utility of Unsteady Burn in simulations where pressure oscillations occur in small places. For more information on Unsteady Burn see 7.4.3.

Simulation Specifics

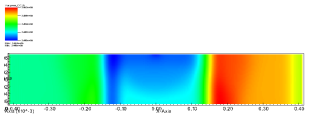
Component used: mpmice (MPM-ICE)
Input file name: `TBurner_2dRT.ups`
Command used to run input file: mpirun -np 4 sus TBurner_2dRT.ups
Simulation Domain: 0.822 x 0.138 x 0.003 m
Example Runtimes:
25 minutes (1 processor, 2.8 GHz Xeon)

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 Δt constant

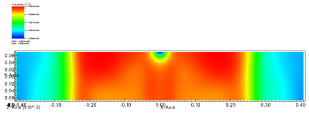
Associated visit session: TBurner.session

Results

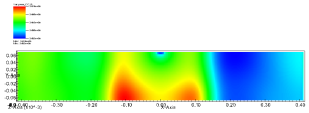
Figure 7.6a, 7.6b and 7.6c show successive snapshots of the simulation. Contour plot depicts pressure and represents the wave front as it oscillates between two sheets of burning PBX 9501. Figure 7.6d shows velocities of gas cells.



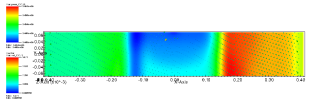
(a) Time 1: Oscillatory behavior in the form of a pressure wave in a T-Burner. Contour plot depicts pressure



(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 7.6

Figure 7.6d 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

Pinwheel

Problem Description

This simulation provides an example of fluid-solid interaction through driving a pinwheel via a supersonic jet of air. A 500 m/s jet of air is directed at the face of one of four fins attached to a hollowed cylinder surrounding a pin. This continuous jet of air spins the wheel around the stationary pin creating an easily visually verifiable fluid-solid interaction model.

Simulation Specifics

Component used: mpmice (MPM-ICE)
Input file name: `pinWheel.ups`
Command used to run input file: mpirun -np 6 sus inputs/UintahRelease/MPMICE/pinWheel.ups
Simulation Domain: 4.5 x 3.0 x 4.5 m
Example Runtimes:
10 hours (6 processors, 2.66 GHz Xeon)

Physical time simulated: 1 second

Associated visit session: pinWheel.session

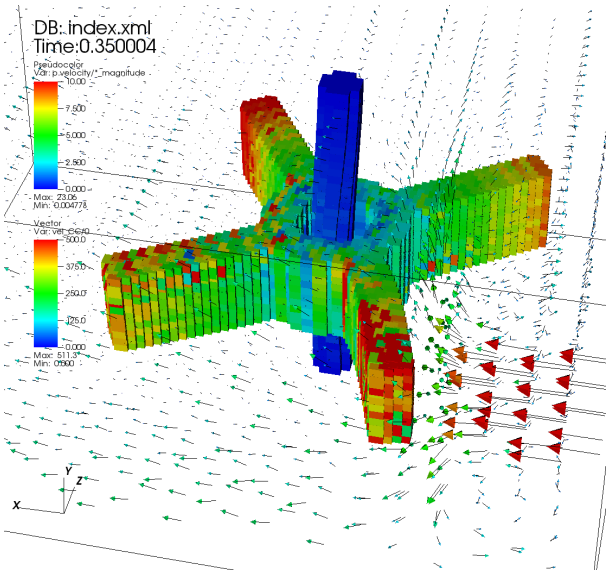
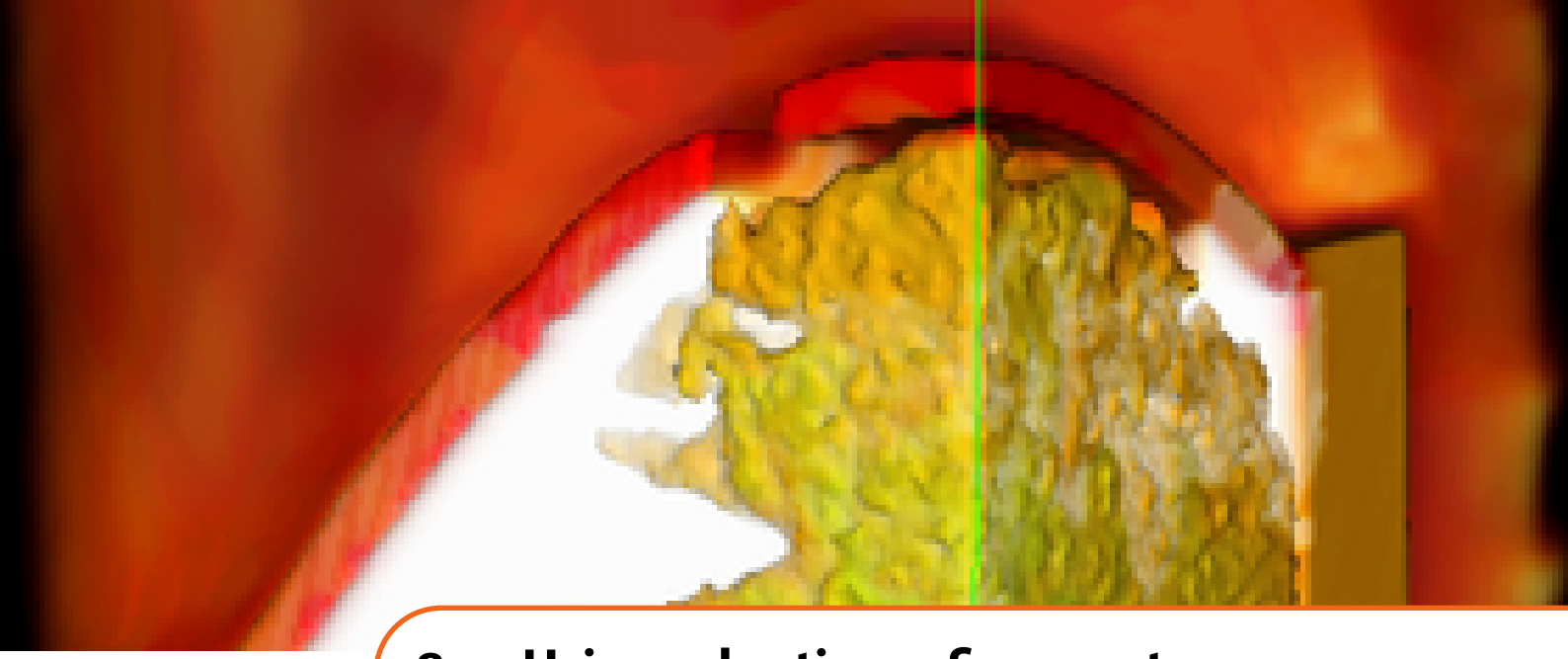


Figure 7.7: Pinwheel driven by a jet of air

Results

Figure 7.7 shows a visualization of the pinwheel alongside the air jet represented by the velocity vectors. The air jet contacts a fin on the wheel causing it to continuously rotate counterclockwise as seen by the colored velocity of the boxes.



8 — Using adaptive refinement

Whether working with an adaptive or a static grid, adaptive mesh refinement (AMR) AMR problems follow the same cycle. 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 necesary will be created on those patches by interpolation from the coarser level.

After an entire cycle, 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.

Figure 8.1 illustrates the W-cycle approach for a time refinement ratio of 2. Figure 8.2 illustrates a lockstep-cycle .

8.1 AMR inputs

In general, the adaptive mesh refinement (AMR) input for CFD problems in the .ups file looks like the following:

```
<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="BNR">
```

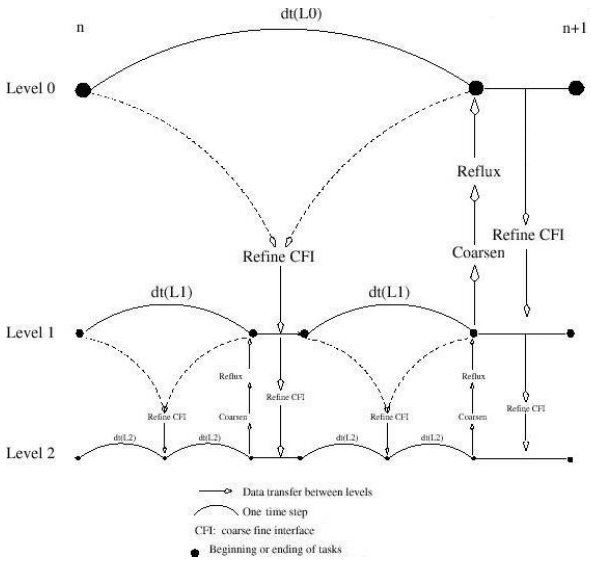


Figure 8.1: AMR Cycle W

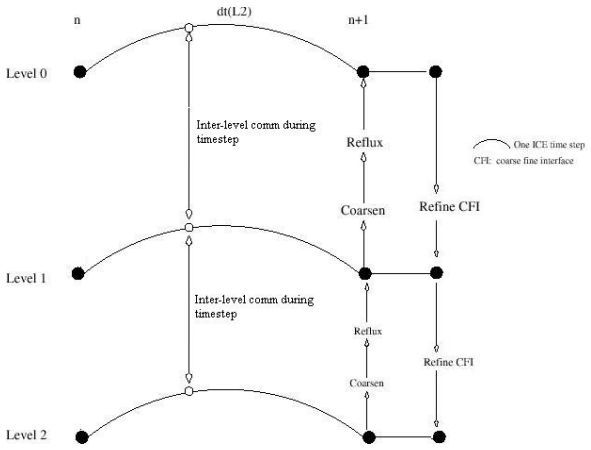


Figure 8.2: AMR Cycle Lock

```
15      <!--to use hierarchical regridder set the type to "Hierarchical",
16          to use the tiled regridder set the type to "Tiled" -->
17      <type> BNR </type>
18
19      <!--General Regridder Settings-->
20      <max_levels>              2         </max_levels>
21      <cell_refinement_ratio>   [[2,2,1]] </cell_refinement_ratio>
22      <cell_stability_dilation> [2,2,0]   </cell_stability_dilation>
23      <cell_regrid_dilation>    [1,1,0]   </cell_regrid_dilation>
24      <min_boundary_cells>      [1,1,0]   </min_boundary_cells>
25
26      <!--Hierarchical Specific Settings-->
27      <lattice_refinement_ratio> [[4,4,1],[2,2,1]] </lattice_refinement_ratio>
28
29      <!--Berger-Rigoutsos Specific Settings-->
30      <min_patch_size>        [[8,8,1]] </min_patch_size>
31      <patch_ratio_to_target> 0.2       </patch_ratio_to_target>
32
33      <!--Tiled Specific Settings-->
34      <min_patch_size>             [[8,8,1]] </min_patch_size>
35      <patches_per_level_per_proc> 8         </patches_per_level_per_proc>
36    </Regridder>
37  </AMR>
```

If you run an ICE simulation, then you must specify the ICE section.

- do_refluxing - specifies whether or not to perform refluxing, 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).
- the 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, BNR (Berger-Rigoutsos), Hierarchical (default). Only the Tiled regridder can be used with ICE and MPM problems.
- 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, as in the example for lattice_refinement_ratio, and for each level not included in the list, it will be set to the last value specified.
- cell_stability_dilation - How much to pad the refinement flags in each dimension for stability reasons.
- cell_regrid_dilation - How much to pad the refinement flags in each dimension in order to reduce regridding frequency.
- 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).

Hierarchical Specific Settings

- lattice_refinement_ratio - Specific to Hierarchical Regridder. Determines how many patches to potentially divide a coarser patch into on the finer level. See Regridding section below.

Berger-Rigoutsos 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.
- patch_ratio_to_target - sets the maximum patch size to the average work load per processor times this value. Theoretical load imbalance should be close to one half of this value. Setting this value too small will create an excess number of patches and cause excessive overhead. .2 seems to be a reasonable value.

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.

If you are using the Berger-Rigoutsos regridder you should also include a LoadBalancer (see Section 9).

8.2 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 take the flags, and, wherever there are refinement flags, patches are constructed around them on a finer level. grids are constructed from the set of levels in the input file.

8.2.1 Regridding approaches

For an adaptive problem, specify the Regridder section in the input file.

Hierarchical regridder

The Hierarchical regridder works as follows:

Divide each patch into subpatches according to the lattice_refinement_ratio . E.g., with a ratio of [2,2,2], it will create 8 subpatches. Then, if there are refinement flags within the region of that subpatch, then a patch (with resolution increased by the cell_refinement_ratio) will be added with the subpatch's range on the next finer level. This regridder is inefficient and has been superseded by the Tiled regridder.

Berger-Rigoutsos regridder

The Berger-Rigoutsos regridder works as follows:

Tiles of the size of the minimum patch size are laid across the next finer level. The refinement flags are then used to give each of these tiles a single flag indicting if a flag exists within them. The Berger-Rigoutsos algorithm is then used to create an initial patch set. Next a post processing phase splits patches in order to meet alignment constraints. Finally another post process phase splits the largest patches further depending on the patch_ratio_to_target input file specification. The Berger-Rigoutsos regridder should produce patch sets with a much smaller number of patches than the Hierarchical Regridder causing less overhead allowing decreasing AMR overhead. Unfortunately the cost of the running the BNR algorithm can be substantial at large numbers of processors and the patch sets produced by this regridder are difficult to load balance. Because of this the Tiled regridder should be used.

Tiled regridder

The Tiled regridder works as follows:

Tiles of the size of the minimum patch size are laid across the next finer level. 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 tiles 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. This regridder will eventually obsolete the Berger-Rigoutsos and Hierarchical regridders.

8.2.2 Regridding theory

The following is some general regridding information:

If there was a patch on a fine level during one timestep, and then there are no refinement flags in its region on the coarser level, then it will be removed during the regridding process.

After patches are added, data are stored on them Then data will be initialized for those new patches, and the next timestep, those patches will be included in the regridding process.

A constraint of the Regridder, is that any patch that shares a boundary with a patch on a different level must be within level. See (E) and (F) below.

At initialization time, the regridder can be executed and then the problem reinitialized so the problem can be initialized with all max_levels of refinement.

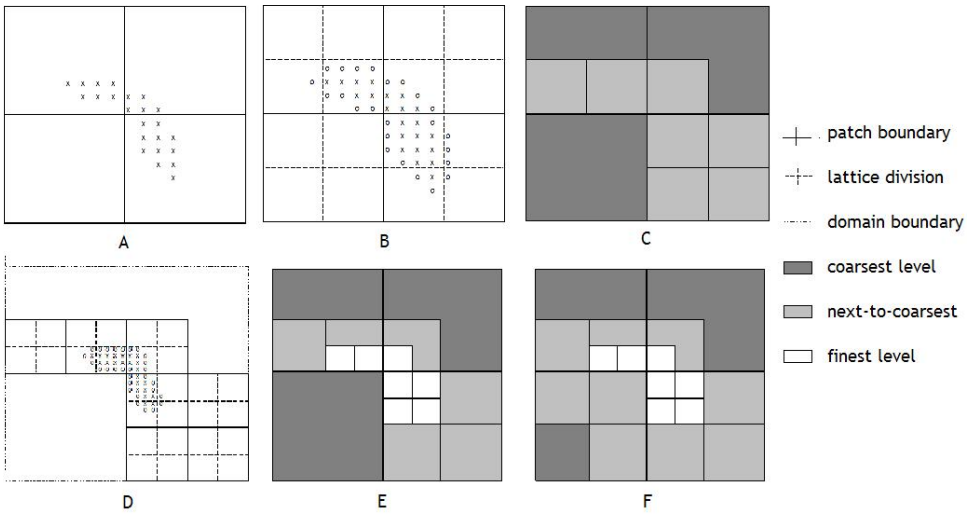


Figure 8.3: AMR Regridding

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 reulting level with the other levels. However there are some patch boundaries that span more than one level. So (F) we must expand out the middle level to compensate.

Note that if you 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 (but make sure to not include a Regridder section) in the input file. See the multiple level example in Grid 2.10.

8.2.3 Regridder inputs

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 VAANGO 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.

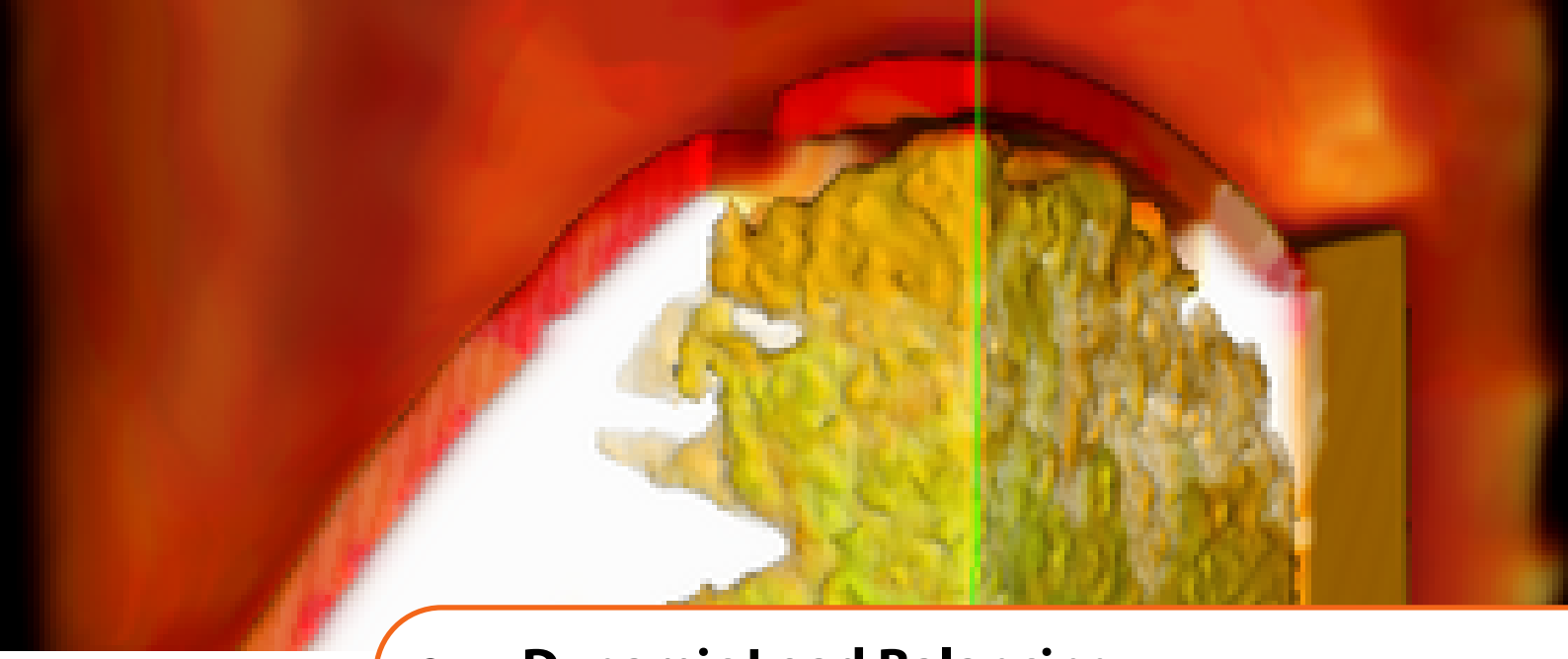
The following is an example of this regridder.

```
<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.



9 — Dynamic Load Balancing

VAANGO has a few 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.

9.1 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 VAANGO. 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.

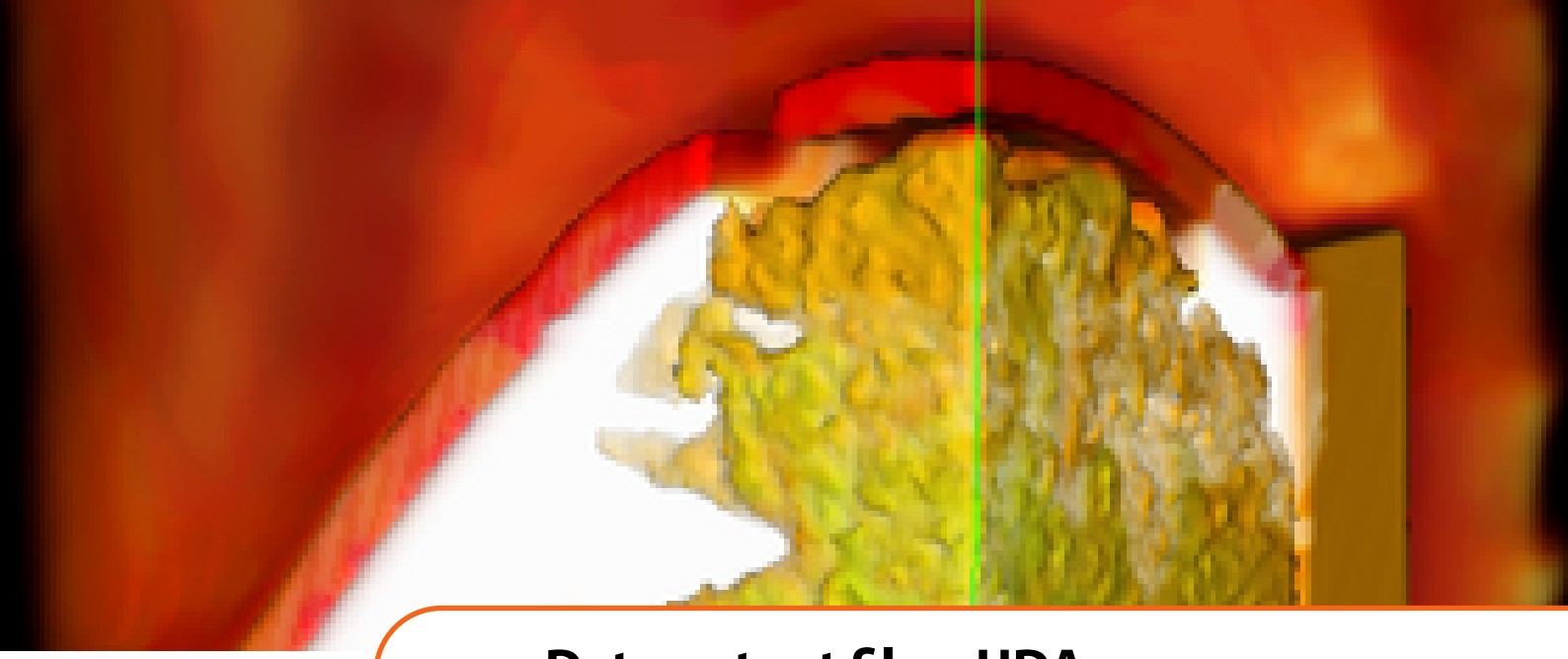
This algorithm first orders the patches linearly. If doSpaceCurve is set to true then this ordering is done according to a Hilbert Space-Filling curve, which will likely provide better clusterings. Once the patches are ordered linearly, costs are assigned to each patch and the patches are distributed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



10 — Data output files: UDA

The UDA is a file/directory structure used to save VAANGO 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 (vaango) is run, a new UDA is created. VAANGO 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 ¡filename¿.uda is updated and will point to the newest version of this simulations UDA. For example,

```
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:

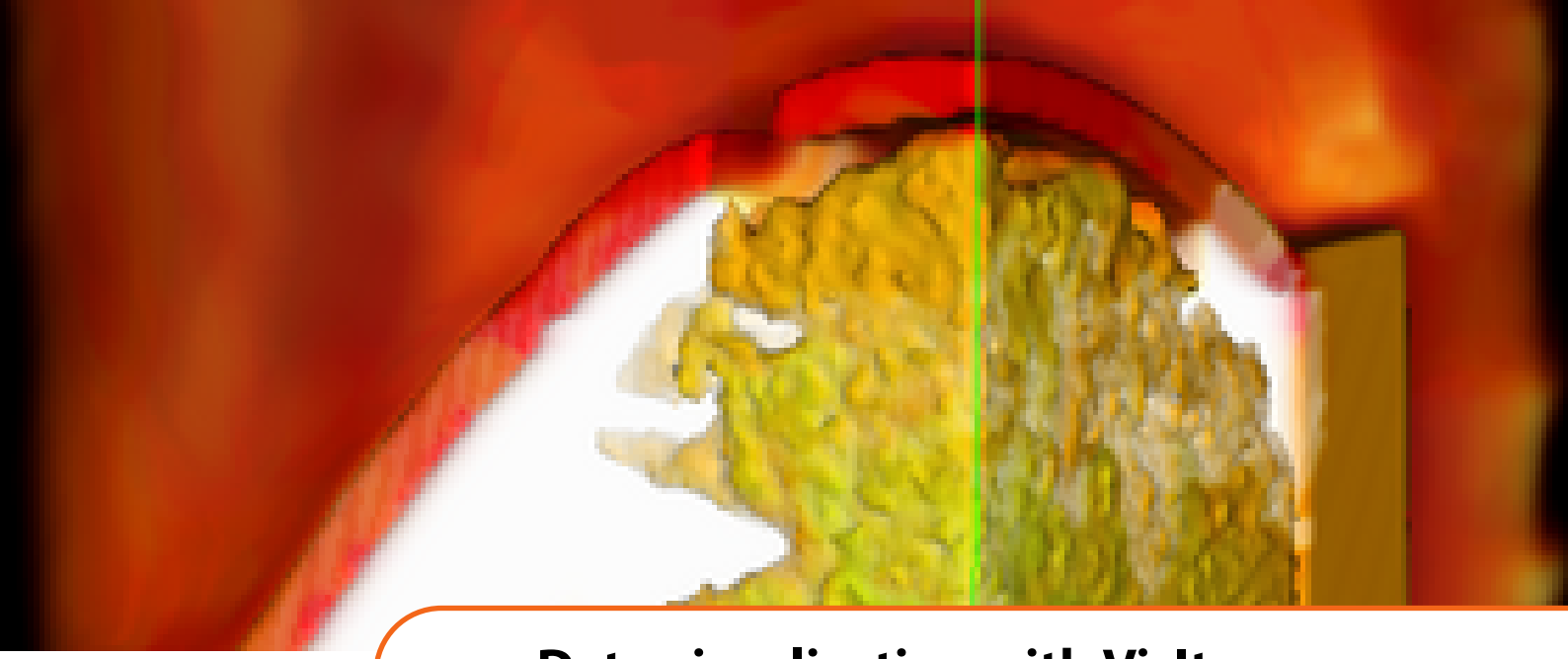
- .dat files contain global information about the simulation (each line in the .dat files contains: simulation_time value).
- checkpoints directory contains a limited number of time step data subdirectories that contain a complete snapshot of the simulation (allowing for the simulation to be restarted from that time).
- input.xml contains the original problem specification (the .ups file).
- index.xml contains information on the actual simulation run.
- toooo# 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 in src/Packages/Uintah/scripts/ can be used to test the integrity of a UDA directory. It does not interrogate the data for correctness, but 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.



11 — Data visualization with VisIt

11.1 Introduction

Visualization of Vaango data is currently performed using VisIt . The VisIt package from LLNL is general purpose visualization software that offers all of the usual capabilities for rendering scientific data. It is developed and maintained by LLNL staff, and its interface to Vaango data is supported by the Uintah team. The latest VisIt version that Vaango has been tested on is VisIt 3.1.1. The UDA reader plugin is currently incorporated directly into VisIt and does not have to be installed separately.

You can install VisIt by downloading one of the pre-built executables from `https://wci.llnl.gov/simulation/computer-codes/visit/executables`. Alternatively, you can build your own version from source using instructions given in the VisIt web page.

11.2 Particle data visualization

To visualize particle information, run VisIt . To open a UDA, select Open File from the File menu (Figure 11.1). Browse into the UDA you want to load (Figure 11.2) and select the index.xml file (Figure 11.3). Then hit on OK and a list of timesteps should now appear on the gui (Figure 11.4).

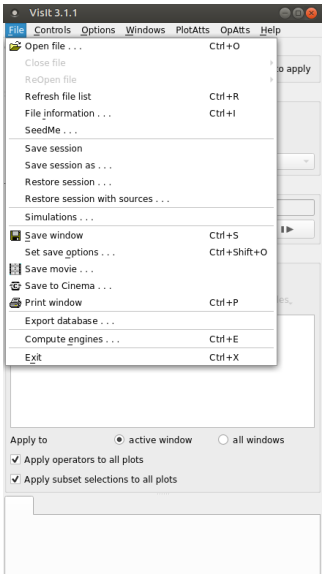


Figure 11.1: VisIt control window.

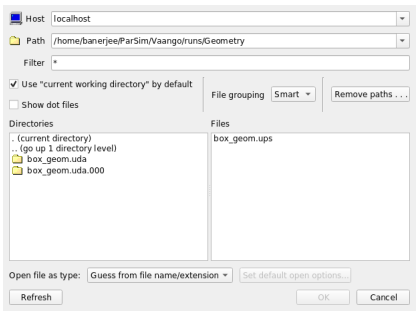


Figure 11.2: VisIt file open dialog.

Now a window indicating an active source is created with an Add button. Click on this button and select Pseudocolor and the particle variable p.volume (or any other quantity that is available in the UDA) as

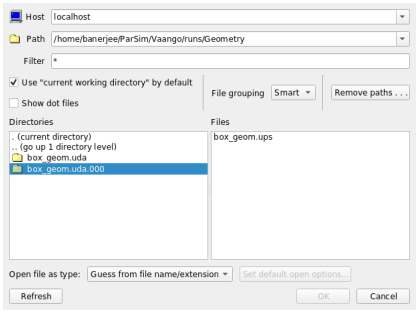


Figure 11.3: Selecting a UDA to load.

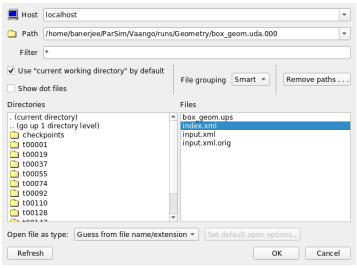


Figure 11.4: Selecting the index.xml file.

shown in Figure 11.5. This allows you to plot scalar quantities that are associated with the MPM particles. The variable to be plotted is highlighted in green. Click on the Draw button (Figure 11.6) to visualize the data.

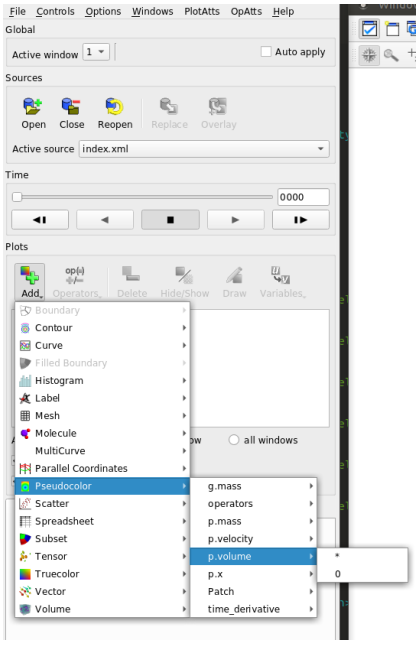


Figure 11.5: Selecting the scalar particle variable to plot.

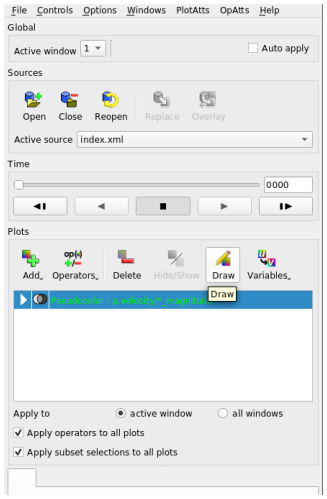


Figure 11.6: Clicking on the Draw button.

A plot of the data shows up on the display window as seen in Figure 11.7.

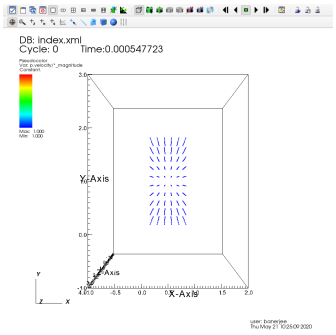


Figure 11.7: Plot of the particle data in the UDA file.

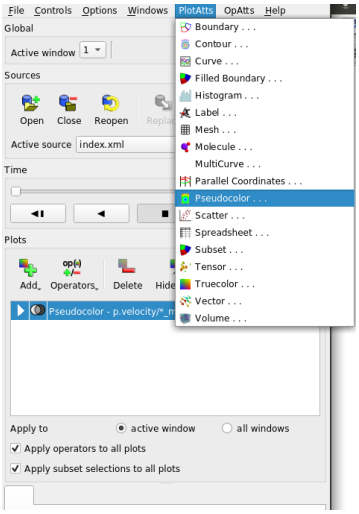


Figure 11.8: Selecting pseudocolor plot attributes.

The particles are too small in this plot because they are represented as points. We can increase the size by changing the particle type into spheres. To do that select PlotAtts as shown in Figure 11.8. If you then choose the Pseudocolor item from the list, a window pops up that allows you to modify the appearance of the particles (Figure 11.9). Change the button for Point Type from Point tp Sphere (Figure 11.10).

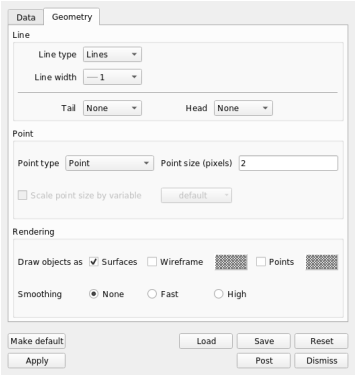


Figure 11.9: Changing the appearance of particles.

Figure 11.10: Switching from point to sphere respresentation.

Figure 11.11 shows the updated view of the MPM particles. However, the color is still too dark. If you want a lighter color, you will have to change the color map (Figure 11.12. Select any one of the options that pops up when you select Color table and you will get a set of particles colored differently from the default as seen in Figure 11.13.

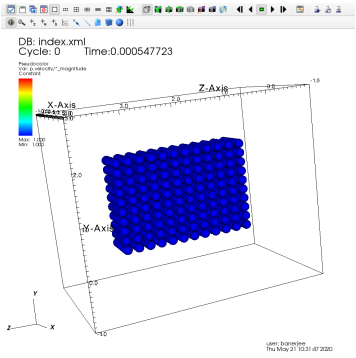


Figure 11.11: Particles drawn as spheres.

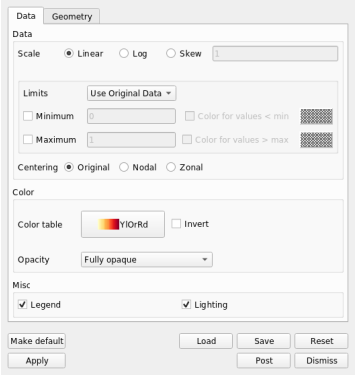


Figure 11.12: Changing the color map.

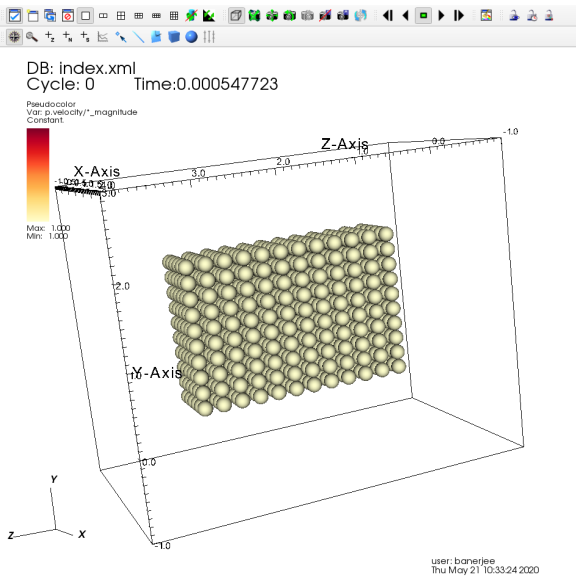


Figure 11.13: Changing the appearance of particles.

11.3 Volumetric data plots

VisIt displays data as plots. A plot might render a specific variable or it might render the structure of the mesh. Figure 11.14 illustrates this. Note that VisIt attempts to analyze the variables and associate them with the appropriate plots. As shown in Figure 11.14, 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.

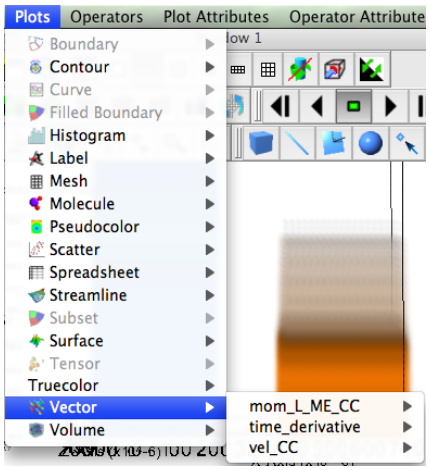


Figure 11.14: Various plots in VisIt

Once you have a plot, you change plot attributes by clicking on the PlotAtts menu and selecting the plot of you choice. Alternatively, 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 11.15 pops up.
As seen in Figure 11.15, you can change the color map, opacity curve, rendering method, no. of samples, lighting options, etc. in this window.

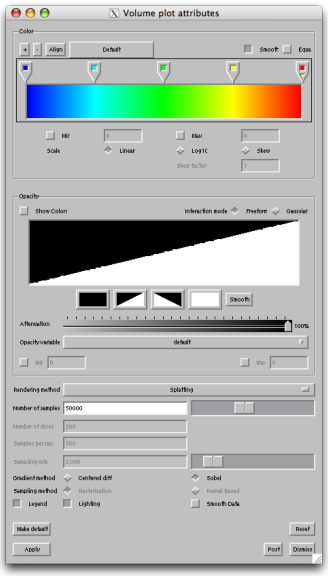


Figure 11.15: Volume plot attributes in VisIt

11.4 Operators

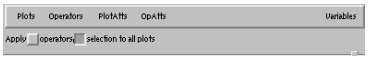
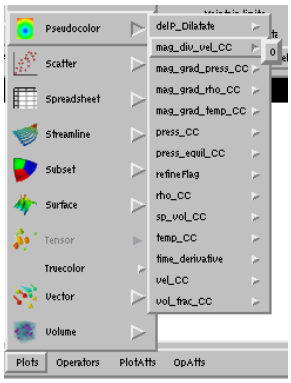


Figure 11.16: Unchecking "selection to all plots"

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 11.16.

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 11.17a and 11.17b 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 11.18a. Once you have this

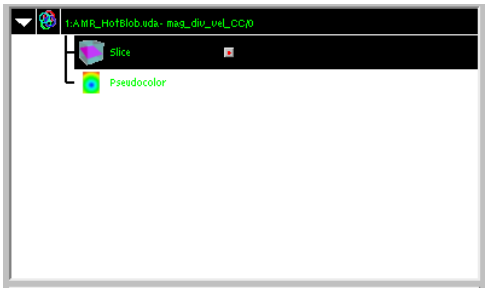


(a) Applying the Pseudocolor plot to a variable

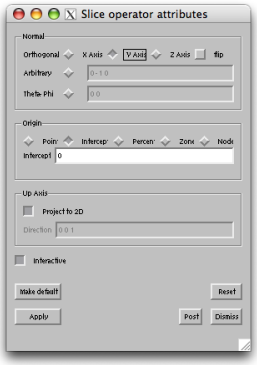
(b) Applying an operator to a plot

Figure 11.17: Pseudocolors and applying operators.

order, select Slice from the OpAtts menu. This will pop up the Slice operator attributes window, as shown in Figure 11.18b.



(a) Ordering of an operator and a plot



(b) Slice plot attributes in VisIt

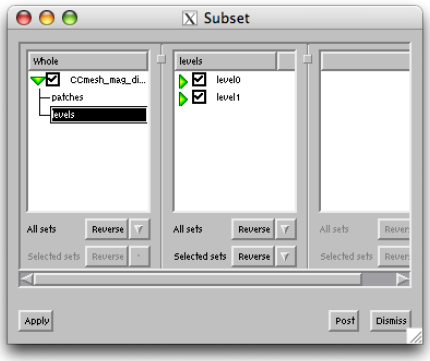
Figure 11.18: Ordering and slicing.

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.

11.5 Vectors

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 11.19, the number of vectors has been increased to 2000. Also if you would like all the vectors to 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 11.20 describes this.

(a) Clicking on this icon opens the Subset window



(b) The Subset window in VisIt

Figure 11.21: Visualizing AMR data

Figure 11.19: Increasing the number of Vectors

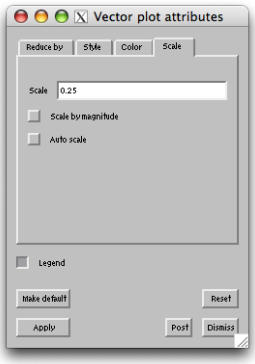


Figure 11.20: Increasing the scale of Vectors

11.6 AMR datasets

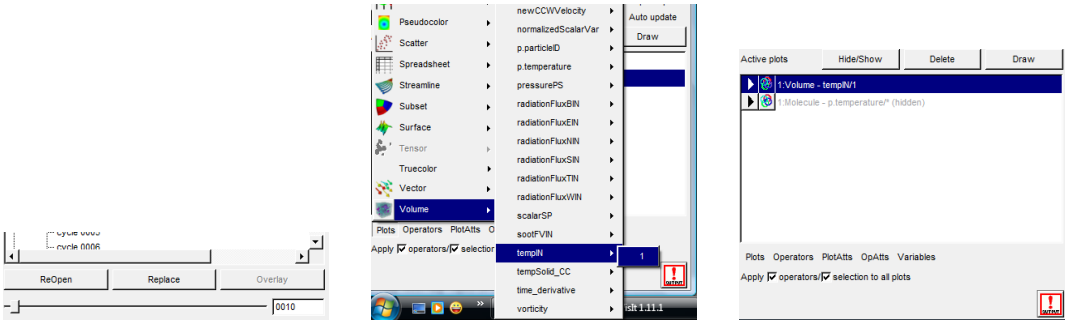
AMR datasets are read the same way as single level datasets. Once you have it read, you can apply an plot/ operator on it. Since the dataset is organized as levels and patches, you now have the flexibility of visualizing each of them independently or as in a group. To achieve this (assuming that you 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 11.21a and 11.21b.

11.7 Examples

11.7.1 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 11.22a), 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 11.22b. The number '1' refers to the material associated with the variable.
4. The variable tempIN/1 now appears on the Active plots window (Figure 11.22c). Select the variable

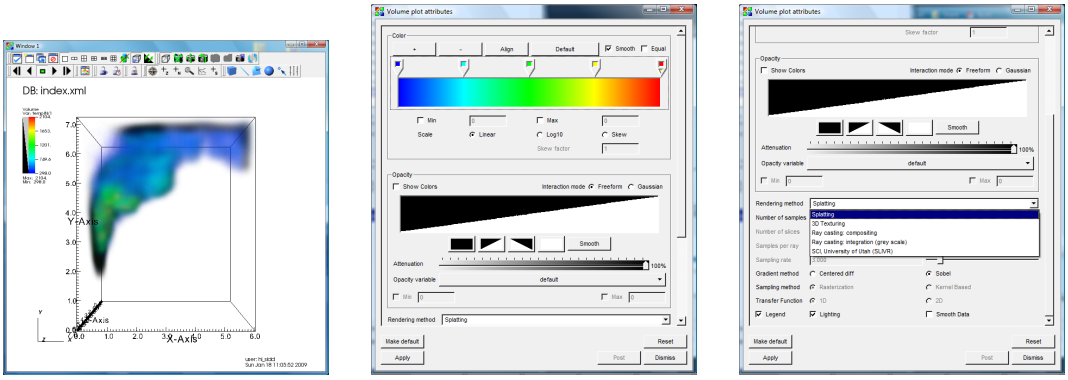
and click Draw.



(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 11.22

5. A visualization now appears on the Viewer window, as shown in Figure 11.23a. You can interact with the visualization in terms of rotating it (holding the left mouse button and dragging it), 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 11.23b and figure 11.23c).



(a) Visualization of a volume on the viewer window (b) Volume visualization attributes window (c) Volume visualization attributes window

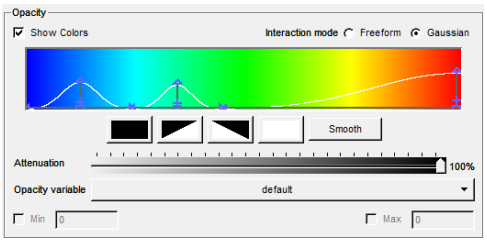
Figure 11.23

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 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 Interac-



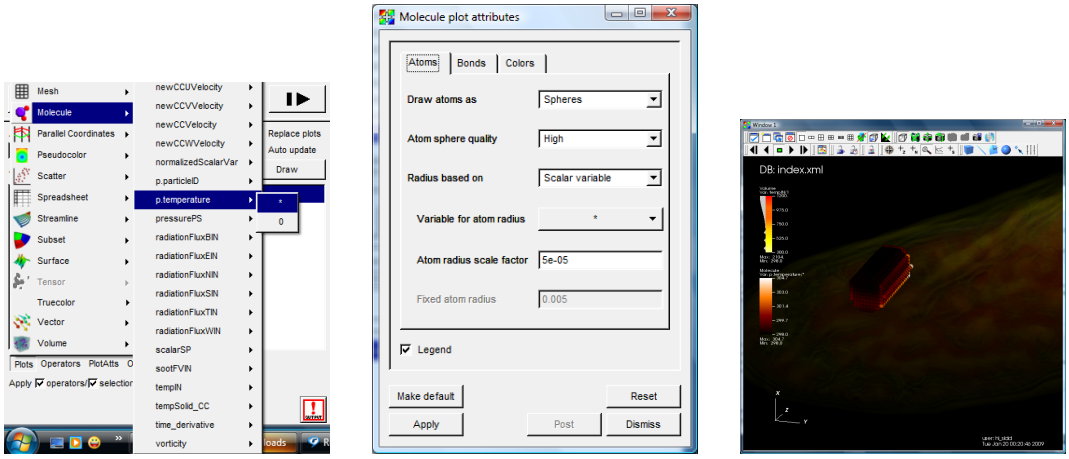
tion Mode as Gaussian lets you draw curves and specify a more accurate color table (Figure 11.24).

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.

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.

11.7.2 Particle visualization

1. To add particles, we select the Molecule plot and then click on the variable p.temperature as shown in the Figure 11.25a. 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 11.25b.



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

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

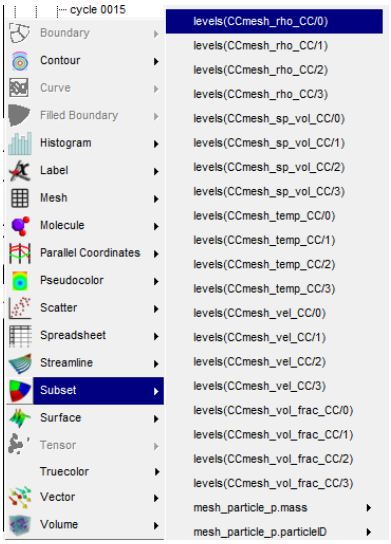
(c) Selecting a molecule plot and an associated variable/ material

Figure 11.25

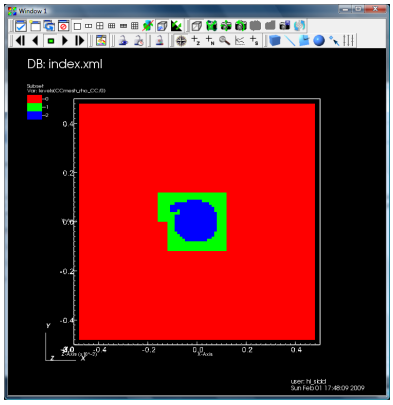
We choose to visualize the particles as Sphere Impostors (doesn't runs the GPU out of memory, drawing as Spheres does). We also choose to scale the sphere radius by a Scalar Variable and specify that variable to be p.temperature/* itself (therefore the * appears). Since the temperature values are too 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 11.25c.

11.7.3 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 11.26a, we select one such variable. Next, we hit Draw. This produces a visualization as shown in Figure 11.26b.



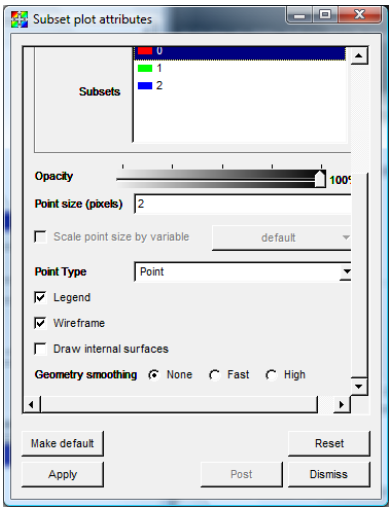
(a) A patch/ level variable, associated with every kind of variable



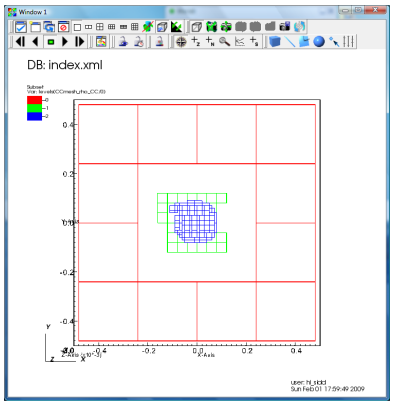
(b) The default visualization of patches

Figure 11.26

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‘11.27a. This would produce a visualization, similar to one shown in Figure 11.27b.



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



(b) The patch boundaries after enabling the wireframe mode

Figure 11.27

11.7.4 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 11.28 illustrates this.

Once the plot is selected, we hit 'Draw'. This would produce a visualization, similar to one shown in

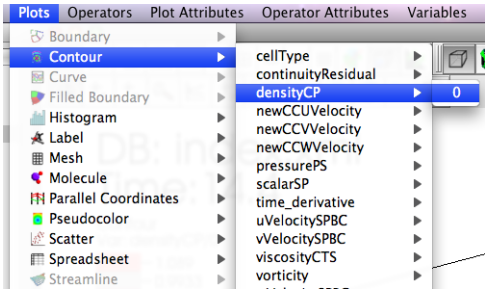
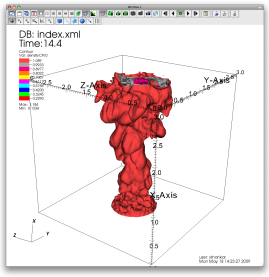


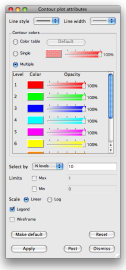
Figure 11.28: Selecting the 'Contour' plot on a regular 3D scalar variable

Figure 11.29a. 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 11.29b.

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



(a) Iso-surface visualization



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

Figure 11.29

11.7.5 Streamlines

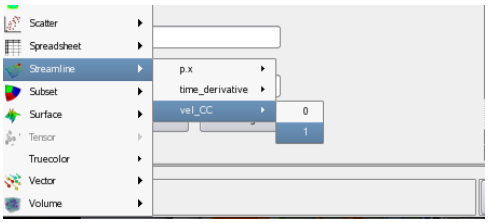


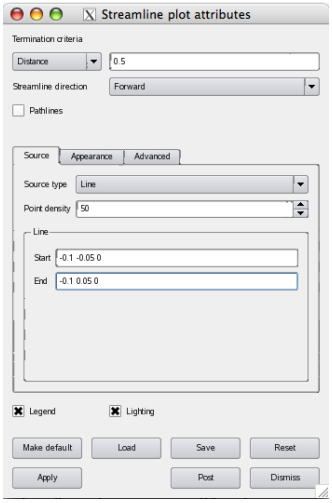
Figure 11.30: Selecting the 'Streamlines' plot on a vector variable

As shown in Figure 11.30 we select the 'Streamlines' plot on a vector variable. We then double click on the plot itself, which pops up the 'Streamlines attributes window'.

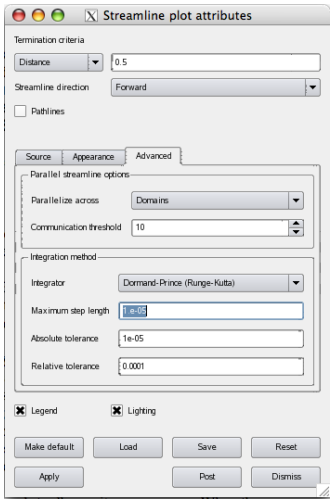
We set the 'Distance' parameter such that it covers the entire computational domain. We set the 'Streamline direction' as forward. In the 'Source' tab, Figure 11.31a, we define the 'Source type' as 'Line'. We can select other options too, notably 'Single Point', 'Sphere' etc. We now define the line 'Start' 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.

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 should be order of magnitude smaller than the length of the computational domain. This is shown in Figure 11.31b.

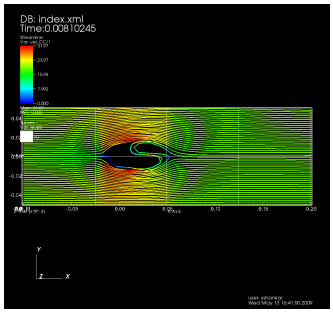
Once these parameters are set, we hit 'Apply' and then click on the 'Draw' button on the gui. This produces a visualization similar to one shown in Figure 11.31c.



(a) Setting the 'Source' tab parameters



(b) Setting the 'Advanced' tab parameters



(c) Streamlines visualization

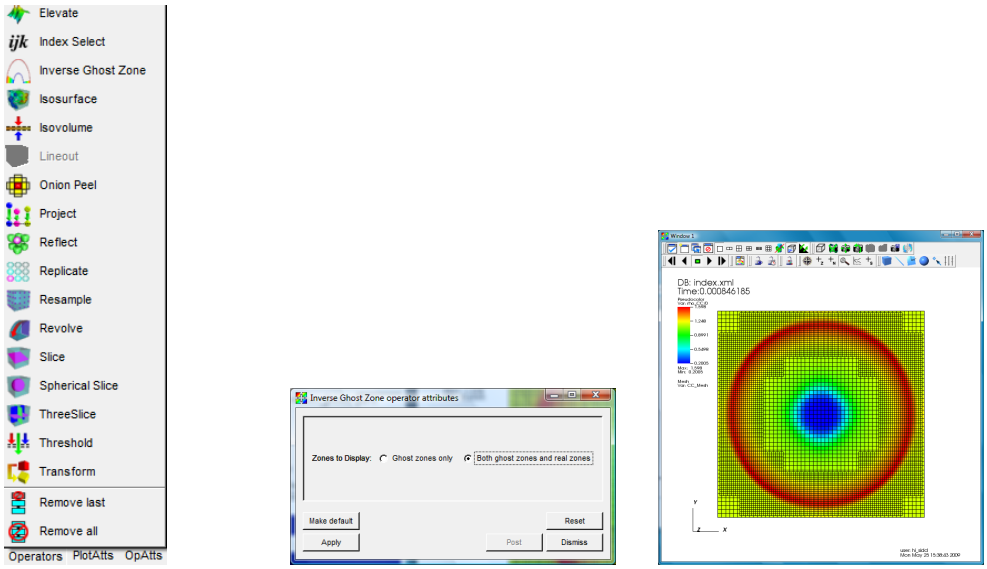
Figure 11.31

11.7.6 Visualizing extra cells

For visualizing extra cells we use the 'Inverse Ghost Zone' operator 11.32a 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 11.32b and hit 'Apply'.

We then hit 'Draw'. When combined with the 'Mesh' plot we get a visualization similar to the one shown in Figure 11.32c. The pick operations on the viewer can then be used to investigate the value(s) in these extra cells.



(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 11.32

11.7.7 Picking on particles

Figure 11.33: The 'Node pick mode' on the visualization window

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' 11.33 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 can be added by selecting additional variables from the 'Variables' menu and as shown in the Figure 11.34.

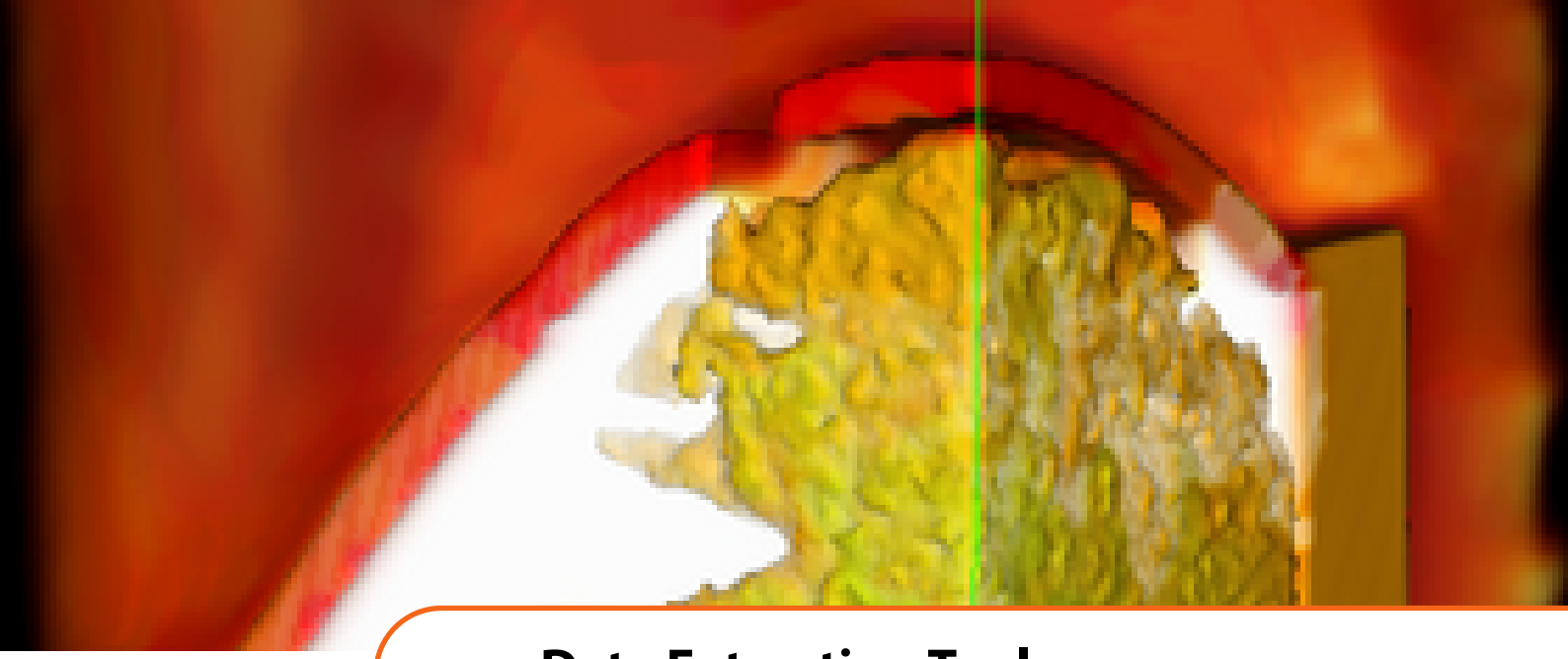
Figure 11.34: The 'Pick' window

11.7.8 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,

```
1 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'.



12 — Data Extraction Tools

Vaango offers a number of tools for accessing simulation output data stored in UDA archives. The user can use these tools in combination with Python, R, or Octave scripts to examine and plot the simulation data.

12.1 puda

The command line extraction utility puda (short for "parse UDA") 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 <build_dir>/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 -2.72031498e-10 -1.23584688e-09 -2.53781271e-08 -1.23584688e-09 1.63840079e-07
1.00016560e-01 1 1 0 1.93256890e-13 6.56787331e-18 1.85514400e-14 6.56787331e-18 2.24310469e-13 1.85519650e-14 1.85514400e-14 1.85519650e-14 -3.20052991e+06
```

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 (σ_{11},σ_{12},σ_{13}, ..., σ_{32},σ_{33}). If desired, the terminal output can be redirected to a text file for further use.

12.2 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. 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 -1.05
    064208e-26 -2.72031498e-10 -1.23584688e-09 -2.53781271e-08 -1.23584688e-09 1.6384007
    9e-07
```

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 σ_{11},σ_{22}, σ_{33}, σ_{23}, σ_{13} , σ_{12}.

12.3 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
```

12.4 compute_Lnorm_udas

Compute_Lnorm_udas computes the L_1, L_2 and L_∞ 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:

$$d[i] = \|\mathrm{UDA}_1[i] - \mathrm{UDA}_2[i]\| \tag{12.1}$$

$$L_1 = \frac{\sum_i^{\text{All Cells}} d[i]}{\text{number of cells}}, \quad L_2 = \sqrt{\frac{\sum_i^{\text{All Cells}} d[i]^2}{\text{number of cells}}}, \quad L_\infty = \max(d[i]) \tag{12.2}$$

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_∞. 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

$$|\mathrm{physicalTime}_{\mathrm{uda}_1} - \mathrm{physicalTime}_{\mathrm{uda}_2}| < 1e^{-5}.$$

12.5 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
```

12.6 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, & libtiffxxoc2 , 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:

$$CC_{ave} = \frac{\sum_{p=1}^{\text{nParticles}} \text{Double[p]}}{\text{nParticles}}$$

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

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

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
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. 12.1. In this simulation two similar disks collided at the center of the domain.

12.7 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

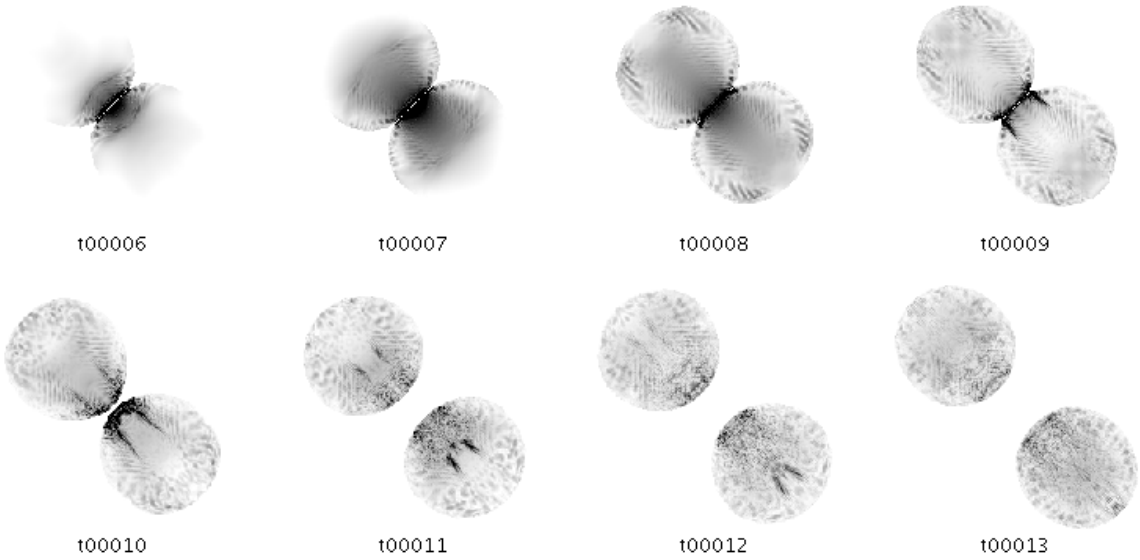
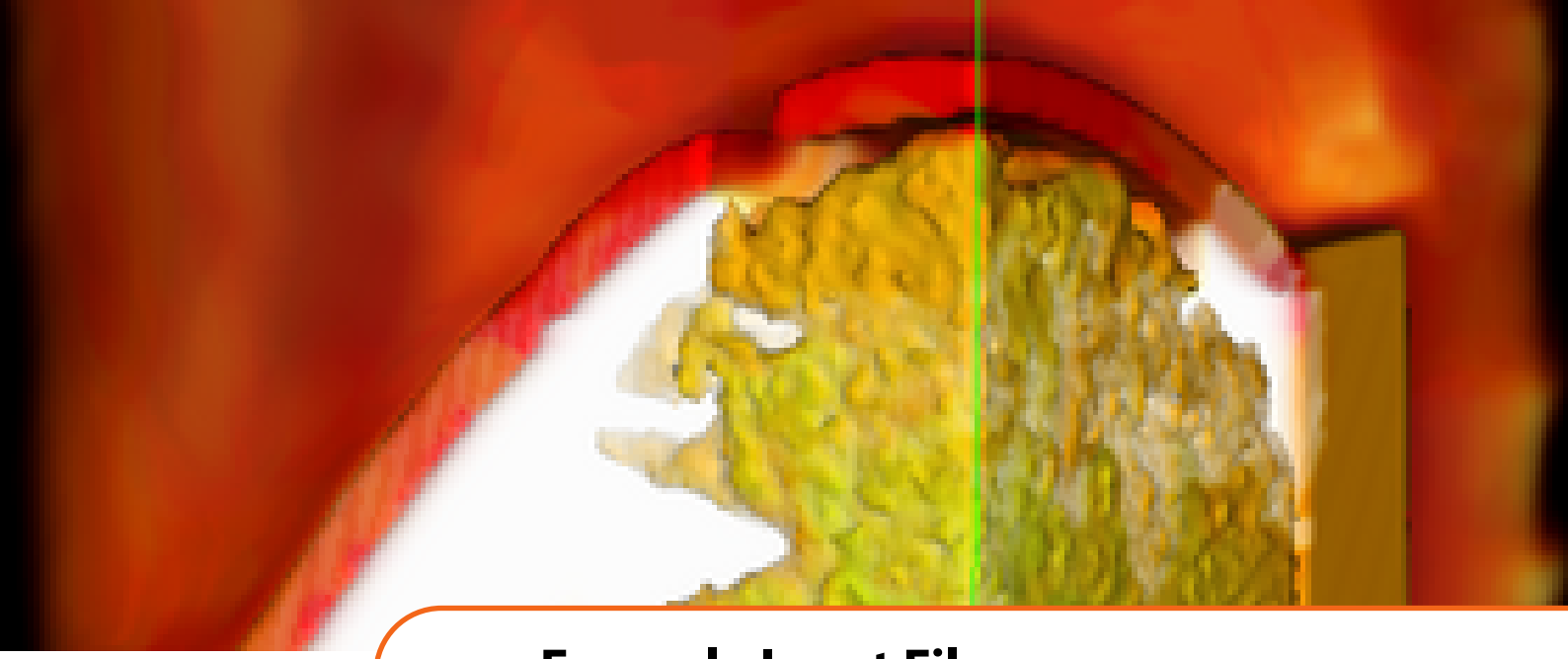


Figure 12.1: Montage showing the averaged particle stress for two cylindrical disks colliding.

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>
```



13 — Example Input Files

13.1 Hypoelastic-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="hypoelastic_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>

  <burn type = "null" />
  <velocity_field>1</velocity_field>

  <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>
  <plasticity_model type="johnson_cook">
    <A>89.6e6</A>
    <B>292.0e6</B>
    <C>0.025</C>
    <n>0.31</n>
    <m>1.09</m>
  </plasticity_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-Vaango 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 `.../Vaango/StandAlone/inputs/MPM/MaterialData`.

13.2 Elastic-plastic model

The `<constitutive_model type="elastic_plastic">` model is more stable (and also more general) than the `<constitutive_model type="hypoelastic_plastic">` model. A sample input file for this model is shown below.

```
<MPM>
```

```
    <do_grid_reset> false </do_grid_reset>
    <time_integrator>explicit</time_integrator>
    <boundary_traction_faces>[zminus,zplus]</boundary_traction_faces>
    <dynamic>true</dynamic>
    <solver>simple</solver>
    <convergence_criteria_disp>1.e-10</convergence_criteria_disp>
    <convergence_criteria_energy>4.e-10</convergence_criteria_energy>
    <DoImplicitHeatConduction>true</DoImplicitHeatConduction>
    <interpolator>linear</interpolator>
    <minimum_particle_mass> 1.0e-8</minimum_particle_mass>
    <maximum_particle_velocity> 1.0e8</maximum_particle_velocity>
    <artificial_damping_coeff> 0.0 </artificial_damping_coeff>
    <artificial_viscosity> true </artificial_viscosity>
    <accumulate_strain_energy> true </accumulate_strain_energy>
    <use_load_curves> false </use_load_curves>
    <turn_on_adiabatic_heating>   false    </turn_on_adiabatic_heating>
    <do_contact_friction_heating> false    </do_contact_friction_heating>
    <create_new_particles>        false    </create_new_particles>
    <erosion algorithm = "none"/>
  </MPM>

  <MaterialProperties>
    <MPM>
      <material name = "OFHCCu">
        <density>              8930.0 </density>
        <thermal_conductivity> 386.0  </thermal_conductivity>
        <specific_heat>        414.0  </specific_heat>
        <room_temp>            294.0  </room_temp>
        <melt_temp>            1356.0 </melt_temp>
        <constitutive_model type="elastic_plastic">
          <isothermal>                    false   </isothermal>
          <tolerance>                     1.0e-12 </tolerance>
          <do_melting>                    false   </do_melting>
          <evolve_porosity>               false   </evolve_porosity>
          <evolve_damage>                 false   </evolve_damage>
          <check_TEPLA_failure_criterion> false   </check_TEPLA_failure_criterion>
          <check_max_stress_failure>      false   </check_max_stress_failure>
          <initial_material_temperature>  696.0   </initial_material_temperature>

          <shear_modulus>           46.0e9  </shear_modulus>
          <bulk_modulus>            129.0e9 </bulk_modulus>
          <coeff_thermal_expansion> 1.76e-5 </coeff_thermal_expansion>
          <taylor_quinney_coeff>    0.9     </taylor_quinney_coeff>
          <critical_stress>         129.0e9 </critical_stress>

          <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>

          <plasticity_model type="mts_model">
            <sigma_a>40.0e6</sigma_a>
            <mu_0>47.7e9</mu_0>
            <D>3.0e9</D>
            <T_0>180</T_0>
            <koverbcubed>0.823e6</koverbcubed>
            <g_0i>0.0</g_0i>
            <g_0e>1.6</g_0e>
            <edot_0i>0.0</edot_0i>
            <edot_0e>1.0e7</edot_0e>
            <p_i>0.0</p_i>
            <q_i>0.0</q_i>
            <p_e>0.666667</p_e>
            <q_e>1.0</q_e>
            <sigma_i>0.0</sigma_i>
            <a_0>2390.0e6</a_0>
            <a_1>12.0e6</a_1>
            <a_2>1.696e6</a_2>
            <a_3>0.0</a_3>
            <theta_IV>0.0</theta_IV>
            <alpha>2</alpha>
            <edot_es0>1.0e7</edot_es0>
```

```
            <g_0es>0.2625</g_0es>
            <sigma_es0>770.0e6</sigma_es0>
          </plasticity_model>

          <shear_modulus_model type="mts_shear">
            <mu_0>47.7e9</mu_0>
            <D>3.0e9</D>
            <T_0>180</T_0>
          </shear_modulus_model>

          <melting_temp_model type = "constant_Tm">
          </melting_temp_model>

          <yield_condition type = "vonMises">
          </yield_condition>

          <stability_check type = "none">
          </stability_check>

          <damage_model type = "hancock_mackenzie">
            <D0> 0.0001 </D0>
            <Dc> 0.7    </Dc>
          </damage_model>

          <compute_specfic_heat> false </compute_specfic_heat>
          <specific_heat_model type="constant_Cp">
          </specific_heat_model>

        </constitutive_model>
        <geom_object>
          <box label = "box">
            <min>[0.0,    0.0,    0.0]</min>
            <max>[1.0e-2, 1.0e-2, 1.0e-2]</max>
          </box>
          <res>[1,1,1]</res>
          <velocity>[0.0, 0.0, 0.0]</velocity>
          <temperature>696</temperature>
        </geom_object>
      </material>

    </MPM>
  </MaterialProperties>
```

13.2.1 An exploding ring experiment

The follwing shows the complete input file for an expanding ring test.

```

<?xml version='1.0' encoding='ISO-8859-1' ?>
<Uintah_specification>
<!--Please use a consistent set of units, (mks, cgs,...)-->
    <!-- First crack at the tuna can problem -->

   <Meta>
       <title>Pressurization of a container via burning w/o fracture</title>
   </Meta>&gt;
   <SimulationComponent>
     <type> mpmice </type>
   </SimulationComponent>
    <!--____________________________________________________________________--->
    <!--   T  I  M  E     V  A  R  I  A  B  L  E  S                          --->
    <!--____________________________________________________________________--->
   <Time>
       <max_Timesteps>   99999          </max_Timesteps>>
       <maxTime>             2.00e-2    </maxTime>
       <initTime>            0.0        </initTime>
       <delt_min>            1.0e-12    </delt_min>
       <delt_max>            1.0        </delt_max>
       <delt_init>           2.1e-8     </delt_init>
       <timestep_multiplier> 0.5        </timestep_multiplier>
   </Time>
```

```
    <!--__________________________________________________________________-->
    <!--   G  R  I  D     V  A  R  I  A  B  L  E  S                        -->
    <!--__________________________________________________________________-->
    <Grid>
    <BoundaryConditions>
      <Face side = "x-">
        <BCType id = "all" label = "Symmetric" var = "symmetry">
        </BCType>
      </Face>
      <Face side = "x+">
        <BCType id = "0" label = "Pressure" var = "Neumann">
                              <value>  0.0 </value>
        </BCType>
        <BCType id = "all" label = "Velocity" var = "Dirichlet">
                              <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>
      </Face>
      <Face side = "y-">
        <BCType id = "all" label = "Symmetric" var = "symmetry">
        </BCType>
      </Face>
      <Face side = "y+">
        <BCType id = "0" label = "Pressure" var = "Neumann">
                              <value> 0.0   </value>
        </BCType>
        <BCType id = "all" label = "Velocity" var = "Dirichlet">
                              <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>
      </Face>
      <Face side = "z-">
        <BCType id = "all" label = "Symmetric" var = "symmetry">
        </BCType>
      </Face>
      <Face side = "z+">
        <BCType id = "all" label = "Symmetric" var = "symmetry">
        </BCType>
      </Face>
    </BoundaryConditions>
       <Level>
           <Box label = "1">
              <lower>        [ -0.08636, -0.08636, -0.0016933] </lower>
              <upper>        [ 0.08636, 0.08636, 0.0016933]    </upper>
              <extraCells>   [1,1,1]    </extraCells>
              <patches>      [2,2,1]    </patches>
              <resolution>   [102, 102, 1]                 </resolution>
           </Box>
       </Level>
    </Grid>

    <!--__________________________________________________________________-->
    <!--   O  U  P  U  T     V  A  R  I  A  B  L  E  S                     -->
    <!--__________________________________________________________________-->
   <DataArchiver>
      <filebase>explodeRFull.uda</filebase>
      <outputTimestepInterval> 20 </outputTimestepInterval>
      <save label = "rho_CC"/>
      <save label = "press_CC"/>
      <save label = "temp_CC"/>
      <save label = "vol_frac_CC"/>
      <save label = "vel_CC"/>
      <save label = "g.mass"/>
```

```
      <save label = "p.x"/>
      <save label = "p.mass"/>
      <save label = "p.temperature"/>
      <save label = "p.porosity"/>
      <save label = "p.particleID"/>
      <save label = "p.velocity"/>
      <save label = "p.stress"/>
      <save label = "p.damage" material = "0"/>
      <save label = "p.plasticStrain" material = "0"/>
      <save label = "p.strainRate" material = "0"/>
      <save label = "g.stressFS"/>
      <save label = "delP_Dilatate"/>
      <save label = "delP_MassX"/>
      <save label = "p.localized"/>
      <checkpoint cycle = "2" timestepInterval = "20"/>
   </DataArchiver>

   <Debug>
   </Debug>
    <!--__________________________________________________________________-->
    <!--    I  C  E     P  A  R  A  M  E  T  E  R  S                       -->
    <!--__________________________________________________________________-->
    <CFD>
       <cfl>0.5</cfl>
       <CanAddICEMaterial>true</CanAddICEMaterial>
       <ICE>
        <advection type = "SecondOrder"/>
        <ClampSpecificVolume>true</ClampSpecificVolume>
      </ICE>
    </CFD>

    <!--__________________________________________________________________-->
    <!--     P  H  Y  S  I  C  A  L     C  O  N  S  T  A  N  T  S         -->
    <!--__________________________________________________________________-->
    <PhysicalConstants>
       <gravity>            [0,0,0]   </gravity>
       <reference_pressure> 101325.0  </reference_pressure>
    </PhysicalConstants>

    <MPM>
      <time_integrator>               explicit   </time_integrator>
      <nodes8or27>                    27         </nodes8or27>
      <minimum_particle_mass>         3.e-12     </minimum_particle_mass>
      <maximum_particle_velocity>     1.e3       </maximum_particle_velocity>
      <artificial_damping_coeff>      0.0        </artificial_damping_coeff>
      <artificial_viscosity>          true       </artificial_viscosity>
      <artificial_viscosity_coeff1>   0.07       </artificial_viscosity_coeff1>
      <artificial_viscosity_coeff2>   1.6        </artificial_viscosity_coeff2>
      <turn_on_adiabatic_heating>     false      </turn_on_adiabatic_heating>
      <accumulate_strain_energy>      false      </accumulate_strain_energy>
      <use_load_curves>               false      </use_load_curves>
      <create_new_particles>          false      </create_new_particles>
      <manual_new_material>           false      </manual_new_material>
      <DoThermalExpansion>            false      </DoThermalExpansion>
      <testForNegTemps_mpm>           false      </testForNegTemps_mpm>
      <erosion algorithm = "ZeroStress"/>
    </MPM>

    <!--__________________________________________________________________-->
    <!--    MATERIAL PROPERTIES INITIAL CONDITIONS                        -->
    <!--__________________________________________________________________-->
    <MaterialProperties>
       <MPM>
         <material name = "Steel Ring">
           <include href="inputs/MPM/MaterialData/MatConst4340St.xml"/>
           <constitutive_model type="elastic_plastic">
             <isothermal>                  false </isothermal>
             <tolerance>                1.0e-10  </tolerance>
             <evolve_porosity>             true  </evolve_porosity>
             <evolve_damage>               true  </evolve_damage>
             <compute_specific_heat>       true  </compute_specific_heat>
             <do_melting>                  true  </do_melting>
             <useModifiedEOS>              true  </useModifiedEOS>
```

```
171             <check_TEPLA_failure_criterion> true  </check_TEPLA_failure_criterion>
172             <initial_material_temperature>  600.0 </initial_material_temperature>
173             <taylor_quinney_coeff>          0.9   </taylor_quinney_coeff>
174             <check_max_stress_failure>      false </check_max_stress_failure>
175             <critical_stress>              12.0e9 </critical_stress>
176
177             <!-- Warning: you must copy link this input file into your -->
178             <!-- sus directory or these paths won't work.              -->
179
180             <include href="inputs/MPM/MaterialData/IsoElastic4340St.xml"/>
181             <include href="inputs/MPM/MaterialData/MieGrunEOS4340St.xml"/>
182             <include href="inputs/MPM/MaterialData/ConstantShear.xml"/>
183             <include href="inputs/MPM/MaterialData/ConstantTm.xml"/>
184             <include href="inputs/MPM/MaterialData/JCPlastic4340St.xml"/>
185             <include href="inputs/MPM/MaterialData/VonMisesYield.xml"/>
186             <include href="inputs/MPM/MaterialData/DruckerBeckerStabilityCheck.xml"/>
187             <include href="inputs/MPM/MaterialData/JCDamage4340St.xml"/>
188             <specific_heat_model type="steel_Cp"> </specific_heat_model>
189
190             <initial_mean_porosity>         0.005 </initial_mean_porosity>
191             <initial_std_porosity>          0.001 </initial_std_porosity>
192             <critical_porosity>             0.3   </critical_porosity>
193             <frac_nucleation>               0.1   </frac_nucleation>
194             <meanstrain_nucleation>         0.3   </meanstrain_nucleation>
195             <stddevstrain_nucleation>       0.1   </stddevstrain_nucleation>
196             <initial_porosity_distrib>      gauss </initial_porosity_distrib>
197
198             <initial_mean_scalar_damage>    0.005  </initial_mean_scalar_damage>
199             <initial_std_scalar_damage>     0.001 </initial_std_scalar_damage>
200             <critical_scalar_damage>        1.0   </critical_scalar_damage>
201             <initial_scalar_damage_distrib> gauss </initial_scalar_damage_distrib>
202           </constitutive_model>
203
204               <geom_object>
205                  <difference>
206                   <cylinder label = "outer cylinder">
207                    <bottom>           [0.0,0.0,-.05715]   </bottom>
208                    <top>              [0.0,0.0, .05715]   </top>
209                    <radius>           0.05715             </radius>
210                  </cylinder>
211                  <cylinder label = "inner cylinder">
212                    <bottom>           [0.0,0.0,-.0508]    </bottom>
213                    <top>              [0.0,0.0, .0508]    </top>
214                    <radius>           0.0508              </radius>
215                  </cylinder>
216                  </difference>
217                <res>                [2,2,2]          </res>
218                <velocity>           [0.0,0.0,0.0]    </velocity>
219                <temperature>        600              </temperature>
220               </geom_object>
221         </material>
222         <material name = "reactant">
223             <include href="inputs/MPM/MaterialData/MatConstPBX9501.xml"/>
224             <constitutive_model type = "visco_scram">
225               <include href="inputs/MPM/MaterialData/ViscoSCRAMPBX9501.xml"/>
226               <include href="inputs/MPM/MaterialData/TimeTempPBX9501.xml"/>
227               <randomize_parameters>          false </randomize_parameters>
228               <use_time_temperature_equation> true  </use_time_temperature_equation>
229               <useObjectiveRate>              true  </useObjectiveRate>
230               <useModifiedEOS>                true  </useModifiedEOS>
231             </constitutive_model>
232                <geom_object>
233                   <difference>
234                     <cylinder label = "inner cylinder"> </cylinder>
235                     <cylinder label = "inner hole">
236                       <bottom>           [0.0,0.0,-.0508]    </bottom>
237                       <top>              [0.0,0.0, .0508]    </top>
238                       <radius>           0.01                </radius>
239                     </cylinder>
240                   </difference>
241                   <res>                [2,2,2]          </res>
242                   <velocity>           [0.0,0.0,0.0]    </velocity>
243                   <temperature>        440.0            </temperature>
```

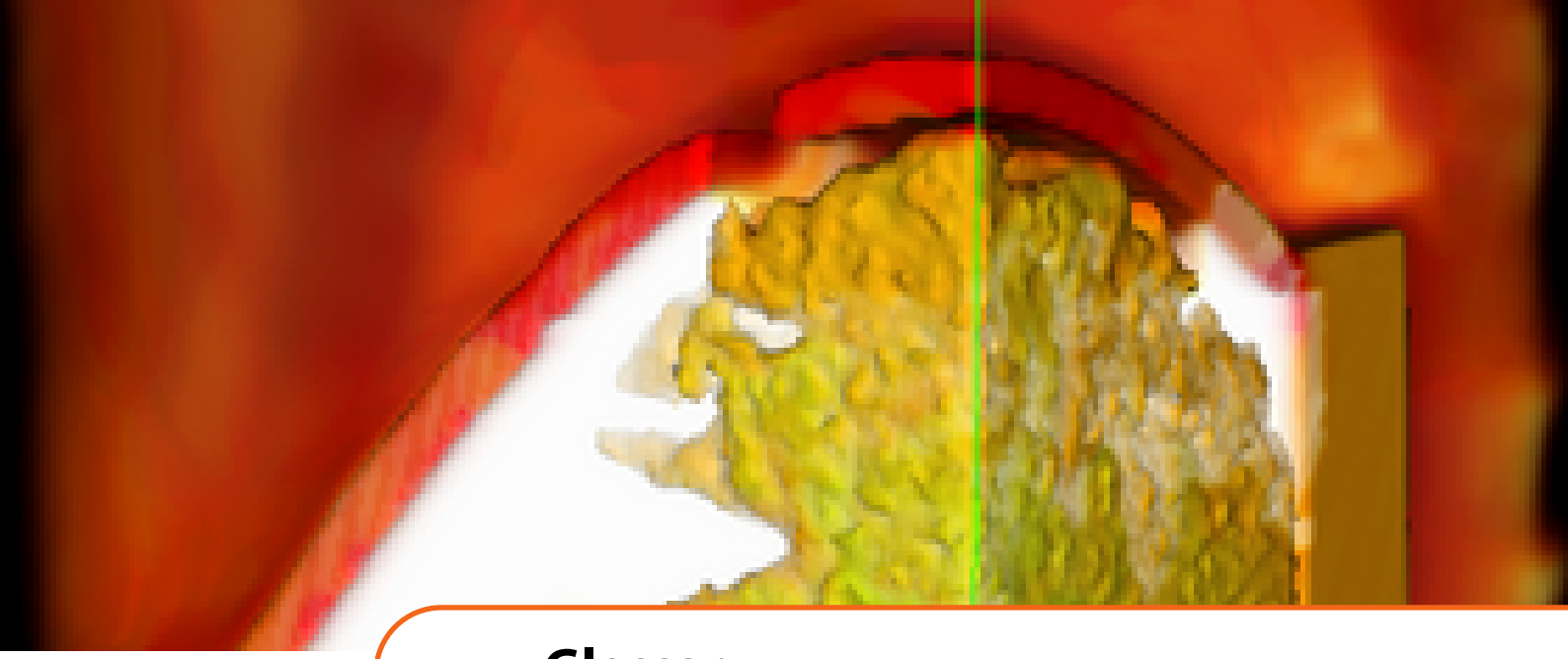
```
244                </geom_object>
245          </material>
246
247             <contact>
248               <type>approach</type>
249               <materials>              [0,1]          </materials>
250               <mu> 0.0 </mu>
251             </contact>
252             <thermal_contact>
253             </thermal_contact>
254       </MPM>
255
256        <ICE>
257          <material>
258            <EOS type = "ideal_gas">
259            </EOS>
260            <dynamic_viscosity>          0.0            </dynamic_viscosity>
261            <thermal_conductivity>       0.0            </thermal_conductivity>
262            <specific_heat>              716.0          </specific_heat>
263            <gamma>                      1.4            </gamma>
264            <geom_object>
265              <difference>
266                   <box>
267                     <min>           [-0.254,-0.254,-0.254] </min>
268                     <max>           [ 0.254, 0.254, 0.254] </max>
269                   </box>
270                 <cylinder label = "outer cylinder"> </cylinder>
271               </difference>
272              <cylinder label="inner hole"> </cylinder>
273              <res>                        [2,2,2]        </res>
274              <velocity>           [0.0,0.0,0.0]       </velocity>
275              <!--
276              <temperature>        300.0               </temperature>
277              <density>    1.1792946927374306000e+00   </density>
278              -->
279              <temperature>        400.0               </temperature>
280              <density>            0.884471019553073   </density>
281              <pressure>           101325.0             </pressure>
282            </geom_object>
283          </material>
284       </ICE>
285
286       <exchange_properties>
287          <exchange_coefficients>
288               <momentum>  [0, 1e15, 1e15]     </momentum>
289               <heat>      [0, 1e10, 1e10]     </heat>
290         </exchange_coefficients>
291       </exchange_properties>
292     </MaterialProperties>
293
294     <AddMaterialProperties>
295       <ICE>
296          <material name = "product">
297            <EOS type = "ideal_gas">
298            </EOS>
299            <dynamic_viscosity>          0.0            </dynamic_viscosity>
300            <thermal_conductivity>       0.0            </thermal_conductivity>
301            <specific_heat>              716.0          </specific_heat>
302            <gamma>                      1.4            </gamma>
303            <geom_object>
304                  <box>
305                    <min>           [ 1.0, 1.0, 1.0] </min>
306                    <max>           [ 2.0, 2.0, 2.0] </max>
307                  </box>
308              <res>                        [2,2,2]        </res>
309              <velocity>           [0.0,0.0,0.0]       </velocity>
310              <temperature>        300.0               </temperature>
311              <density>    1.1792946927374306000e+00   </density>
312              <pressure>           101325.0             </pressure>
313            </geom_object>
314          </material>
315       </ICE>
316
```

```
      <exchange_properties>
         <exchange_coefficients>
              <momentum>  [0, 1e15, 1e15, 1e15, 1e15, 1e15]     </momentum>
              <heat>      [0, 1e10, 1e10, 1e10, 1e10, 1e10]     </heat>
              <!--
              <heat>      [0, 1, 1, 1, 1, 1]     </heat>
              -->
        </exchange_coefficients>
      </exchange_properties>
    </AddMaterialProperties>


    <Models>
      <Model type="Simple_Burn">
        <Active>       false        </Active>
        <fromMaterial> reactant     </fromMaterial>
        <toMaterial>   product      </toMaterial>
        <ThresholdTemp>       450.0 </ThresholdTemp>
        <ThresholdPressure> 50000.0 </ThresholdPressure>
        <Enthalpy>        2000000.0 </Enthalpy>
        <BurnCoeff>            75.3 </BurnCoeff>
        <refPressure>      101325.0 </refPressure>
      </Model>
    </Models>


</Uintah_specification>
```

The PBS script used to run this test is

```

#
# ASK PBS TO SEND YOU AN EMAIL ON CERTAIN EVENTS: (a)bort (b)egin (e)nd (n)ever
#
# (User May Change)

#PBS -m abe

#
# SET THE NAME OF THE JOB:
#
# (User May Change)

#PBS -N ExplodeRing

#
# SET THE QUEUE IN WHICH TO RUN THE JOB.
#      (Note, there is currently only one queue, so you should never change this field.)

#PBS -q defaultq

#
# SET THE RESOURCES (# NODES, TIME) REQESTED FROM THE BATCH SCHEDULER:
#    - select: <# nodes>,ncpus=2,walltime=<time>
#      - walltime: walltime before PBS kills our job.
#                [[hours:]minutes:]seconds[.milliseconds]
#                Examples:
#                  walltime=60      (60 seconds)
#                  walltime=10:00   (10 minutes)
#                  walltime=5:00:00 (5 hours)
#
# (User May Change)

#PBS -l select=2:ncpus=2,walltime=24:00

#
# START UP LAM

cd $PBS_O_WORKDIR
lamboot

# [place your command here] >& ${PBS_O_WORKDIR}/output.${PBS_JOBID}
mpirun -np 4 ../sus_opt explodeRFull.ups >& output.${PBS_JOBID}
```

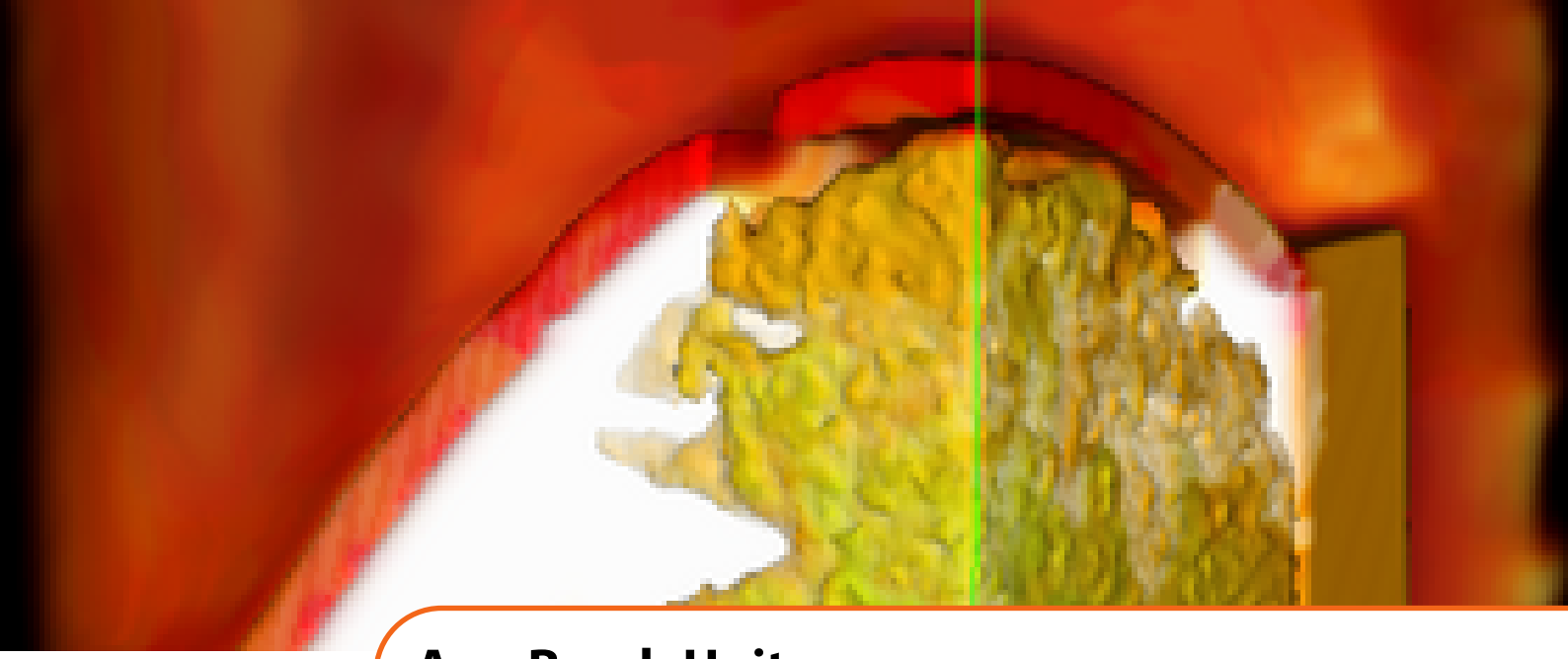
```
44
45 #
46 # REMEMBER, IF YOU ARE RUNNING TWO SERIAL JOBS, YOU NEED A:
47 # wait
48
49 #
50 # STOP LAM
51
52 lamhalt -v
53 exit
```



14 — 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 (physcial 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.

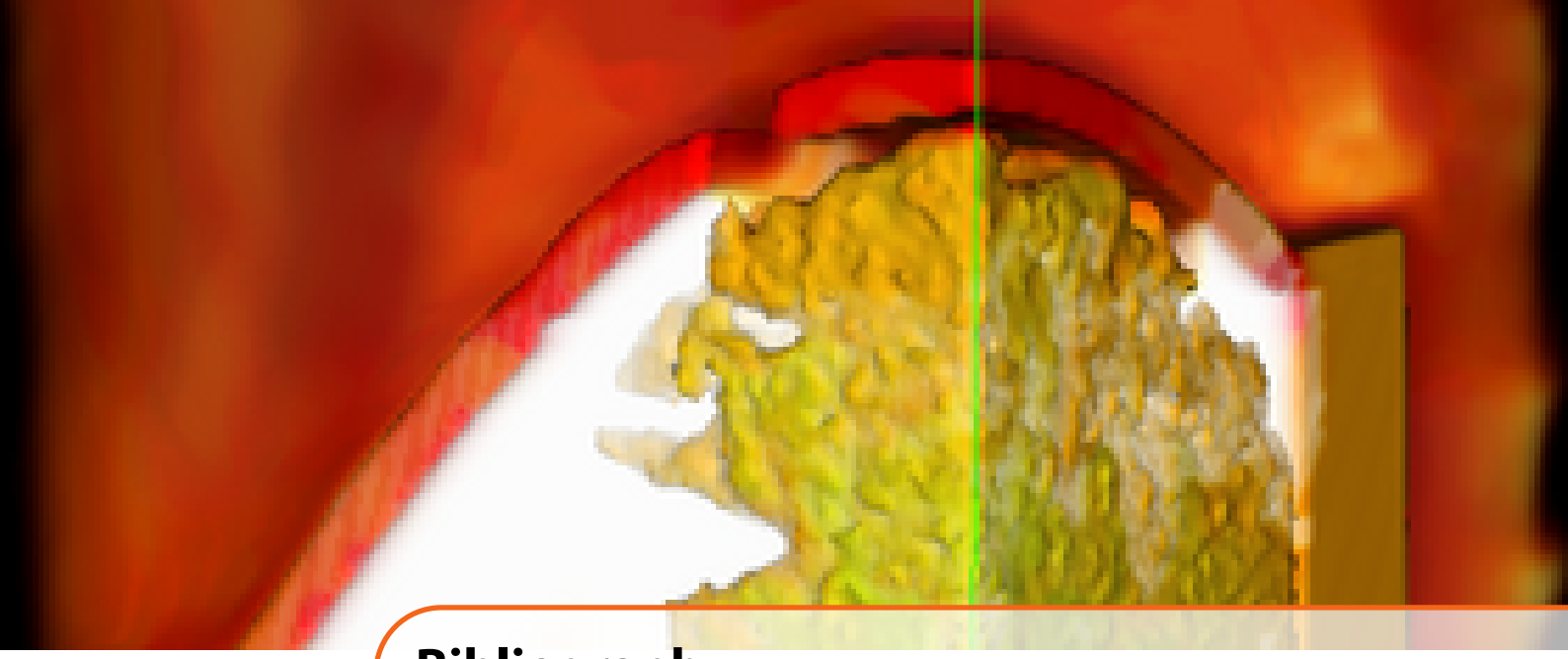
- SCIRun - A Problem Solving Environment (PSE) originally used to provide core software building blocks for Uintah as well as an extensive visualization package for viewing Uintah data archives.
- SUS - Standalone Uintah Simulator. This is the main executable program in the Uintah project.
- SVN - Subversion code versioning system.
- Uintah - The general name of the C-SAFE simulation code. Sometimes also refered 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.



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	μg
length	cm
time	μs
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	$1\frac{J}{kg} = 10^{-8} \frac{cm^2}{\mu s^2}$



Bibliography

[1] S.G. Bardenhagen et al. "An Improved Contact Algorithm for the Material Point Method and Application to Stress Propagation in Granular Material". In: *Computer Modeling in Engineering and Sciences* 2 (2001), pages 509–522 (cited on pages 46, 56).

[2] John A Nairn, Chad C Hammerquist, and Grant D Smith. "New material point method contact algorithms for improved accuracy, large-deformation problems, and proper null-space filtering". In: *Computer Methods in Applied Mechanics and Engineering* 362 (2020), page 112859 (cited on page 46).

[3] N.P. Daphalapurkar et al. "Simulation of dynamic crack growth using the generalized interpolation material point (GIMP) method". In: *Int. J. Fracture* 143 (2007), pages 79–102 (cited on page 52).

[4] D. Sulsky, Z. Chen, and H. L. Schreyer. "A particle method for history dependent materials". In: *Comput. Methods Appl. Mech. Engrg.* 118 (1994), pages 179–196 (cited on page 54).

[5] W. H. Gust. "High impact deformation of metal cylinders at elevated temperatures". In: *J. Appl. Phys.* 53.5 (1982), pages 3566–3575 (cited on page 55).

[6] A.D. Brydon et al. "Simulation of the Densification of Real Open-Celled Foam Microstructures". In: *Journal of the Mechanics and Physics of Solids* 53 (2005), pages 2638–2660 (cited on page 57).

[7] L. Cueto-Felgueroso et al. "On the Galerkin formulation of the smoothed particle hydrodynamics method". In: *Int. J. Numer. Meth. Engng* 60 (2004), pages 1475–1512 (cited on page 64).

[8] JC Simo and TJR Hughes. *Computational Inelasticity*. New York: Springer-Verlag, 1998 (cited on pages 65, 83).

[9] S. Nemat-Nasser. "Rate-independent finite-deformation elastoplasticity: a new explicit constitutive algorithm". In: *Mech. Mater.* 11 (1991), pages 235–249 (cited on page 85).

[10] S. Nemat-Nasser and D. -T. Chung. "An explicit constitutive algorithm for large-strain, large-strain-rate elastic-viscoplasticity". In: *Comput. Meth. Appl. Mech. Engrg* 95.2 (1992), pages 205–219 (cited on page 85).

[11] 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". In: *Computational Mechanics* 13 (1994), pages 380–389 (cited on page 85).

[12] P. J. Maudlin and S. K. Schiferl. "Computational anisotropic plasticity for high-rate forming applications". In: *Comput. Methods Appl. Mech. Engrg.* 131 (1996), pages 1–30 (cited on page 85).

[13] M. A. Zocher et al. "An evaluation of several hardening models using Taylor cylinder impact data". In: *Proc., European Congress on Computational Methods in Applied Sciences and Engineering*. ECCOMAS. Barcelona, Spain, 2000 (cited on pages 85, 91, 97).

[14] G. Ravichandran et al. "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*. American Physical Society. 2001, pages 557–562 (cited on page 88).

[15] 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*. APS. Portland, Oregon, 2003 (cited on page 88).

[16] F. L. Lederman, M. B. Salamon, and L. W. Shacklette. "Experimental verification of scaling and test of the universality hypothesis from specific heat data". In: *Phys. Rev. B* 9.7 (1974), pages 2981–2988 (cited on page 89).

[17] D. J. Steinberg. *Equation of state and strength properties of selected materials*. Technical report UCRL-MA-106439. Livermore, California: Lawrence Livermore National Laboratory, 1991 (cited on page 91).

[18] M. L. Wilkins. *Computer Simulation of Dynamic Phenomena*. Berlin: Springer-Verlag, 1999 (cited on pages 91, 93).

[19] F. J. Zerilli and Armstrong R. W. "A constitutive equation for the dynamic deformation behavior of polymers". In: *J. Mater. Sci.* 42 (2007), pages 4562–4574 (cited on pages 94, 103).

[20] J.-P Poirier. *Introduction to the Physics of the Earth's Interior*. Cambridge, UK: Cambridge University Press, 1991 (cited on page 95).

[21] D. J. Steinberg, S. G. Cochran, and M. W. Guinan. "A constitutive model for metals applicable at high-strain rate". In: *J. Appl. Phys.* 51.3 (1980), pages 1498–1504 (cited on pages 95, 97, 101).

[22] L. Burakovsky and D. L. Preston. "Analysis of dislocation mechanism for melting of elements". In: *Solid State Comm.* 115 (2000), pages 341–345 (cited on page 96).

[23] Y. P. Varshni. "Temperature dependence of the elastic constants". In: *Physical Rev. B* 2.10 (1970), pages 3952–3958 (cited on page 97).

[24] S. R. Chen and G. T. Gray. "Constitutive behavior of tantalum and tantalum-tungsten alloys". In: *Metall. Mater. Trans. A* 27A (1996), pages 2994–3006 (cited on page 97).

[25] 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". In: *J. Appl. Phys.* 93.5 (2003), pages 2472–2480 (cited on page 97).

[26] A. L. Gurson. "Continuum theory of ductile rupture by void nucleation and growth: Part 1. Yield criteria and flow rules for porous ductile media". In: *ASME J. Engg. Mater. Tech.* 99 (1977), pages 2–15 (cited on page 99).

[27] V. Tvergaard and A. Needleman. "Analysis of the cup-cone fracture in a round tensile bar". In: *Acta Metall.* 32.1 (1984), pages 157–169 (cited on page 99).

[28] S. Ramaswamy and N. Aravas. "Finite element implementation of gradient plasticity models Part II: Gradient-dependent evolution equations". In: *Comput. Methods Appl. Mech. Engrg.* 163 (1998), pages 33–53 (cited on page 100).

[29] C. C. Chu and A. Needleman. "Void nucleation effects in biaxially stretched sheets". In: *ASME J. Engg. Mater. Tech.* 102 (1980), pages 249–256 (cited on page 100).

[30] 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*. 1983, pages 541–547 (cited on page 101).

[31] D. J. Steinberg and C. M. Lund. "A constitutive model for strain rates from 10^{-4} to 10^{6} s^{-1}". In: *J. Appl. Phys.* 65.4 (1989), pages 1528–1533 (cited on page 101).

[32] K. G. Hoge and A. K. Mukherjee. "The temperature and strain rate dependence of the flow stress of tantalum". In: *J. Mater. Sci.* 12 (1977), pages 1666–1672 (cited on page 102).

[33] F. J. Zerilli and R. W. Armstrong. "Dislocation-mechanics-based constitutive relations for material dynamics calculations". In: *J. Appl. Phys.* 61.5 (1987), pages 1816–1825 (cited on page 102).

[34] F. J. Zerilli and R. W. Armstrong. "Constitutive relations for the plastic deformation of metals". In: *High-Pressure Science and Technology - 1993*. American Institute of Physics. Colorado Springs, Colorado, 1993, pages 989–992 (cited on page 102).

[35] F. J. Zerilli. "Dislocation mechanics-based constitutive equations". In: *Metall. Mater. Trans. A* 35A (2004), pages 2547–2555 (cited on page 102).

[36] F. H. Abed and G. Z. Voyiadjis. "A consistent modified Zerilli-Armstrong flow stress model for bcc and fcc metals for elevated temperatures". In: *Acta Mechanica* 175 (2005), pages 1–18 (cited on page 103).

[37] 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". In: *Acta Metall.* 36 (1988), pages 82–93 (cited on page 104).

[38] D. M. Goto et al. "Anisotropy-corrected MTS constitutive strength modeling in HY-100 steel". In: *Scripta Mater.* 42 (2000), pages 1125–1131 (cited on page 104).

[39] U. F. Kocks. "Realistic constitutive relations for metal plasticity". In: *Materials Science and Engrg.* A317 (2001), pages 181–187 (cited on page 104).

[40] D. L. Preston, D. L. Tonks, and D. C. Wallace. "Model of plastic deformation for extreme loading conditions". In: *J. Appl. Phys.* 93.1 (2003), pages 211–220 (cited on page 105).

[41] B. Banerjee. "Material Point Method simulations of fragmenting cylinders". In: *Proc. 17th ASCE Engineering Mechanics Conference (EM2004)*. Newark, Delaware, 2004 (cited on page 107).

[42] B. Banerjee. *MPM Validation: Sphere-Cylinder Impact Tests: Energy Balance*. Technical report C-SAFE-CD-IR-04-001. University of Utah, USA: Center for the Simulation of Accidental Fires and Explosions, 2004. URL: http://www.csafe.utah.edu/documents/C-SAFE-CD-IR-04-001.pdf (cited on page 107).

[43] B. Banerjee. "Validation of UINTAH: Taylor impact and plasticity models". In: *Proc. 2005 Joint ASME/ASCE/SES Conference on Mechanics and Materials (McMat 2005)*. Baton Rouge, LA, 2005 (cited on page 107).

[44] G. R. Johnson and W. H. Cook. "Fracture characteristics of three metals subjected to various strains, strain rates, temperatures and pressures". In: *Int. J. Eng. Fract. Mech.* 21 (1985), pages 31–48 (cited on page 107).

[45] G. R. Johnson and T. J. Holmquist. "Evaluation of cylinder-impact test data for constitutive models". In: *J. Appl. Phys.* 64.8 (1988), pages 3901–3910 (cited on page 107).

[46] D. C. Drucker. "A definition of stable inelastic material". In: *J. Appl. Mech.* 26 (1959), pages 101–106 (cited on page 108).

[47] J. W. Rudnicki and J. R. Rice. "Conditions for the localization of deformation in pressure-sensitive dilatant materials". In: *J. Mech. Phys. Solids* 23 (1975), pages 371–394 (cited on page 108).

[48] P. Perzyna. "Constitutive modelling of dissipative solids for localization and fracture". In: *Localization and Fracture Phenomena in Inelastic Solids: CISM Courses and Lectures No. 386*. Edited by Perzyna P. New York: SpringerWien, 1998, pages 99–241 (cited on page 108).

[49] R. Becker. "Ring fragmentation predictions using the Gurson model with material stability conditions as failure criteria". In: *Int. J. Solids Struct.* 39 (2002), pages 3555–3580 (cited on page 108).

[50] R. Hill and J. W. Hutchinson. "Bifurcation phenomena in the plane tension test". In: *J. Mech. Phys. Solids* 23 (1975), pages 239–264 (cited on page 108).

[51] Z. P. Bazant and T. Belytschko. "Wave propagation in a strain-softening bar: Exact solution". In: *ASCE J. Engg. Mech* 111.3 (1985), pages 381–389 (cited on page 108).

[52] V. Tvergaard and A. Needleman. "Ductile failure modes in dynamically loaded notched bars". In: *Damage Mechanics in Engineering Materials: AMD 109/MD 24*. Edited by J. W. Ju, D. Krajcinovic, and H. L. Schreyer. New York, NY: American Society of Mechanical Engineers, 1990, pages 117–128 (cited on page 108).

[53] S. Ramaswamy and N. Aravas. "Finite element implementation of gradient plasticity models Part I: Gradient-dependent yield functions". In: *Comput. Methods Appl. Mech. Engrg.* 163 (1998), pages 11–32 (cited on page 108).

[54] S. Hao, W. K. Liu, and D. Qian. "Localization-induced band and cohesive model". In: *J. Appl. Mech.* 67 (2000), pages 803–812 (cited on page 108).

[55] R. 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". In: *International Journal of Fracture* 163.1 (2010), pages 133–149 (cited on page 110).

[56] R.D. Falgout, J.E. Jones, and U.M. Yang. "The Design and Implementation of hypre, a Library of Parallel High Performance Preconditioners". In: *Numerical Solution of Partial Differential Equations on Parallel Computers*. Edited by A.M. Bruaset and A. Tveito. Lecture Notes in Computational Science and Engineering. Springer-Verlag, 2006 (cited on page 135).

[57] J. S. Thomsen and T. J. Hartka. "Strange Carnot cycles; thermodynamics fo a system with a density extremum". In: *Am. J. Phys.* 30 (1962), pages 26–33 (cited on page 137).

[58] A. Bejan. *Advanced Engineering Thermodynamics*. USA: John Wiley and Sons, 1988, pages 724–725 (cited on page 137).

[59] P. C. Souers S. Anderson J. Mercer E. McGuire and P. Vitello. "JWL++: A Simple Reactive Flow Code Package for Detonation". In: *Propellants, Explosives, Pyrotechnics* 25 (2000), pages 54–58 (cited on pages 137, 154, 166–168).

[60] P. A. Thompson. *Compressible-Fluid Dynamics*. New York: McGraw-Hill, 1988 (cited on page 138).

[61] G. R. Gathers. *Selected Topics in Shock Wave Physics and Equation of State Modeling*. River Edge, NJ: World Scientific, 1994 (cited on page 138).

[62] J. C. Sutherland and Kenndey C.A. "Improved Boundary conditions for viscous, reacting compressible flows". In: *Journal of Computational Physics* 191 (2003), pages 502–524 (cited on page 141).

[63] F. M. White. *Viscous Fluid Flow, 2nd Edition*. McGraw-Hill, 1991 (cited on page 146).

[64] C. B. Laney. *Computational Gasdynamics*. Cambridge: Cambridge University Press, 1998 (cited on page 148).

[65] G. A. Sod. "A Survey of Several Finite Difference Methods for Systems of Nonlinear Hyperbolic Conservation Laws". In: *J. Comput. Phys* 27 (1978), pages 1–31 (cited on page 148).

[66] E. F. Toro. *Riemann Solvers and Numerical Methods for Fluid Dynamics*. Berlin Heidelberg: Springer, 1997 (cited on pages 148, 152).

[67] J. P. Collins C. W. Schulz-Rinne and H. M. Glaz. "Numerical Solution of the Riemann Problem for Two-Dimensional Gas Dynamics". In: *J. Comput. Phys* 14 (1993), pages 1394–1414 (cited on page 150).

[68] R. Liska and B. Wendroff. "Comparison of several difference schemes on 1D and 2D test problems for the Euler Equations". In: *J. Comput. Phys* 25 (2003), pages 995–1017 (cited on page 150).

[69] J.E. Guilkey, T.B. Harman, and B. Banerjee. "An Eulerian-Lagrangian Approach for Simulating Explosions of Energetic Devices". In: *Computers and Structures* 85 (2007), pages 660–674 (cited on page 157).

[70] V. Sergey and C.A. Wight. "Isothermal and Nonisothermal reaction Kinetics in Solids: In Search of Ways toward Consensus". In: *Journal of Physical Chemistry A* 101 (1997), pages 8279–8284 (cited on page 158).

[71] A. Khawam and D.R. Flanagan. "Basics and application of solid-stat kinetics: A pharmaceutical perspective". In: *Journal of Pharmaceutical Science* 95 (2006), pages 472–498 (cited on page 158).

[72] M.J. Ward, S.F. Son, and M.Q. Brewster. "Steady Deflagration of HMX With Simple Kinetics: A Gas Phase Chain Reaction Model". In: *Combustion and Flame* 114 (1998), pages 556–568 (cited on pages 160, 170).

[73] C.A. Wight and E.G. Eddings. "Science-Based Simulation Tools for Hazard Assessment and Mitigation". In: *Advancements in Energetic Materials and Chemical Propulsion* (2008), pages 921–937 (cited on pages 161, 163, 177).

[74] E.L. Lee and C.M. Tarver. "Phenomenological Model of Shock Initiation in Heterogeneouse Explosives". In: *Physics of Fluids* (1980), pages 2362–2372 (cited on page 165).

[75] H.L. Berghout et al. "Combustion of damaged PBX 9501 explosive". In: *Thermochimica Acta* (2002), pages 261–277 (cited on pages 167, 168).

[76] J.R. Peterson and C.A. Wight. "An Eulerian-Lagrangian computational model for deflagration and detonation of high explosives". In: *Combustion and Flame* (2012), pages 2491–2499 (cited on page 168).

[77] S. F. Son et al. "Flame spread across surfaces of PBX 9501". In: *Combustion Institute*. Edited by Elsevier. Volume 31. 2007, pages 2063–2070 (cited on page 170).

[78] M. A. Saad. *Compressible Fluid Flow*. New Jersey: Prentice-Hall, 1985 (cited on page 171).

[79] J.C. Finlinson, R. Stalnaker, and Blomshield F.S. "HMX and RDX T-burner pressure coupled response at 200, 500, and 1000 psi". In: 1999 (cited on page 179).