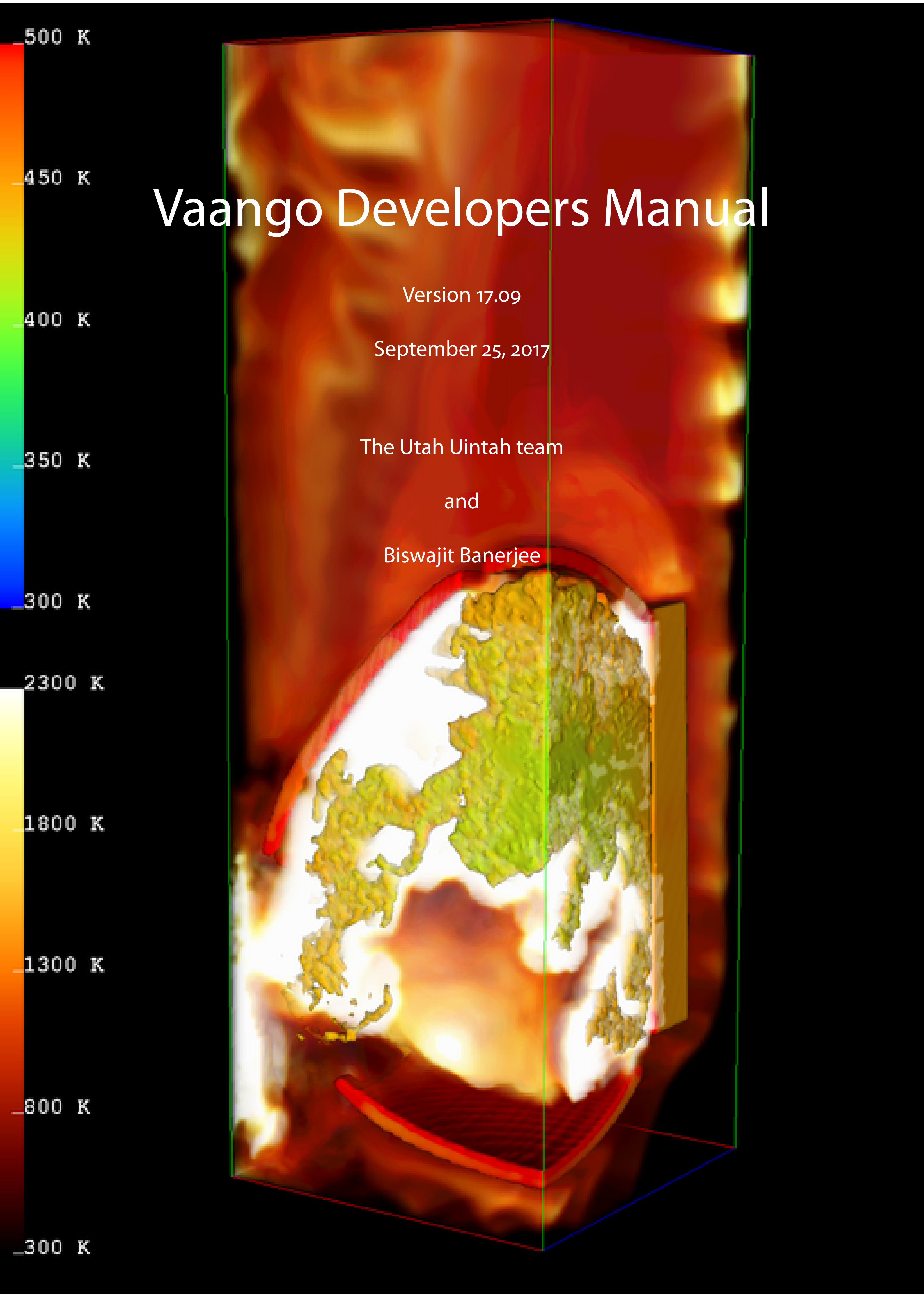


Vaango Developers Manual

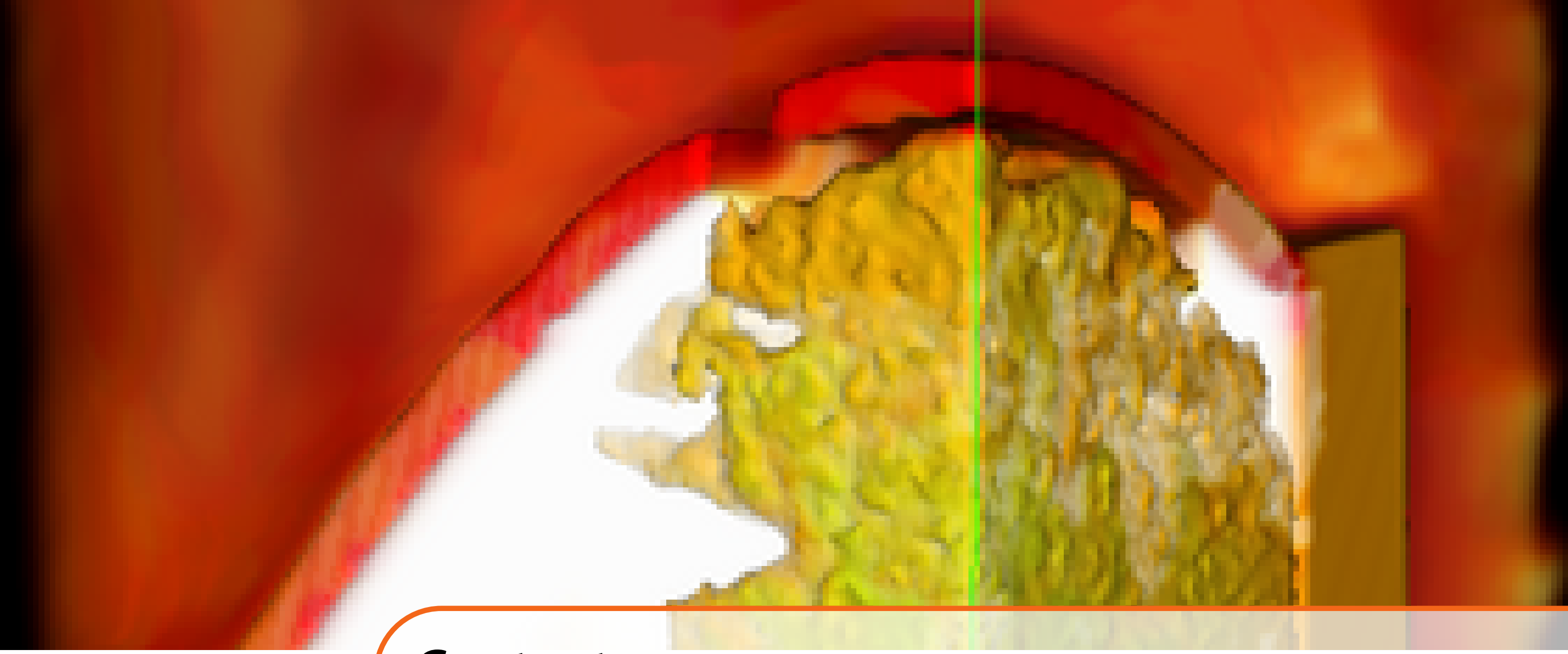
Version 17.09

September 25, 2017

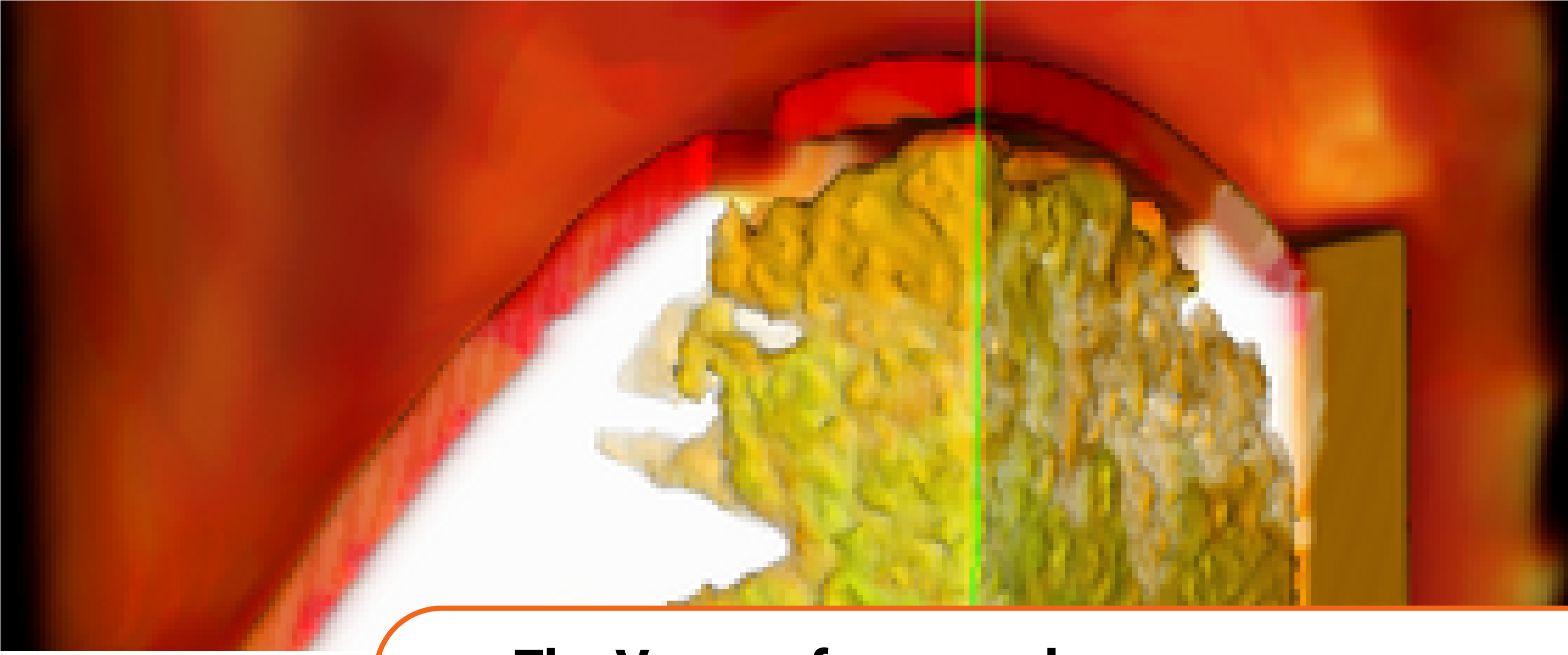
The Utah Uintah team

and

Biswajit Banerjee



Contents



1 — The Vaango framework

1.1 Historical information

Vaango is a fork of the Uintah Computational Framework (Uintah) created in 2012 to allow for the development of tools to solve solid mechanics problems in mechanical and civil engineering. The orginal Uintah code continues to be actively developed, but the focus of that code is multiphysics problems, particularly computational fluid dynamics (CFD) and chemical engineering. Around once a year, the underlying parallel computing infrastructure of Vaango is updated to keep up with developments in Uintah .

The Vaango framework, like Uintah , consists of a set of software components and libraries that facilitate the solution of Partial Differential Equations (PDEs) on Structured AMR (SAMR) grids using hundreds to thousands of processors. However, unlike the CFD problems that Uintah was designed for, the use of the grid is often only incidental to the solution of the governing PDEs of solid mechanics.

1.2 Overview

One of the challenges in designing a parallel, component-based 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.

Vaango uses a non-traditional approach to achieving parallelism, employing an abstract taskgraph representation to describe computation and communication. The taskgraph 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) see figure 1.1. Vaango components delegate decisions about parallelism to a scheduler component, using variable dependencies to describe communication patterns and characterizing computational workloads to facilitate a global resource optimization. The taskgraph 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.

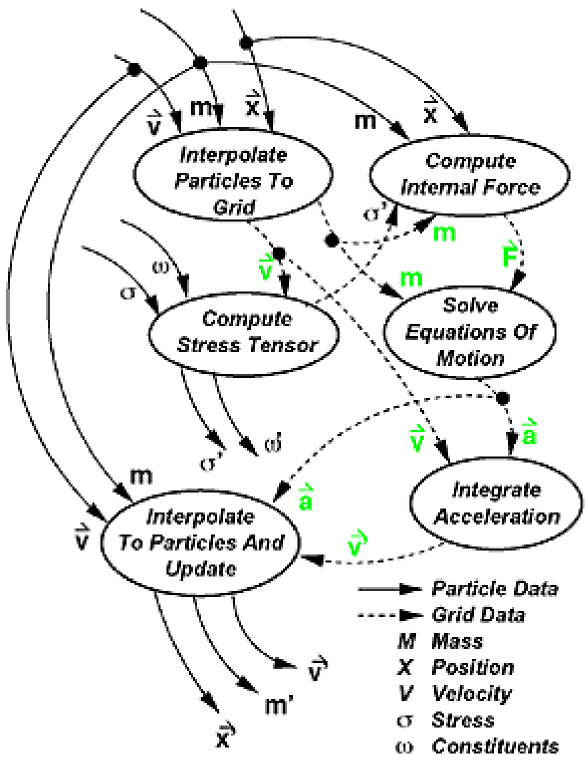


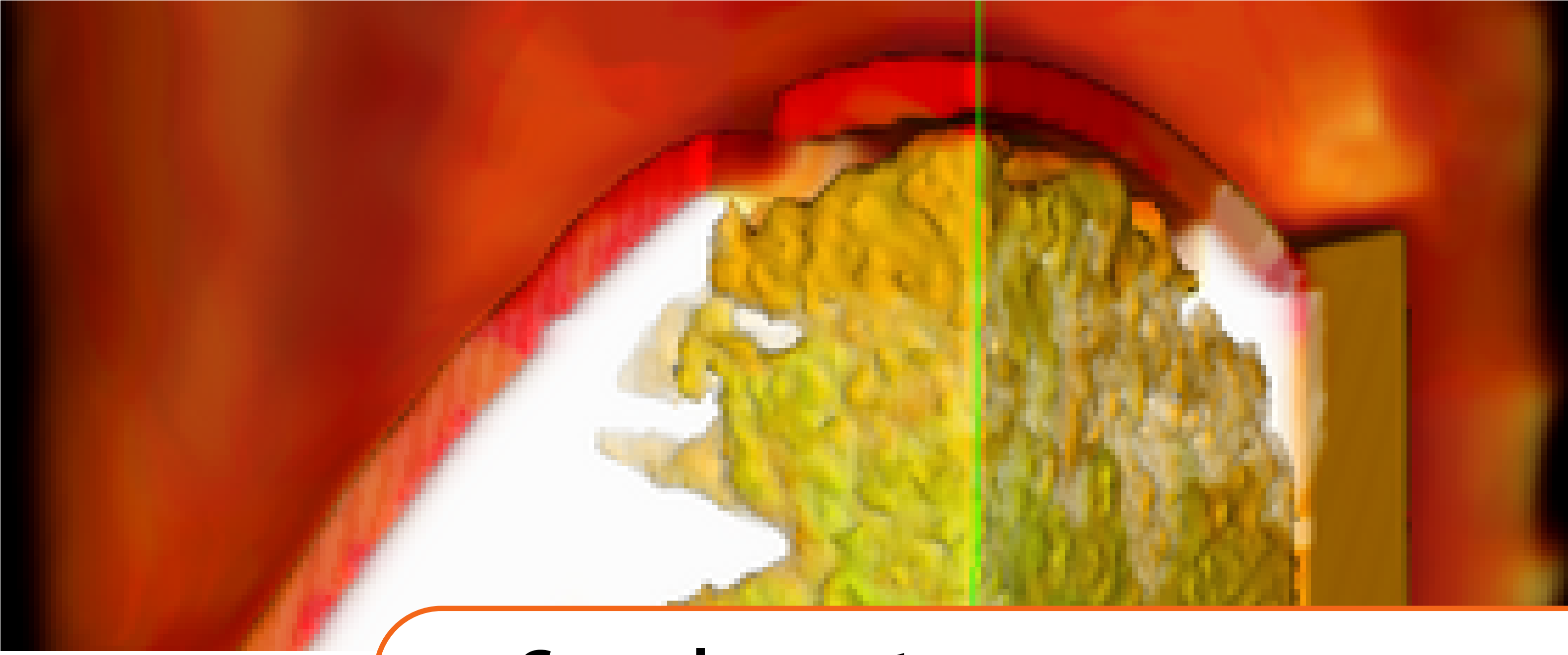
Figure 1.1: Example Task Graph

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, Vaango 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.

1.3 An example

The sequence of steps performed in a simulation can be seen in the abbreviated example below. A complete version of this example can be found in the unit test for TabularPlasticity located at src/CCA/Components/MPM/ConstitutiveModel/UnitTests/testTabularPlasticity.cc .

```
try {
    // Read the input file
    ProblemSpecP ups = VaangoEnv::createInput();
    ups->getNode()->_private = (void *) ups_file.c_str();

    // Create the MPI/threading environment
    const ProcessorGroup* world = Uintah::Parallel::getRootProcessorGroup();

    // Create the simulation controller
    SimulationController* ctl = scinew AMRSimulationController(world, false,
        ups);

    // Create a regridder if needed
    RegridderCommon* reg = 0;

    // Create an implicit solver if needed
    SolverInterface* solve = SolverFactory::create(ups, world, "");

    // Create the component for the simulation (MPM/MPMICE/Peridynamics)
    UintahParallelComponent* comp = ComponentFactory::create(ups, world, false,
         "");

    // Create the simulation base object
    SimulationInterface* sim = dynamic_cast<SimulationInterface*>(comp);
    ctl->attachPort("sim", sim);
    comp->attachPort("solver", solve);
    comp->attachPort("regridder", reg);

    // Create a load balancer
    LoadBalancerCommon* lbc = LoadBalancerFactory::create(ups, world);
    lbc->attachPort("sim", sim);

    // Create a data archiver
    DataArchiver* dataarchiver = scinew DataArchiver(world, -1);
    Output* output = dataarchiver;
    ctl->attachPort("output", dataarchiver);
    dataarchiver->attachPort("load balancer", lbc);
    comp->attachPort("output", dataarchiver);
    dataarchiver->attachPort("sim", sim);

    // Create a task scheduler
    SchedulerCommon* sched = SchedulerFactory::create(ups, world, output);
    sched->attachPort("load balancer", lbc);
    ctl->attachPort("scheduler", sched);
    lbc->attachPort("scheduler", sched);
    comp->attachPort("scheduler", sched);
    sched->setStartAddr( start_addr );
    sched->addReference();

    // Run the simulation
    ctl->run();

    // Clean up after the simulation is complete
    delete ctl;
    sched->removeReference();
    delete sched;
    delete lbc;
    delete sim;
    delete solve;
    delete output;

  } catch (ProblemSetupException& e) {
    std::cout << e.message() << std::endl;
    thrownException = true;
  } catch (Exception& e) {
    std::cout << e.message() << std::endl;
    thrownException = true;
  } catch (...) {
    std::cout << "**ERROR** Unknown exception" << std::endl;
    thrownException = true;
  }
```



2 — General concepts

This chapter discusses the main concepts used in a Vaango simulation. These concepts are independent of the type of problem being considered and are core to the computational framework. Special treatment is needed for components such as MPM or Peridynamics.

2.1 Scheduler

The Scheduler in Vaango is responsible for determining the order of tasks and ensuring that the correct inter-processor data is made available when necessary. Each software component passes a set of tasks to the scheduler. Each task is responsible for computing some subset of variables, and may require previously computed variables, possibly from different processors. The scheduler will then compile this task information into a task graph, and the task graph will contain a sorted order of tasks, along with any information necessary to perform inter-process communication via MPI or threading. Then, when the scheduler is executed, the tasks will execute in the pre-determined order.

2.1.1 needRecompile()

Each component has a needRecompile() function that is called once per timestep. If, for whatever reason, a Component determines that the list of tasks it had previously scheduled is no longer valid, then the Component must return 'true' when its needRecompile() function is called. This will cause the scheduler to rebuild the task graph (by asking each component to re-specify tasks). Note, rebuilding the taskgraph is a relatively expensive operation, so only should be done if necessary.

2.2 Tasks

A task contains two essential components: a pointer to a function that performs the actual computations, and the data inputs and outputs, i.e. the data dependencies required by the function. When a task requests a previously computed variable from the data warehouse, the number of ghost cells are also specified. The Unitah framework uses the ghost cell information to excecute inter-process communication to retrieve the necessary ghost cell data.

An example of a task description is presented showing the essential features that are commonly used by the application developer when implementing an algorithm within the Vaango framework. The task component is assigned a name and in this particular example, it is called taskexample and a function pointer, &Example::taskexample . Following the instantiation of the task itself, the dependency information is assigned to the tasks. In the following example, the task requires data from the previous timestep (Task::OldDW) associated with the name variable1_label and requires one ghost node (Ghost::AroundNodes,1) level of information which will be retrieved from another processor via MPI. In addition, the task will compute two new pieces of data each associated with different variables, i.e. variable1_label , and variable2_label . Finally, the task is added to the scheduler component with specifications about what patches and materials are associated with the actual computation.

```
Task* task = scinew Task("Example::taskexample",this, &Example::taskexample);
task->requires(Task::OldDW, variable1_label, Ghost::AroundNodes, 1);
task->computes(variable1_label);
task->computes(variable2_label);
sched->addTask(task, level->eachPatch(), sharedState_->allMaterials());
```

For more complex problems involving multiple materials and multi-physics calculations, a subset of the materials may only be used in the calculation of particular tasks. The VAANGO framework allows for the independent scheduling and computation of multi-material within a multi-physics calculation.

2.3 Simulation Component Class Description

Each VAANGO component can be described as a C++ class that is derived from two other classes: UintahParallelComponent and a SimulationInterface . The new derived class must provide the following virtual methods: problemSetup , scheduleInitialize , scheduleComputeStableTimestep , and scheduleTimeAdvance . Here is an example of the typical *.h file that needs to be created for a new component.

```
class Example : public UintahParallelComponent, public SimulationInterface {
  public:

    virtual void problemSetup(const ProblemSpecP& params, const ProblemSpecP&
         restart_prob_spec, GridP& grid, SimulationStateP&);

    virtual void scheduleInitialize(const LevelP& level,SchedulerP& sched);

    virtual void scheduleComputeStableTimestep(const LevelP& level,
        SchedulerP&);

    virtual void scheduleTimeAdvance(const LevelP& level, SchedulerP&);

  private:
    Example(const ProcessorGroup* myworld);
    virtual ~Example();


    void initialize(const ProcessorGroup*, const PatchSubset* patches, const
        MaterialSubset* matls, DataWarehouse* old_dw, DataWarehouse* new_dw);


    void computeStableTimestep(const ProcessorGroup*, const PatchSubset*
        patches, const MaterialSubset* matls, DataWarehouse* old_dw,
        DataWarehouse* new_dw);

    void timeAdvance(const ProcessorGroup*, const PatchSubset* patches, const
         MaterialSubset* matls, DataWarehouse* old_dw, DataWarehouse* new_dw);
}
```

Each new component inherits from the classes UintahParallelComponent and SimulationInterface . The component overrides default implementations of various methods. The above methods are the essential functions that a new component must implement. Additional methods to do AMR will be described as more complex examples are presented.

The roles of each of the scheduling methods are described below. Each scheduling method, i.e. scheduleInitialize , scheduleComputeStableTimestep , and scheduleTimeAdvance describe

2.3.1 ProblemSetup

The purpose of this method is to read a problem specification which requires a minimum of information about the grid used, time information, i.e. time step size, length of time for simulation, etc, and where and what data is actually saved. Depending on the problem that is solved, the input file can be rather complex, and this method would evolve to establish any and all parameters needed to initially setup the problem.

2.3.2 ScheduleInitialize

The purpose of this method is to initialize the grid data with values read in from the problemSetup and to define what variables are actually computed in the TimeAdvance stage of the simulation. A task is defined which references a function pointer called initialize .

2.3.3 ScheduleComputeStableTimestep

The purpose of this method is to compute the next timestep in the simulation. A task is defined which references a function pointer called computeStableTimestep .

2.3.4 ScheduleTimeAdvance

The purpose of this method is to schedule the actual algorithmic implementation. For simple algorithms, there is only one task defined with a minimal set of data dependencies specified. However, for more complicated algorithms, the best way to schedule the algorithm is to break it down into individual tasks. Each task of the algorithm will have its own data dependencies and function pointers that reference individual computational methods.

2.4 Data Storage Concepts

During the course of the simulation, data is computed and stored in a data structure called the DataWarehouse. The DataWarehouse is an abstraction for storage, retrieval, and access of VAANGO simulation data. The Data warehouse presents a localized view of the global data distribution.

2.4.1 Data Archiver

The Data Archiver is a component of the framework that allows for reading, saving, and accessing simulation data. It presents a global shared memory abstraction to the simulation data. The component developer does not have to worry about retrieving data that has been produced on a remote processor or sending data from a local processor to another processor for use. The framework takes care of these tasks implicitly for the component.

2.4.2 Two Data Warehouses

During each time step (assuming a single level problem), there are two data warehouses associated with the simulations. The `new_dw` and the `old_dw` (as they are commonly referred to in the code). The `old_dw` contains data that was generated in the previous timestep, and may not be modified. Data that is generated during the current timestep will be placed in the `new_dw` . At the end of a timestep, the current `old_dw` is deleted, and then replaced with the current `new_dw` . Then a new (empty) `new_dw` is created.

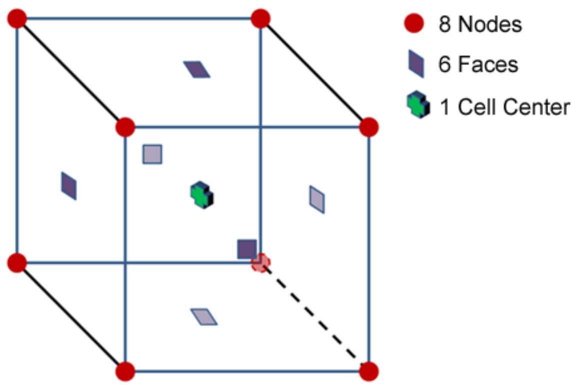


Figure 2.1: Variable locations with respect to a single cell.

At the end of a time step, some data is automatically transferred from the current time step's `new_dw` into the next time step's `old_dw`.

2.4.3 Storing and Retrieving Variables

In order to store data to the data warehouse, or to retrieve data, several things are required. First, the task (which procedure wishes access to the data) must have registered this information with the Scheduler during task creation time. Then, inside of the task, the following call is used to pull the data from the data warehouse:

```
SFCXVariable<double> uVelocity;
new_dw->get( uVelocity, d_lab->d_uVelocitySPBCLabel, matlIndex, patch, Ghost
    ::AroundFaces, Arches::ONEGHOSTCELL );
```

Similarly, to put data into the data warehouse, use this call:

```
PerPatch<CellInformationP> cellInfoP;
new_dw->put( cellInfoP, d_lab->d_cellInfoLabel, matlIndex, patch );
```

2.4.4 Variables

There are three (general) types of variables in Vaango: Particle, Grid, and Reduction. Each type of variable is discussed below. Remember, in order to interact with the Data Warehouse, each variable must have a corresponding label.

Particle variables contain information about particles.

A grid variable is a representation of data across (usually) the entire computational domain. It can be used to represent temperature, volume, velocity, etc. It is implemented using a 3D array.

A reduction, in terms of distributed software (using, for example, MPI) is a point in which all (or some defined set of) processors all communicate with each other. It usually is an expensive operation (because it requires all processors to synchronize with each other and pass data). However, many times this operation cannot be avoided. Vaango provides a "Reduction Variable" to facilitate this operation. Common reductions operations include finding the min or max of a single number on all processors, or creating the sum of a number from every processor, and returning the result to all processors.

Variables can be stored at several different locations. Variables are defined with respect to how they relate to a single cell in the computational domain. They may be located on the faces (FC) or nodes (NC) of the cell, or in the center (CC) of the cell (see Figure 2.1).

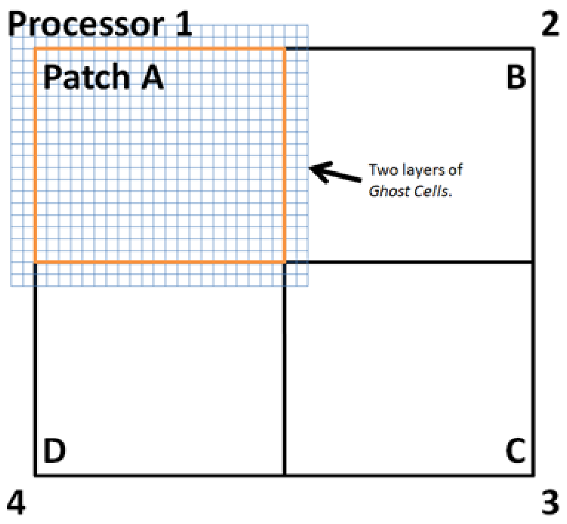


Figure 2.2: Schematic of ghost cell locations laid out with respect to patches.

2.5 Grid data

Data (Variables) used by the framework are stored on the Grid . The Grid is comprised of one or more (in the case of AMR) Levels. Each level contains one or more patches; and each patch contains a number of cells. The simulation data itself exists in (at) each cell (and is stored in a 'variable' which is implemented as a 3D array of the data in each cell across the patch).

2.5.1 Cells

Data in (at) each cell can be specified in several ways. Specifically the data can be Node centered (NC), Cell centered (CC), or Face centered (FC). For CC data, there is one value associated with each cell; for NC data, 4 values; and for FC data, 6 values.

2.5.2 Patches

For calculations to take place on a patch, information from bordering patches is required. This information is stored in boundary cells.

2.5.3 Boundary Cells

Boundary cells (Figure 2.2) represent portions of the computational domain outside the boundaries of the data assigned to a given processor. Specifically, they are almost always used as boundary cells to a patch that are necessary for computation, but are not "owned" by the patch that is currently being computed. As can be seen in Figure 2.2, patch A owns all the cells within the orange rectangle, but also has two layers of ghost cells. Portions of these cells are actually owned by patches B, C, and D. (The data found in these cells is automatically transfered (by the framework) from processors 2, 3, and 4 to processor 1 in order for the cell data to be available for use in computations on patch A.)

Many operations will require a stencil consisting of several cells in order to perform calculations at a given cell. However, at the border of a patch, there are no cells belonging to that patch that contain the required information - that information is "owned" by another processor. In this situation, data that belongs to another patch must be accessed. This is the purpose of ghost cells. The Data Warehouse takes care of moving the required ghost data from the "owner" processor to the neighbor processor so that the individual task can assume the required data is available.

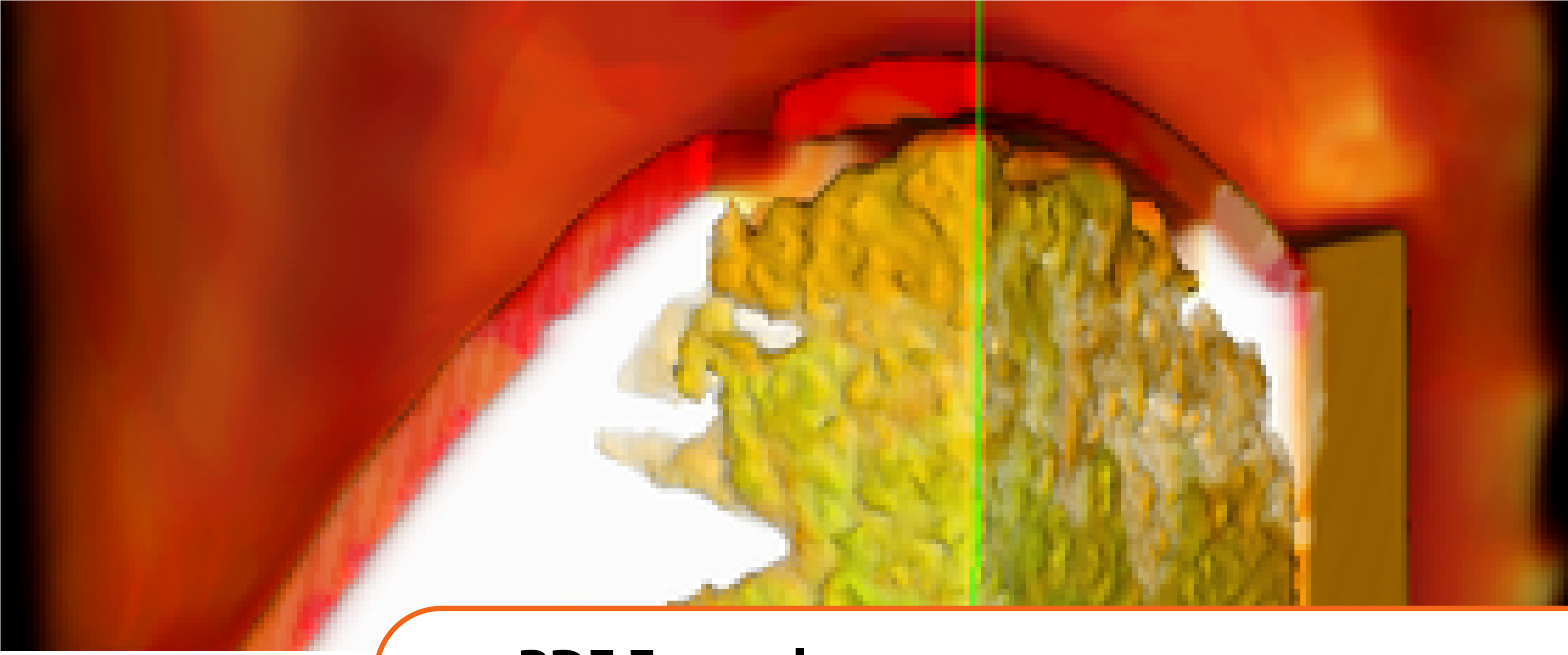
In summary, ghost cells exist between patches on the same level. They are cells that are owned by the neighboring patch but are required for computation due to the stencil width.

Extra cells exist at the edge of the computational domain and are used for boundary conditions. Extra cells exist on the boundaries of the level where no neighboring patches exist. This can either be the edge of a domain or on a coarse-fine interface.

2.5.4 Indexing grid cells

Vaango uses a straightforward indexing scheme. Each cell on a given level is uniquely identified by an x,y,z coordinate (stored in an `IntVector`). An IntVector is a vector of 3 integers that represent an X,Y,Z coordinate. See Core/Grid/Level.cc/h for more information. The following pseudo code shows how indices across levels are mapped to each other.

```
IntVector
Level::mapCellToCoarser(const IntVector& idx) const
{
  IntVector ratio = idx/d_refinementRatio;

  return ratio;
}

IntVector
Level::mapCellToFiner(const IntVector& idx) const
{
  IntVector r_ratio = grid->getLevel(d_index+1)->d_refinementRatio;
  IntVector fineCell = idx*r_ratio;

  return fineCell;
}
```



3 — PDE Examples

This chapter will describe a set of example problems showing various stages of algorithm complexity and how the VAANGO framework is used to solve the discretized form of the solutions. Emphasis will not be on the most efficient or fast algorithms, but intead will demonstrate straightforward implementations of well known algorithms within the VAANGO Framework. Several examples will be given that show an increasing level of complexity.

All examples described are found in the directory src/CCA/Components/Examples .

3.1 Poisson1

Poisson1 solves Poisson's equation on a grid using Jacobi iteration. Since this is not a time dependent problem and VAANGO is fundamentally designed for time dependent problems, each Jacobi iteration is considered to be a timestep. The timestep specified and computed is a fixed value obtained from the input file and has no bearing on the actual computation.

The following equation is discretized and solved using an iterative method. Each timestep is one iteration. At the end of the timestep, we the residual is computed showing the convergence of the solution and the next iteration is computed until.

The following shows a simplified form of the Poisson1 of the .h and .cc files found in the Examples directory. The argument list for some of the methods are eliminated for readibility purposes. Please refer to the actual source for a complete description of the arguments required for each method.

```
class Poisson1 : public UintahParallelComponent, public SimulationInterface {
public:
  Poisson1(const ProcessorGroup* myworld);
  virtual ~Poisson1();
  virtual void problemSetup(const ProblemSpecP& params, const ProblemSpecP&
      restart_prob_spec, GridP& grid, SimulationStateP&);
  virtual void scheduleInitialize(const LevelP& level,SchedulerP& sched);
  virtual void scheduleComputeStableTimestep(const LevelP& level,SchedulerP&)
      ;
  virtual void scheduleTimeAdvance(const LevelP& level,SchedulerP&);

private:
  void initialize(const ProcessorGroup*, const PatchSubset* patches, const
      MaterialSubset* matls, DataWarehouse* old_dw, DataWarehouse* new_dw);
  void computeStableTimestep(const ProcessorGroup*,const PatchSubset* patches
      , const MaterialSubset* matls,DataWarehouse* old_dw, DataWarehouse*
      new_dw);
   void timeAdvance(const ProcessorGroup,const PatchSubset* patches, const
      MaterialSubset* matls,DataWarehouse* old_dw, DataWarehouse* new_dw);

   SimulationStateP sharedState_;
   double delt_;
   const VarLabel* phi_label;
   const VarLabel* residual_label;
   SimpleMaterial* mymat_;

   Poisson1(const Poisson1&);
   Poisson1& operator=(const Poisson1&);
};
```

The private methods and data shown are the functions that are function pointers referred to in the task descriptions. The VarLabel data type stores the names of the various data that can be referenced uniquely by the data warehouse. The SimulationStateP data type is essentially a global variable that stores information about the materials that are needed by other internal VAANGO framework components. SimpleMaterial is a data type that refers to the material properties.

Within each schedule function, i.e. sheduleInitialize, scheduleComputeStableTimestep, and scheduleTimeAdvance, a task is specified that has a function pointer associated with it. The function pointers point to the actual implementation of the specific task and have a different argument list than the associated schedule method.

The typical task implementation, i.e. timeAdvance() contains the following arguments: ProcessorGroup, PatchSubset, MaterialSubset, and two DataWarehouse objects. The purpose of the ProcessorGroup is to hold various MPI information such as the MPI_Communicator, the rank of the process and the number of processes that are actually being used.

3.1.1 Description of Scheduling Functions

The actual implementation with descriptions are presented following the code snippets.

```
Poisson1::Poisson1(const ProcessorGroup* myworld)
: UintahParallelComponent(myworld)
{
   phi_label = VarLabel::create("phi",
   NCVariable<double>::getTypeDescription());
   residual_label = VarLabel::create("residual",
   sum_vartype::getTypeDescription());

}

Poisson1::~Poisson1()
{
   VarLabel::destroy(phi_label);
   VarLabel::destroy(residual_label);
}
```

Typical constructor and destructor for simple examples where the data label names (phi and residual) are created for data wharehouse storage and retrieval.

```
void Poisson1::problemSetup(const ProblemSpecP& params, const ProblemSpecP&
    restart_prob_spec, GridP& /*grid*/, SimulationStateP& sharedState)
{
   sharedState_ = sharedState;
   ProblemSpecP poisson = params->findBlock("Poisson");
```

```
    poisson->require("delt", delt_);

    mymat_ = scinew SimpleMaterial();

    sharedState->registerSimpleMaterial(mymat_);
  }
```

The problemSetup is based in a xml description of the input file. The input file is parsed and the delt tag is set. The sharedState is assigned and is used to register a material and store it for later use by the Vaango internals.

```
  void Poisson1::scheduleInitialize(const LevelP& level, SchedulerP& sched)
  {
    Task* task = scinew Task("Poisson1::initialize",
    this, &Poisson1::initialize);

    task->computes(phi_label);
    task->computes(residual_label);
    sched->addTask(task, level->eachPatch(), sharedState_->allMaterials());
  }
```

A task is defined which contains a name and a function pointer, i.e. initialize which is described later in Poisson1.cc The task defines two variables that are computed in the initialize function, phi and residual. The task is then added to the scheduler. This task is only computed once at the beginning of the simulation.

```
  void Poisson1::scheduleComputeStableTimestep(const LevelP& level, SchedulerP&
       sched)
  {
    Task* task = scinew Task("Poisson1::computeStableTimestep",
    this, &Poisson1::computeStableTimestep);

    task->requires(Task::NewDW, residual_label);
    task->computes(sharedState_->get_delt_label());
    sched->addTask(task, level->eachPatch(), sharedState_->allMaterials());
  }
```

A task is defined for the computing the stable timestep and uses the function pointer, computeStable-Timestep defined later in Poisson1.cc. This requires data from the New DataWarehouse, and the next timestep size is computed and stored.

```
  void
  Poisson1::scheduleTimeAdvance( const LevelP& level, SchedulerP& sched)
  {
    Task* task = scinew Task("Poisson1::timeAdvance", this, &Poisson1::
        timeAdvance);

    task->requires(Task::OldDW, phi_label, Ghost::AroundNodes, 1);
    task->computes(phi_label);
    task->computes(residual_label);
    sched->addTask(task, level->eachPatch(), sharedState_->allMaterials());
  }
```

The timeAdvance function is the main function that describes the computational algorithm. For simple examples, the entire algorithm is usually defined by one task with a small set of data dependencies. However, for more complicated algorithms, it is best to break the algorithm down into a set of tasks with each task describing its own set of data dependencies.

For this example, a single task is described and the timeAdvance function pointer is specified. Data from the previous timestep (OldDW) is required for the current timestep. For a simple seven (7) point stencil, only one level of ghost cells is required. The algorithm is set up for nodal values, the ghost cells are specified by the the Ghost::AroundNodes syntax. The task computes both the new data values for phi and a residual.

3.1.2 Description of Computational Functions

```
void Poisson1::computeStableTimestep(const ProcessorGroup* pg, const
    PatchSubset* /*patches*/, const MaterialSubset* /*matls*/, DataWarehouse*,
     DataWarehouse* new_dw)
{
  if(pg->myrank() == 0){
    sum_vartype residual;
    new_dw->get(residual, residual_label);
    cerr << "Residual=" << residual << '\n';
  }
  new_dw->put(delt_vartype(delt_), sharedState_->get_delt_label());
}
```

In this particular example, no timestep is actually computed, instead the original timestep specified in the input file is used and stored in the data warehouse (new_dw→put(delt_vartype(delt_), sharedState_→get_delt_label())). The residual computed in the main timeAdvance function is retrieved from the data warehouse and printed out to standard error for only the processor with a rank of 0.

```
void Poisson1::initialize(const ProcessorGroup*, const PatchSubset* patches,
    const MaterialSubset* matls, DataWarehouse* /*old_dw*/, DataWarehouse*
    new_dw)
{
  int matl = 0;
  for(int p=0;p<patches->size();p++){
    const Patch* patch = patches->get(p);
```

The node centered variable (NCVariable<double>phi) has space reserved in the DataWarehouse for the given patch and the given material (int matl = 0;). The phi variable is initialized to 0 for every grid node on the patch.

```
    NCVariable<double> phi;
    new_dw->allocateAndPut(phi, phi_label, matl, patch);
    phi.initialize(0.);
```

The boundary faces on the xminus face of the computational domain are specified and set to a value of 1. All other boundary values are set to a value of 0 as well as the internal nodes via the phi.initialize(0.) construct. Vaango provides helper functions for determining which nodes are on the boundaries. In addition, there are convenient looping constructs such as NodeIterator that alleviate the need to specify triply nested loops for visiting each node in the domain.

```
    if(patch->getBCType(Patch::xminus) != Patch::Neighbor){
      IntVector l,h;
      patch->getFaceNodes(Patch::xminus, 0, l, h);

      for(NodeIterator iter(l,h); !iter.done(); iter++){
        phi[*iter]=1;
      }
    }
    new_dw->put(sum_vartype(-1), residual_label);
  }
}
```

The initial residual value of -1 is stored at the beginning of the simulation.

The main computational algorithm is defined in the timeAdvance function. The overall algorithm is based on a simple Jacobi iteration step.

```
void Poisson1::timeAdvance(const ProcessorGroup*, const PatchSubset* patches,
     const MaterialSubset* matls, DataWarehouse* old_dw, DataWarehouse* new_dw
    )
{
  int matl = 0;
  for(int p=0;p<patches->size();p++){
    const Patch* patch = patches->get(p);
    constNCVariable<double> phi;
```

Data from the previous timestep is retrieved from the data warehouse and copied to the current timestep's phi variable (newphi).

```
    old_dw->get(phi, phi_label, matl, patch, Ghost::AroundNodes, 1);
    NCVariable<double> newphi;

    new_dw->allocateAndPut(newphi, phi_label, matl, patch);
    newphi.copyPatch(phi, newphi.getLowIndex(), newphi.getHighIndex());
```

The indices for the patch are obtained and altered depending on whether or not the patch's internal boundaries are on the coincident with the grid domain. If the patch boundaries are the same as the grid domain, the boundary values are not overwritten since the lower and upper indices are modified to only specify internal nodal grid points.

```
    double residual=0;
    IntVector l = patch->getNodeLowIndex__New();
    IntVector h = patch->getNodeHighIndex__New();

    l += IntVector(patch->getBCType(Patch::xminus) == Patch::Neighbor?0:1,
    patch->getBCType(Patch::yminus) == Patch::Neighbor?0:1,
    patch->getBCType(Patch::zminus) == Patch::Neighbor?0:1);
    h -= IntVector(patch->getBCType(Patch::xplus)  == Patch::Neighbor?0:1,
    patch->getBCType(Patch::yplus)  == Patch::Neighbor?0:1,
    patch->getBCType(Patch::zplus)  == Patch::Neighbor?0:1);
```

The Jacobi iteration step is applied at each internal grid node. The residual is computed based on the old and new values and stored as a reduction variable (sum_vartype) in the data warehouse.

```
    //__________________________________
    //  Stencil
    for(NodeIterator iter(l, h);!iter.done(); iter++){
      IntVector n = *iter;

      newphi[n]=(1./6)*(
      phi[n+IntVector(1,0,0)] + phi[n+IntVector(-1,0,0)] +
      phi[n+IntVector(0,1,0)] + phi[n+IntVector(0,-1,0)] +
      phi[n+IntVector(0,0,1)] + phi[n+IntVector(0,0,-1)]);

      double diff = newphi[n] - phi[n];
      residual += diff * diff;
    }
    new_dw->put(sum_vartype(residual), residual_label);
  }
}
```

3.1.3 Input file

The input file that is used to run this example is given below and is given in SCIRun/src/Packages/Uintah/StandAlone/inputs/Examples/poisson1.ups . Relevant sections of the input file that can be modified are found in the <Time> section, and the <Grid> section, specifically, the number of patches and the grid resolution.

```
<Uintah_specification>
  <Meta>
    <title>Poisson1 test</title>
  </Meta>
  <SimulationComponent>
    <type> poisson1 </type>
  </SimulationComponent>
  <Time>
    <maxTime>        1.0        </maxTime>
    <initTime>       0.0        </initTime>
    <delt_min>       0.00001    </delt_min>
    <delt_max>       1          </delt_max>
    <max_Timesteps> 100          </max_Timesteps>
    <timestep_multiplier>  1  </timestep_multiplier>
  </Time>
  <DataArchiver>
    <filebase>poisson.uda</filebase>
    <outputTimestepInterval>1</outputTimestepInterval>
    <save label = "phi"/>
    <save label = "residual"/>
    <checkpoint cycle = "2" timestepInterval = "1"/>
  </DataArchiver>
  <Poisson>
    <delt>.01</delt>
    <maxresidual>.01</maxresidual>
  </Poisson>
  <Grid>
    <Level>
      <Box label = "1">
        <lower>      [0,0,0]          </lower>
        <upper>      [1.0,1.0,1.0]    </upper>
        <resolution>[50,50,50]       </resolution>
        <patches>    [2,1,1]          </patches>
      </Box>
    </Level>
  </Grid>
</Uintah_specification>
```

Running the Poisson1 Example

To run the poisson1.ups example,

```
  cd ~/SCIRun/dbg/Packages/Uintah/StandAlone/
```

create a symbolic link to the inputs directory:

```
  ln -s ~/SCIRun/src/Packages/Uintah/StandAlone/inputs
```

For a single processor run type the following:

```
  vaango inputs/Examples/poisson1.ups
```

For a two processor run, type the following:

```
  mpirun -np 2 vaango -mpi inputs/Examples/poisson1.ups
```

Changing the number of patches in the poisson1.ups and the resolution, enables you to run a more refined problem on more processors.

ADVICE: For non-AMR problems, it is advised to have at least the same number of patches as processors. You can always have more patches than processors, but you cannot have fewer patches than processors.

3.2 Poisson2

The next example also solves the Poisson's equation but instead of iterating in time, the subscheduler feature iterates within a given timestep, thus solving the problem in one timestep. The use of the subscheduler is important for implementing algorithms which solve non-linear problems which require iterating on a solution for each timestep.

The majority of the schedule and computational functions are similar to the Poisson1 example and are not repeated. Only the revised code is presented with explanations about the new features of Uintah.

```
void Poisson2::scheduleTimeAdvance( const LevelP& level, SchedulerP& sched)
{
  Task* task = scinew Task("timeAdvance", this, &Poisson2::timeAdvance, level
      , sched.get_rep());
  task->hasSubScheduler();
  task->requires(Task::OldDW, phi_label, Ghost::AroundNodes, 1);
  task->computes(phi_label);
  LoadBalancer* lb = sched->getLoadBalancer();
  const PatchSet* perproc_patches = lb->getPerProcessorPatchSet(level);
  sched->addTask(task, perproc_patches, sharedState_->allMaterials());
}
```

Within this function, the task is specified with two additional arguments, the level and the scheduler sched.get_rep() . The task must also set the flag that a subscheduler will be used within the scheduling of the various tasks. Similar code to the Poisson1 example is used to specify what data is required and computed during the actual task execution. In addition, a loadbalancer component is required to query the patch distribution for each level of the grid. The task is then added to the top level scheduler with the requisite information, i.e. patches and materials.

The actual implementation of the timeAdvance function is also different from the Poisson1 example. The code is specified below with text explaining the use of the subscheduler. The new feature of the subscheduler shows the creation of a the iterate task within the subscheduler. This task will perform the actual Jacobi iteration for a given timestep.

```
void Poisson2::timeAdvance(const ProcessorGroup* pg, const PatchSubset*
    patches, const MaterialSubset* matls, DataWarehouse* old_dw, DataWarehouse
    * new_dw, LevelP level, Scheduler* sched)
{
```

The subscheduler is instantiated and initialized.

```
  SchedulerP subsched = sched->createSubScheduler();
  subsched->initialize();
  GridP grid = level->getGrid();
```

An iterate task is created and added to the subscheduler. The typical computes and requires are specified for a 7 point stencil used in Jacobi iteration scheme with one layer of ghost cells. The new task is added to the subscheduler. A residual variable is only computed within the subscheduler and not passed back to the main scheduler. This is in contrast to the phi variable which was specified in scheduleTimeAdvance in the computes, as well as being specified in the computes for the subscheduler. Any variables that are only computed and used in an iterative step of an algorithm do not need to be added to the dependency specification for the top level task.

```
  // Create the tasks
  Task* task = scinew Task("iterate", this, &Poisson2::iterate);
  task->requires(Task::OldDW, phi_label, Ghost::AroundNodes, 1);
  task->computes(phi_label);
  task->computes(residual_label);
  subsched->addTask(task, level->eachPatch(), sharedState_->allMaterials());
```

The subscheduler has its own data wharehouse that is separate from the top level's scheduler's data warehouse. This data warehouse must be initialized and any data from the top level's data warehouse must be passed to the subscheduler's version. This data resides in the data warehouse position NewDW .

```
    // Compile the scheduler
    subsched->advanceDataWarehouse(grid);
    subsched->compile();

    int count = 0;
    double residual;
    subsched->get_dw(1)->transferFrom(old_dw, phi_label, patches, matls);
```

Within each iteration, the following must occur for the subscheduler: the data warehouse's new data must be moved to the OldDW position, since any new values will be stored in NewDW and the old values cannot be overwritten. The OldDW is referred to in the subscheduler via the subsched->get_dw(0) and the NewDW is referred to in the subscheduler via subsched->get_dw(1) . Once the iteration is deemed to have met the tolerance, the data from the subscheduler is transferred to the scheduler's data warehouse.

```
    // Iterate
    do {
      subsched->advanceDataWarehouse(grid);
      subsched->get_dw(0)->setScrubbing(DataWarehouse::ScrubComplete);
      subsched->get_dw(1)->setScrubbing(DataWarehouse::ScrubNonPermanent);
      subsched->execute();

      sum_vartype residual_var;
      subsched->get_dw(1)->get(residual_var, residual_label);
      residual = residual_var;

      if(pg->myrank() == 0)
        cerr << "Iteration " << count++ << ", residual=" << residual << '\n';
    } while(residual > maxresidual_);

    new_dw->transferFrom(subsched->get_dw(1), phi_label, patches, matls);
  }
```

The iteration cycle is identical to Poisson1 's timeAdvance algorithm using Jacobi iteration with a 7 point stencil. Refer to the discussion about the algorithm implementation in the Poisson1 description.

```
  void Poisson2::iterate(const ProcessorGroup*, const PatchSubset* patches,
       const MaterialSubset* matls, DataWarehouse* old_dw, DataWarehouse* new_dw)
  {
    for(int p=0;p<patches->size();p++){
      const Patch* patch = patches->get(p);
      for(int m = 0;m<matls->size();m++){
        int matl = matls->get(m);
        constNCVariable<double> phi;
        old_dw->get(phi, phi_label, matl, patch, Ghost::AroundNodes, 1);
        NCVariable<double> newphi;
        new_dw->allocateAndPut(newphi, phi_label, matl, patch);
        newphi.copyPatch(phi, newphi.getLow(), newphi.getHigh());
        double residual=0;
        IntVector l = patch->getNodeLowIndex__New();
        IntVector h = patch->getNodeHighIndex__New();
        l += IntVector(patch->getBCType(Patch::xminus) == Patch::Neighbor?0:1,
        patch->getBCType(Patch::yminus) == Patch::Neighbor?0:1,
        patch->getBCType(Patch::zminus) == Patch::Neighbor?0:1);
        h -= IntVector(patch->getBCType(Patch::xplus) == Patch::Neighbor?0:1,
        patch->getBCType(Patch::yplus) == Patch::Neighbor?0:1,
        patch->getBCType(Patch::zplus) == Patch::Neighbor?0:1);
        for(NodeIterator iter(l, h);!iter.done(); iter++){
          newphi[*iter]=(1./6)*(
          phi[*iter+IntVector(1,0,0)]+phi[*iter+IntVector(-1,0,0)]+
          phi[*iter+IntVector(0,1,0)]+phi[*iter+IntVector(0,-1,0)]+
          phi[*iter+IntVector(0,0,1)]+phi[*iter+IntVector(0,0,-1)]);
          double diff = newphi[*iter]-phi[*iter];
          residual += diff*diff;
        }
        new_dw->put(sum_vartype(residual), residual_label);
      }
    }
  }
```

The input file src/StandAlone/inputs/Examples/poisson2.ups is very similar to the poisson1.ups file shown above. The only additional tag that is used is the <maxresidual > specifying the tolerance within the iteration performed in the subscheduler.

To run the poisson2 input file execute the following in the dbg build StandAlone directory:

```
  vaango inputs/Examples/poisson2.ups
```

3.3 Burger

In this example, the inviscid Burger's equation is solved in three dimensions:

$$\frac{du}{dt} = -u\frac{du}{dx} \tag{3.1}$$

with the initial conditions:

$$u = \sin(\pi x) + \sin(2\pi y) + \sin(3\pi z) \tag{3.2}$$

using Euler's method to advance in time. The majority of the code is very similar to the Poisson1 example code with the differences shown below.

The initialization of the grid values for the unknown variable, u, is done at every grid node using the NodeIterator construct. The x,y,z values for a given grid node is determined using the function, patch->getNodePosition(n), where n is the nodal index in i,j,k space.

```
  void Burger::initialize(const ProcessorGroup*, const PatchSubset* patches,
      const MaterialSubset* matls, DataWarehouse*, DataWarehouse* new_dw)
  {
    int matl = 0;
    for(int p=0;p<patches->size();p++){
      const Patch* patch = patches->get(p);

      NCVariable<double> u;
      new_dw->allocateAndPut(u, u_label, matl, patch);

      //Initialize
      // u = sin( pi*x ) + sin( pi*2*y ) + sin(pi*3z )
      IntVector l = patch->getNodeLowIndex__New();
      IntVector h = patch->getNodeHighIndex__New();

      for( NodeIterator iter=patch->getNodeIterator__New(); !iter.done(); iter
          ++ ){
        IntVector n = *iter;
        Point p = patch->nodePosition(n);
        u[n] = sin( p.x() * 3.14159265358 ) + sin( p.y() * 2*3.14159265358) +
             sin( p.z() * 3*3.14159265358);
      }
    }
  }
```

The timeAdvance function is quite similar to the Poisson1's timeAdvance routine. The relavant differences are only shown.

```
void Burger::timeAdvance(const ProcessorGroup*, const PatchSubset* patches,
    const MaterialSubset* matls, DataWarehouse* old_dw, DataWarehouse* new_dw)
{
  int matl = 0;
  //Loop for all patches on this processor
  for(int p=0;p<patches->size();p++){
    const Patch* patch = patches->get(p);
    . . . . .
```

The grid spacing and timestep values are stored.

```
    // dt, dx
    Vector dx = patch->getLevel()->dCell();
    delt_vartype dt;
    old_dw->get(dt, sharedState_->get_delt_label());
    . . . . .
```

Refer to the description in Poisson1 about the specification of the NodeIterator limits. The Euler algorithm is applied to solve the ordinary differential equation in time.

```
    //Iterate through all the nodes
    for(NodeIterator iter(l, h);!iter.done(); iter++){
      IntVector n = *iter;
      double dudx = (u[n+IntVector(1,0,0)] - u[n-IntVector(1,0,0)]) /(2.0 *
          dx.x());
      double dudy = (u[n+IntVector(0,1,0)] - u[n-IntVector(0,1,0)]) /(2.0 *
          dx.y());
      double dudz = (u[n+IntVector(0,0,1)] - u[n-IntVector(0,0,1)]) /(2.0 *
          dx.z());
      double du = - u[n] * dt * (dudx + dudy + dudz);
      new_u[n]= u[n] + du;
    }
```

Zero flux Neumann boundary conditions are applied to the node points on each of the grid faces.

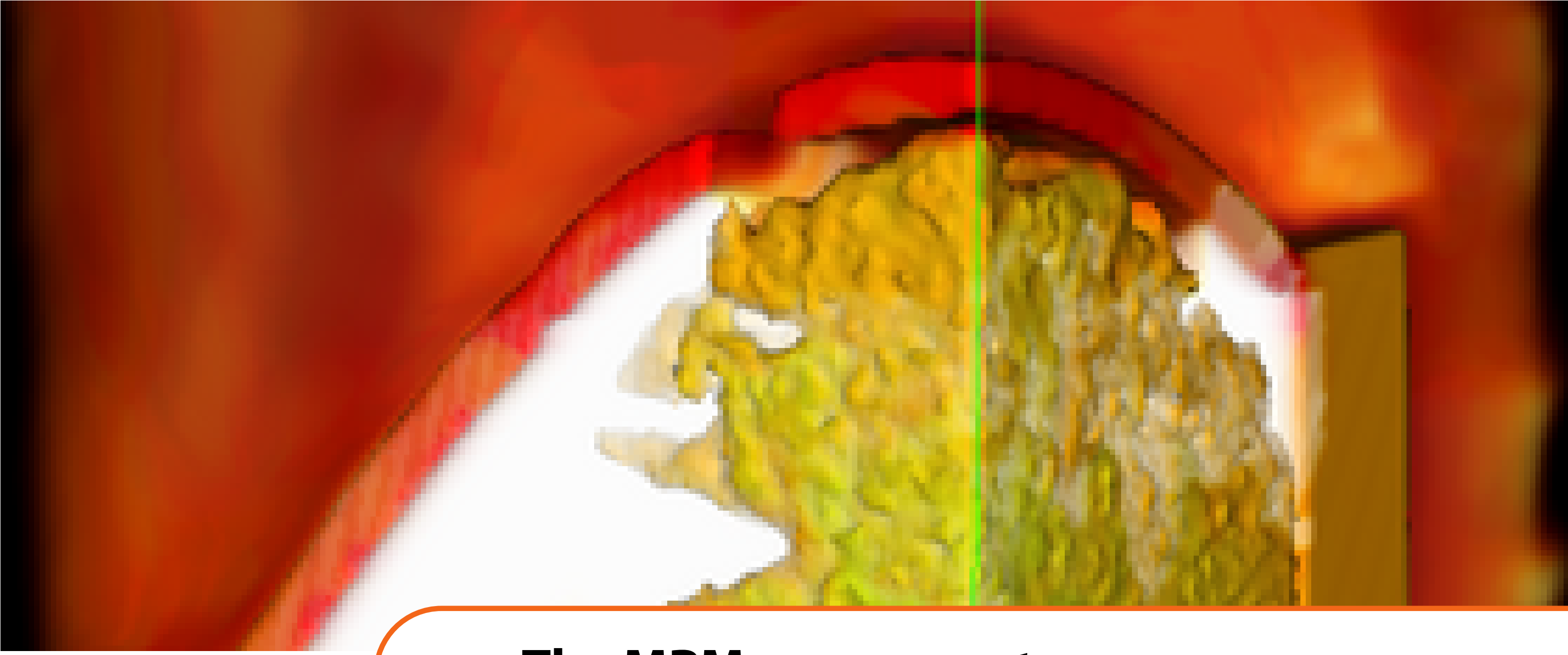
```
    //__________________________________
    // Boundary conditions: Neumann
    // Iterate over the faces encompassing the domain
    vector<Patch::FaceType>::const_iterator iter;
    vector<Patch::FaceType> bf;
    patch->getBoundaryFaces(bf);
    for (iter  = bf.begin(); iter != bf.end(); ++iter){
      Patch::FaceType face = *iter;

      IntVector axes = patch->faceAxes(face);
      int P_dir = axes[0]; // find the principal dir of that face

      IntVector offset(0,0,0);
      if (face == Patch::xminus || face == Patch::yminus || face == Patch::
          zminus){
        offset[P_dir] += 1;
      }
      if (face == Patch::xplus || face == Patch::yplus || face == Patch::
          zplus){
        offset[P_dir] -= 1;
      }

      Patch::FaceIteratorType FN = Patch::FaceNodes;
      for (CellIterator iter = patch->getFaceIterator__New(face,FN);!iter.
          done(); iter++){
```

```
          IntVector n = *iter;
          new_u[n] = new_u[n + offset];
        }
      }
    }
  }
```

The input file for the Burger (burger.ups) problem is very similar to the poisson1.ups with the addition, that the timestep increment used in the timeAdvance is quite small, 1.e-4 for stability reasons.

To run the Burger input file execute the following in the dbg build StandAlone directory:

```
  vaango inputs/Examples/burger.ups
```



4 — The MPM component

The MPM component solves the momentum equations

$$\nabla \cdot \boldsymbol{\sigma} + \rho \mathbf{b} = \rho \dot{\mathbf{v}} \tag{4.1}$$

using an updated Lagrangian formulation. The momentum solve for solid materials is complicated by the fact that the equations need material constitutive models for closure. These material constitutive models vary significantly between materials and contribute a large fraction of the computational cost of a simulation.

In this chapter we discuss the algorithm used in VAANGO to solve the momentum equations using MPM . The implementation follows the approach discussed in the previous chapter.

4.1 The MPM algorithm

The momentum equation is solved using the MPM algorithm while forward Euler time-stepping is use to integrate time derivatives. The pseudocode of the overall algorithm is given below. The main quantities of interest are:

- $t_{\max}$: The maximum time until which the simulation is to run.
- $t, \Delta t$: The current time ($t = t_n$) and the time step.
- $\boldsymbol{h}_g$: The grid spacing vector.
- m_p : The particle mass.
- V_p^n, V_p^{n+1} : The particle volume at $t = t_n$ and $t = t_{n+1}$.
- $\mathbf{x}_p^n, \mathbf{x}_p^{n+1}$: The particle position at $t = t_n$ and $t = t_{n+1}$.
- $\mathbf{u}_p^n, \mathbf{u}_p^{n+1}$: The particle displacement at $t = t_n$ and $t = t_{n+1}$.
- $\mathbf{v}_p^n, \mathbf{v}_p^{n+1}$: The particle velocity at $t = t_n$ and $t = t_{n+1}$.
- $\boldsymbol{\sigma}_p^n, \boldsymbol{\sigma}_p^{n+1}$: The particle Cauchy stress at time $t = t_n$ and $t = t_{n+1}$.
- $\boldsymbol{F}_p^n, \boldsymbol{F}_p^{n+1}$: The particle deformation gradient at time $t = t_n$ and $t = t_{n+1}$.

Algorithm 1 The algorithm

```
1: procedure RUN(inputUPSFile)
2:     $t_{\max}$, $\boldsymbol{h}_g$, xmlProblemSpec, grid, globalState ← READINPUTUPSFILE(inputUPSFile)    ▷Parse
           ↪ the input XML file (¡filename¿.ups), create the background grid, and
           ↪ set up a SIMULATIONSTATE.
```

```
3:     mpmFlags, prescribedDefGrad, particleBC, contactModel, constitutiveModel,
         ↪ defGradComputer, damageModel ← PROBLEMSETUP(xmlProblemSpec, grid,
         ↪ globalState)          ▷Set up flags, the constitutive model, and the deformation gradient
         ↪ algorithm based on data in input file.
4:     $t \leftarrow 0, n \leftarrow 0$
5:     $\mathbf{x}_p^n, \mathbf{u}_p^n, m_p, V_p^n, \mathbf{v}_p^n, \boldsymbol{\sigma}_p^n, \boldsymbol{F}_p^n$ ← INITIALIZE(xmlProblemSpec)     ▷Find the grid size and initialize
         ↪ particle variables based on geometry and other information in the input file.
6:     isSuccess ← FALSE
7:     repeat
8:         $\Delta t$ ← COMPUTESTABLETIMESTEP($\boldsymbol{h}_g, \mathbf{v}_p$)     ▷Find a stable time increment based on
             ↪ grid size and velocity
9:         $t \leftarrow t + \Delta t, n \leftarrow n + 1$     ▷Update the time
10:        isSuccess, $\mathbf{x}_p^{n+1}, \mathbf{u}_p^{n+1}, v_p^{n+1}, \mathbf{v}_p^{n+1}, \boldsymbol{\sigma}_p^{n+1}, \boldsymbol{F}_p^{n+1}$ ← TIMEADVANCE($\boldsymbol{h}_g, \mathbf{x}_p^n, \mathbf{u}_p^n, m_p, V_p^n$,
             ↪ $\mathbf{v}_p^n, \boldsymbol{\sigma}_p^n, \boldsymbol{F}_p^n$)     ▷Compute updated quantities
11:        OUTPUTDATA($\mathbf{x}_p^{n+1}, \mathbf{u}_p^{n+1}, V_p^{n+1}, \mathbf{v}_p^{n+1}, \boldsymbol{\sigma}_p^{n+1}, \boldsymbol{F}_p^{n+1}$)     ▷Save the solution
12:        $n \leftarrow n + 1$
13:    until $t \geq t_{\max}$
14:    return isSuccess
15: end procedure
```

4.2 Reading the input file

The process used to read the input file is identical to that discussed earlier in this manual.

4.3 Problem setup

The overall structure of the problem setup code is given below. Details can be found in the code.

Algorithm 2 Problem setup

```
Require: xmlProblemSpec, grid, globalState
1: procedure PROBLEMSETUP(xmlProblemSpec, grid, globalState)
2:     flags ← READMPMFLAGS(xmlProblemSpec)     ▷Read the option flags that determine
         ↪ the details of the MPM algorithm to be used in the simulation.
3:     if flags.prescribeDeformation = TRUE then
4:         prescribedDefGrad ← READPRESCRIBEDDEFORMATIONS(flags.prescribedFileName)
5:     end if
6:     particleBC ← CREATEMPMPHYSICALBC(xmlProblemSpec, grid, flags)     ▷Create the model
         ↪ used to apply pressures and forces directly to particles.
7:     contactModel ← CREATECONTACTMODEL(xmlProblemSpec, grid, flags, globalState) ▷Create
         ↪ the contact algorithm model used to compute interactions between objects.
8:     constitutiveModel ← CREATECONSTITUTIVEMODELS(xmlProblemSpec, grid, flags,
         ↪ globalState)     ▷Create the constitutive models that are needed
         ↪ for the simulation.
9:     defGradComputer ← CREATEDEFORMATIONGRADIENTCOMPUTER(flags, globalState) ▷Create
         ↪ the model that will be used to compute velocity and
         ↪ deformation gradients.
10:    if flags.doBasicDamage = TRUE then
11:        damageModel ← CREATEBASICDAMAGEMODEL(flags, globalState)
12:    end if
13:    return flags, prescribedDefGrad, particleBC, contactModel, constitutiveModel,
        ↪ damageModel, defGradComputer
14: end procedure
```

4.4 Initialization

An outline of the initialization process is described below. Specific details have been discussed in earlier reports. The new quantities introduced in this section are

- n_p : The number of particles used to discretize a body.
- $\mathbf{b}_p^n, \mathbf{b}_p^{n+1}$: The particle body force acceleration at $t = t_n$ and $t = t_{n+1}$.
- D_p^n, D_p^{n+1} : The particle damage parameter at $t = t_n$ and $t = t_{n+1}$.
- $\mathbf{f}_p^{\text{ext},n}, \mathbf{f}_p^{\text{ext},n+1}$: The particle external force at $t = t_n$ and $t = t_{n+1}$.

Algorithm 3 Initialization

```
Require: xmlProblemSpec, defGradComputer, constitutiveModel, damageModel, particleBC,
        ↪ mpmFlags materialList,
1: procedure INITIALIZE
2:     for matl in materialList do
3:         $n_p$[matl], $\mathbf{x}_p^o$[matl], $\mathbf{u}_p^o$[matl], $m_p$[matl], $V_p^o$[matl], $\mathbf{v}_p^o$[matl], $\mathbf{b}_p^o$[matl],
               ↪ $\mathbf{f}_p^{\text{ext},o}$[matl] ← matl.CREATEPARTICLES()
4:         $\boldsymbol{F}_p^o$[matl] ← defGradComputer.INITIALIZE(matl)
5:         $\boldsymbol{\sigma}_p^o$[matl] ← constitutiveModel.INITIALIZE(matl)
6:         $D_p^o$[matl] ← damageModel.INITIALIZE(matl)
7:     end for
8:     if mpmFlags.initializeStressWithBodyForce = TRUE then
9:         $\mathbf{b}_p^o$ ← INITIALIZEBODYFORCE()
10:        $\boldsymbol{\sigma}_p^o, \boldsymbol{F}_p^o$ ← INITIALIZESTRESSANDDEFGRADFROMBODYFORCE()
11:    end if
12:    if mpmFlags.applyParticleBCs = TRUE then
13:        $\mathbf{f}_p^{\text{ext},o}$ ← particleBC.INITIALIZEPRESSUREBCS()
14:    end if
15:    return $n_p, \mathbf{x}_p^o, \mathbf{u}_p^o, m_p, V_p^o, \mathbf{v}_p^o, \mathbf{b}_p^o, \mathbf{f}_p^{\text{ext},o}, \boldsymbol{F}_p^o, \boldsymbol{\sigma}_p^o, D_p^o$
16: end procedure
```

4.5 Time advance

The operations performed during a timestep are shown in the pseudocode below.

Algorithm 4 The MPM time advance algorithm

```
1: procedure TIMEADVANCE($\boldsymbol{h}_g, x_p^n, u_p^n, m_p, V_p^n, \mathbf{v}_p^n, \mathbf{f}_p^{\text{ext},n}, \boldsymbol{d}_p^n$)
2:     $\mathbf{b}_p^n$ ← COMPUTEPARTICLEBODYFORCE()                    ▷Compute the body force term
3:     $\mathbf{f}_p^{\text{ext},n+1}$ ← APPLYEXTERNALLOADS()            ▷Apply external loads to the particles
4:     $m_g, V_g, \mathbf{v}_g, \mathbf{b}_g, \mathbf{f}_g^{\text{ext}}$ ← INTERPOLATEPARTICLESTOGRID()   ▷Interpolate particle data to the grid
5:     EXCHANGEMOMENTUMINTERPOLATED()          ▷Exchange momentum between bodies on grid.
           ↪ Not discussed in this report.
6:     $\mathbf{f}_g^{\text{int}}, \boldsymbol{\sigma}_g, \mathbf{v}_g$ ← COMPUTEINTERNALFORCE()   ▷Compute the internal force at the grid nodes
7:     $\mathbf{v}_g^\star, \mathbf{a}_g$ ← COMPUTEANDINTEGRATEACCELERATION()          ▷Compute the grid velocity
           ↪ and grid acceleration
8:     EXCHANGEMOMENTUMINTEGRATED()            ▷Exchange momentum between bodies on grid
           ↪ using integrated values. Not discussed in this report.
```

```
 9:     $\mathbf{v}^\star_g, \mathbf{a}_g \leftarrow$ setGridBoundaryConditions()          ▷Update the grid velocity and grid
            ↪ acceleration using the BCs
10:     $\boldsymbol{l}^n_p, \boldsymbol{F}^{n+1}_p, V^{n+1}_p \leftarrow$ computeDeformationGradient()      ▷Compute the velocity gradient
            ↪ and the deformation gradient
11:     $\boldsymbol{\sigma}^{n+1}_p, \boldsymbol{\eta}^{n+1}_p \leftarrow$ computeStressTensor()          ▷Compute the updated stress and
            ↪ internal variables (if any)
12:     $\boldsymbol{\sigma}^{n+1}_p, \boldsymbol{\eta}^{n+1}_p, \chi^{n+1}_p, D^{n+1}_p \leftarrow$ computeBasicDamage()     ▷Compute the damage parameter
            ↪ and update the stress and internal variables
13:     $\chi^{n+1}_p, D^{n+1}_p \leftarrow$ updateErosionParameter()     ▷Update the indicator variable that is used
            ↪ to delete particles at the end of a time step
14:     $V^{n+1}_p, \mathbf{u}^{n+1}_p, \mathbf{v}^{n+1}_p, \mathbf{x}^{n+1}_p, m_p, \boldsymbol{h}^{n+1}_p \leftarrow$ interpolateToParticlesAndUpdate()     ▷Update the
            ↪ particle variables after interpolating grid quantities to particles
15: end procedure
```

The algorithms used for the above operations are discussed next.

4.5.1 Computing the body force

The body force consists of a gravitational term and, optionally, centrifugal and coriolis terms that are needed for simulations inside a rotating frame such as a centrifuge.

Algorithm 5 Computing the body force on particles

```
Require: $\mathbf{x}^n_p, \mathbf{v}^n_p$, materialList, particleList, mpmFlags
 1: procedure computeParticleBodyForce
 2:     for matl in materialList do
 3:         if mpmFlags.rotatingCoordSystem = TRUE then
 4:             $\mathbf{g} \leftarrow$ mpmFlags.gravityAcceleration
 5:             $\mathbf{b}^n_p$[matl] $\leftarrow \mathbf{g}$
 6:         else
 7:             for part in particleList do
 8:                 $\mathbf{g} \leftarrow$ mpmFlags.gravityAcceleration
 9:                 $\mathbf{x}_{rc} \leftarrow$ mpmFlags.coordRotationCenter
10:                 $\mathbf{z}_r \leftarrow$ mpmFlags.coordRotationAxis
11:                 $w \leftarrow$ mpmFlags.coordRotationSpeed
12:                 $\boldsymbol{\omega} \leftarrow w\mathbf{z}_r$                                ▷Compute angular velocity vector
13:                 $\mathbf{a}_{\text{corolis}} \leftarrow 2\boldsymbol{\omega} \times \mathbf{v}^n_p$[matl, part]                 ▷Compute Coriolis acceleration
14:                 $\boldsymbol{r} \leftarrow \mathbf{x}^n_p$[matl, part] $- \mathbf{x}_{rc}$
15:                 $\mathbf{a}_{\text{centrifugal}} \leftarrow \boldsymbol{\omega} \times \boldsymbol{\omega} \times \boldsymbol{r}$     ▷Compute the centrifugal body force acceleration
16:                 $\mathbf{b}^n_p$[matl,part] $\leftarrow \mathbf{g} - \mathbf{a}_{\text{centrifugal}} - \mathbf{a}_{\text{coriolis}}$     ▷Compute the body force acceleration
17:             end for
18:         end if
19:     end for
20:     return $\mathbf{b}^n_p$
21: end procedure
```

4.5.2 Applying external loads

Note that the updated deformation gradient has not been computed yet at this stage and the particle force is applied based on the deformation gradient at the beginning of the timestep. The new quantities introduced in this section are:

- $\boldsymbol{h}^n_p$: The particle size matrix at time $t = t_n$.

Algorithm 6 Applying external loads to particles

```
Require: $t_{n+1}$, $\mathbf{x}_p^n$, $\boldsymbol{h}_p^n$, $\mathbf{u}_p^n$, $\mathbf{f}_p^{\text{ext},n}$, $\boldsymbol{F}_p^n$, materialList, particleList, mpmFlags, particleBC
 1: procedure APPLYEXTERNALLOADS
 2:     $f_p \leftarrow 0$
 3:     if mpmFlags.useLoadCurves = TRUE then
 4:         $f_p \leftarrow$ particleBC.COMPUTEFORCEPERPARTICLE($t^{n+1}$)    ▷Compute the force per particle
            ↪ due to the applied pressure
 5:     end if
 6:     for matl in materialList do
 7:         if mpmFlags.useLoadCurves = TRUE then
 8:             for part in particleList do
 9:                 $\mathbf{f}_p^{\text{ext},n+1}$[matl,part] $\leftarrow$ particleBC.GETFORCEVECTOR($t_{n+1}$, $\mathbf{x}_p^n$, $\boldsymbol{h}_p^n$, $\mathbf{u}_p^n$,
                        ↪ $f_p$, $\boldsymbol{F}_p^n$)    ▷Compute the applied force vector at each particle
10:             end for
11:         else
12:             $\mathbf{f}_p^{\text{ext},n+1}$[matl] $\leftarrow$ $\mathbf{f}_p^{\text{ext},n}$[matl]
13:         end if
14:     end for
15:     return $\mathbf{f}_p^{\text{ext},n+1}$
16: end procedure
```

4.5.3 Interpolating particles to grid

The grid quantities computed during this procedure and not stored for the next timestep except for the purpose of visualization. The new quantities introduced in this section are

- m_g : The mass at a grid node.
- V_g : The volume at a grid node.
- $\mathbf{v}_g$: The velocity at a grid node.
- $\mathbf{f}_g^{\text{ext}}$: The external force at a grid node.
- $\mathbf{b}_g$: The body force at a grid node.

Algorithm 7 Interpolating particle data to background grid

```
Require: $m_p$, $V_p^n$, $\mathbf{x}_p^n$, $\boldsymbol{h}_p^n$, $\mathbf{b}_p^n$, $\mathbf{f}_p^{\text{ext},n+1}$, $\boldsymbol{F}_p^n$, materialList, particleList, gridNodeList mpmFlags, particleBC
 1: procedure INTERPOLATEPARTICLESTOGRID
 2:     interpolator $\leftarrow$ CREATEINTERPOLATOR(mpmFlags)    ▷Create the interpolator
            ↪ and find number of grid nodes that can affect a particle
 3:     for matl in materialList do
 4:         for part in particleList do
 5:             $n_{gp}$, $S_{gp}$ $\leftarrow$ interpolator.FINDCELLSANDWEIGHTS($\mathbf{x}_p^n$, $\boldsymbol{h}_p^n$, $\boldsymbol{F}_p^n$)    ▷Find the node
                    ↪ indices of the cells affecting the particle and the interpolation weights
 6:             $\mathbf{p}_p \leftarrow m_p$[matl][part] $\mathbf{v}_p^n$[matl][part]    ▷Compute particle momentum
 7:             for node in $n_{gp}$ do
 8:                 $m_g$[matl][node] $\leftarrow$ $m_g$[matl][node] + $m_p$[matl][part] $S_{gp}$[node]
 9:                 $V_g$[matl][node] $\leftarrow$ $V_g$[matl][node] + $V_p^n$[matl][part] $S_{gp}$[node]
10:                 $\mathbf{v}_g$[matl][node] $\leftarrow$ $\mathbf{v}_g$[matl][node] + $\mathbf{p}_p$ $S_{gp}$[node]
11:                 $\mathbf{f}_g^{\text{ext}}$[matl][node] $\leftarrow$ $\mathbf{f}_g^{\text{ext}}$[matl][node] + $\mathbf{f}_p^{\text{ext},n+1}$[matl][part] $S_{gp}$[node]
12:                 $\mathbf{b}_g$[node] $\leftarrow$ $\mathbf{b}_g$[node] + $m_p$[matl][part] $\mathbf{b}_p^n$[matl][part] $S_{gp}$[node]
13:             end for
14:         end for
```

```
15:         for node in gridNodeList do
16:             $\mathbf{v}_g$[matl][node] ← $\mathbf{v}_g$[matl][node]/$m_g$[matl][node]
17:         end for
18:         $\mathbf{v}_g$[matl] ← APPLYSYMMETRYVELOCITYBC($\mathbf{v}_g$[matl])        ▷Apply any symmetry
                ↪ velocity BCs that may be applicable
19:     end for
20:     return $m_g$, $V_g$, $\mathbf{v}_g$, $\mathbf{b}_g$, $\mathbf{f}_g^{\text{ext}}$
21: end procedure
```

4.5.4 Exchanging momentum using interpolated grid values

The exchange of momentum is carried out using a contact model. Details can be found in the Uintah Developers Manual.

4.5.5 Computing the internal force

This procedure computes the internal force at the grid nodes. The new quantities introduced in this section are

- n_{gp} : The number of grid nodes that are used to interpolate from particle to grid.
- S_{gp} : The nodal interpolation function evaluated at a particle
- $\boldsymbol{G}_{gp}$: The gradient of the nodal interpolation function evaluated at a particle
- $\boldsymbol{\sigma}_v$: A volume weighted grid node stress.
- $\mathbf{f}_g^{\text{int}}$: The internal force at a grid node.

Algorithm 8 Computing the internal force

```
Require: $\boldsymbol{h}_g$, $V_g$, $V_p^n$, $\mathbf{x}_p^n$, $\boldsymbol{h}_p^n$, $\boldsymbol{\sigma}_p^n$, $\boldsymbol{F}_p^n$, materialList, particleList, gridNodeList mpmFlags
1: procedure COMPUTEINTERNALFORCE
2:     interpolator ← CREATEINTERPOLATOR(mpmFlags)        ▷Create the interpolator and
           ↪ find number of grid nodes that can affect a particle
3:     for matl in materialList do
4:         for part in particleList do
5:             $n_{gp}$, $S_{gp}$, $\mathbf{G}_{gp}$ ←
                   ↪ interpolator.FINDCELLSANDWEIGHTSANDSHAPEDERVATIVES($\mathbf{x}_p^n$, $\boldsymbol{h}_p^n$, $\boldsymbol{F}_p^n$)
                   ↪        ▷Find the node indices of the cells affecting the particle and
                   ↪ the interpolation weights and gradients
6:             $\boldsymbol{\sigma}_v$ ← $V_p$[matl][part] $\boldsymbol{\sigma}_p^n$[matl][part]
7:             for node in $n_{gp}$ do
8:                 $\mathbf{f}_g^{\text{int}}$[matl][node] ← $\mathbf{f}_g^{\text{int}}$[matl][node] - ($\mathbf{G}_{gp}$[node]/$\boldsymbol{h}_g$) ·$\boldsymbol{\sigma}_p^n$[matl][part] $V_p^n$[part]
9:                 $\boldsymbol{\sigma}_g$[matl][node] ← $\boldsymbol{\sigma}_g$[matl][node] + $\boldsymbol{\sigma}_v$ $S_{gp}$[node]
10:            end for
11:        end for
12:        for node in gridNodeList do
13:            $\boldsymbol{\sigma}_g$[matl][node] ← $\boldsymbol{\sigma}_g$[matl][node]/$V_g$[matl][node]
14:        end for
15:        $\mathbf{v}_g$[matl] ← APPLYSYMMETRYTRACTIONBC()        ▷Apply any symmetry tractions BCs
                   ↪ that may be applicable
16:    end for
17:    return $\mathbf{f}_g^{\text{int}}$, $\boldsymbol{\sigma}_g$, $\mathbf{v}_g$
18: end procedure
```

4.5.6 Computing and integrating the acceleration

This procedure computes the accelerations at the grid nodes and integrates the grid accelerations using forward Euler to compute grid velocities. The new quantities introduced in this section are

- $\mathbf{a}_g$: The grid accelerations.
- $\mathbf{v}_g^\star$: The integrated grid velocities.

Algorithm 9 Computing and integrating the acceleration

```
Require: $\Delta t$, $m_g$, $\mathbf{f}_g^{\text{int}}$, $\mathbf{f}_g^{\text{ext}}$, $\mathbf{b}_g$, $\mathbf{v}_g$, materialList, gridNodeList, mpmFlags
1: procedure COMPUTEANDINTEGRATEACCELERATION
2:     for matl in materialList do
3:         for node in gridNodeList do
4:             $\mathbf{a}_g$[matl][node] ← ($\mathbf{f}_g^{\text{int}}$[matl][node] + $\mathbf{f}_g^{\text{ext}}$[matl][node] + $\mathbf{b}_g$[matl][node])/$m_g$[matl][node]
5:             $\mathbf{v}_g^\star$ ← $\mathbf{v}_g$[matl][node] + $\mathbf{a}_g$[matl][node] $* \Delta t$
6:         end for
7:     end for
8:     return $\mathbf{v}_g^\star$, $\mathbf{a}_g$
9: end procedure
```

4.5.7 Exchanging momentum using integrated grid values

The exchange of momentum is carried out using a contact model. Details can be found in the Uintah Developers Manual.

4.5.8 Setting grid boundary conditions

Algorithm 10 Setting grid boundary conditions

```
Require: $\Delta t$, $\mathbf{a}_g$, $\mathbf{v}_g^\star$, $\mathbf{v}_g$, materialList, gridNodeList, mpmFlags
1: procedure SETGRIDBOUNDARYCONDITIONS
2:     for matl in materialList do
3:         $\mathbf{v}_g^\star$[matl] ← APPLYSYMMETRYVELOCITYBC($\mathbf{v}_g^\star$[matl])
4:         for node in gridNodeList do
5:             $\mathbf{a}_g$[matl][node] ← ($\mathbf{v}_g^\star$[matl][node] - $\mathbf{v}_g$[matl][node]) /$\Delta t$
6:         end for
7:     end for
8:     return $\mathbf{v}_g^\star$, $\mathbf{a}_g$
9: end procedure
```

4.5.9 Computing the deformation gradient

The velocity gradient is computed using the integrated grid velocities and then used to compute the deformation gradient. The new quantities introduced in this section are

- $\Delta \boldsymbol{F}_p^n$: The increment of the particle deformation gradient.
- $\boldsymbol{l}_p^{n+1}$: The particle velocity gradient.
- ρ_0 : The initial mass density of the material.

Algorithm 11 Computing the velocity gradient and deformation gradient

```
Require: $\Delta t$, $\mathbf{x}_p^n$, $m_p$, $V_p^n$, $\boldsymbol{h}_p^n$, $\mathbf{v}_p^n$, $\boldsymbol{l}_p^n$, $\boldsymbol{F}_p^n$, $\boldsymbol{h}_g$, $\mathbf{v}_g$, $\mathbf{v}_g^\star$, $\rho_0$ materialList, gridNodeList, mpmFlags, velGradComputer
1: procedure COMPUTEDEFORMATIONGRADIENT
2:     interpolator ← CREATEINTERPOLATOR(mpmFlags)
```

```
3:      for matl in materialList do
4:          for part in particleList do
5:              $\boldsymbol{l}_p^{n+1}$[matl,part] ← velGradComputer.computeVelGrad(interpolator, $\boldsymbol{h}_g$, $\mathbf{x}_p^n$[matl,part],
                    ↪ $\boldsymbol{h}_p^n$[matl,part], $\boldsymbol{F}_p^n$[matl,part], $\mathbf{v}_g^\star$[matl])    ▷Compute the velocity gradient
6:              $\boldsymbol{F}_p^{n+1}$[matl,part], $\Delta\boldsymbol{F}_p^{n+1}$ ← computeDeformationGradientFromVelocity($\boldsymbol{l}_p^n$[matl,part],
                    ↪ $\boldsymbol{l}_p^{n+1}$[matl,part], $\boldsymbol{F}_p^n$[matl,part])    ▷Compute the deformation gradient
7:              $V_p^{n+1}$[matl,part] ← $m_p$[matl,part]/$\rho_0 * \det(\boldsymbol{F}_p^{n+1}$[matl,part])
8:          end for
9:      end for
10:     return $\boldsymbol{l}_p^{n+1}$, $\boldsymbol{F}_p^{n+1}$, $V_p^{n+1}$
11: end procedure
```

Algorithm 12 Computing the deformation gradient using the velocity gradient

```
Require: $\Delta t$, $\boldsymbol{l}_p^{n+1}$, $\boldsymbol{F}_p^n$, mpmFlags
1:  procedure computeDeformationGradientFromVelocity
2:      if mpmFlags.defGradAlgorithm = "first_order" then
3:          $\boldsymbol{F}_p^{n+1}$, $\Delta\boldsymbol{F}_p^{n+1}$ ← seriesUpdateConstantVelGrad(numTerms = 1, $\Delta t$, $\boldsymbol{l}_p^{n+1}$, $\boldsymbol{F}_p^n$)
4:      else if mpmFlags.defGradAlgorithm = "subcycle" then
5:          $\boldsymbol{F}_p^{n+1}$, $\Delta\boldsymbol{F}_p^{n+1}$ ← subcycleUpdateConstantVelGrad($\Delta t$, $\boldsymbol{l}_p^{n+1}$, $\boldsymbol{F}_p^n$)
6:      else if mpmFlags.defGradAlgorithm = "taylor_series" then
7:          $\boldsymbol{F}_p^{n+1}$, $\Delta\boldsymbol{F}_p^{n+1}$ ← seriesUpdateConstantVelGrad(numTerms = mpmFlags.numTaylorSeriesTerms,
    $\Delta t$, $\boldsymbol{l}_p^{n+1}$, $\boldsymbol{F}_p^n$)
8:      else
9:          $\boldsymbol{F}_p^{n+1}$, $\Delta\boldsymbol{F}_p^{n+1}$ ← cayleyUpdateConstantVelGrad($\Delta t$, $\boldsymbol{l}_p^{n+1}$, $\boldsymbol{F}_p^n$)
10:     end if
11:     return $\boldsymbol{F}_p^{n+1}$, $\Delta\boldsymbol{F}_p^{n+1}$
12: end procedure
```

4.5.10 Computing the stress tensor

The stress tensor is compute by individual constitutive models. Details of the Arena partially saturated model are given later. The new quantities introduced in this section are

- $\boldsymbol{\eta}_p^n$, $\boldsymbol{\eta}_p^{n+1}$: The internal variables needed by the constitutive model.

Algorithm 13 Computing the stress tensor

```
Require: $\Delta t$, $\mathbf{x}_p^n$, $m_p$, $V_p^{n+1}$, $\boldsymbol{h}_p^n$, $\boldsymbol{l}_p^{n+1}$, $\boldsymbol{F}_p^{n+1}$, $\boldsymbol{\sigma}_p^n$, $\boldsymbol{\eta}_p^n$, $\rho_0$, materialList, mpmFlags, constitutiveModel
1:  procedure computeStressTensor
2:      for matl in materialList do
3:          $\boldsymbol{\sigma}^{n+1}$, $\boldsymbol{\eta}_p^{n+1}$ ← constitutiveModel[matl].computeStressTensor($\Delta t$, $\mathbf{x}_p^n$, $m_p$, $V_p^{n+1}$, $\boldsymbol{h}_p^n$,
                    ↪ $\boldsymbol{l}_p^{n+1}$, $\boldsymbol{F}_p^{n+1}$, $\boldsymbol{\sigma}_p^n$, $\boldsymbol{\eta}_p^n$, $\rho_0$, mpmFlags)    ▷Update the stress and any
                    ↪ internal variables needed by the constitutive model
4:      end for
5:      return $\boldsymbol{\sigma}_p^{n+1}$, $\boldsymbol{\eta}_p^{n+1}$
6:  end procedure
```

4.5.11 Computing the basic damage parameter

The damage parameter is updated and the particle stress is modified in this procedure. The new quantities introduced in this section are

- $\varepsilon_p^{f,n}$, $\varepsilon_p^{f,n+1}$: The particle strain to failure at $t = T_n$ and $t = T_{n+1}$.
- χ_p^n, χ_p^{n+1} : An indicator function that identifies whether a particle has failed completely.
- $t_p^{\chi,n}$, $t_p^{\chi,n+1}$: The time to failure of a particle.
- D_p^n, D_p^{n+1} : A particle damage parameter that can be used to modify the stress.

Algorithm 14 Computing the damage parameter

```
Require: $t^{n+1}$, $V_p^{n+1}$, $\mathbf{F}_p^{n+1}$, $\boldsymbol{\sigma}_p^{n+1}$, $D_p^n$, $\varepsilon_p^{f,n}$, $\chi_p^n$, $t_p^{\chi,n}$, materialList, mpmFlags
 1: procedure COMPUTEDAMAGE
 2:     for matl in materialList do
 3:         for part in particleList do
 4:             if brittleDamage = TRUE then
 5:                 $\boldsymbol{\sigma}_p^{n+1}$, $\varepsilon_p^{f,n+1}$, $\chi_p^{n+1}$, $t_p^{\chi,n+1}$, $D_p^{n+1}$ ← UPDATEDAMAGEANDMODIFYSTRESS($V_p^{n+1}$, $\mathbf{F}_p^{n+1}$,
                   ↪ $\boldsymbol{\sigma}_p^{n+1}$, $D_p^n$, $\varepsilon_p^{f,n}$, $\chi_p^n$, $t_p^{\chi,n}$)      ▷ Update the damage parameters and stress
 6:             else
 7:                 $\boldsymbol{\sigma}_p^{n+1}$, $\varepsilon_p^{f,n+1}$, $\chi_p^{n+1}$, $t_p^{\chi,n+1}$ ← UPDATEFAILEDPARTICLESANDMODIFYSTRESS($V_p^{n+1}$, $\mathbf{F}_p^{n+1}$,
                   ↪ $\boldsymbol{\sigma}_p^{n+1}$, $\varepsilon_p^{f,n}$, $\chi_p^n$, $t_p^{\chi,n}$, $t^{n+1}$)      ▷ Update the failed particles and stress
 8:             end if
 9:         end for
10:     end for
11:     return $\boldsymbol{\sigma}_p^{n+1}$, $\varepsilon_p^{f,n+1}$, $\chi_p^{n+1}$, $t_p^{\chi,n+1}$, $D_p^{n+1}$
12: end procedure
```

4.5.12 Updating the particle erosion parameter

The particle failure indicator function is updated in this procedure and used later for particle deletion if needed.

Algorithm 15 Updating the particle erosion parameter

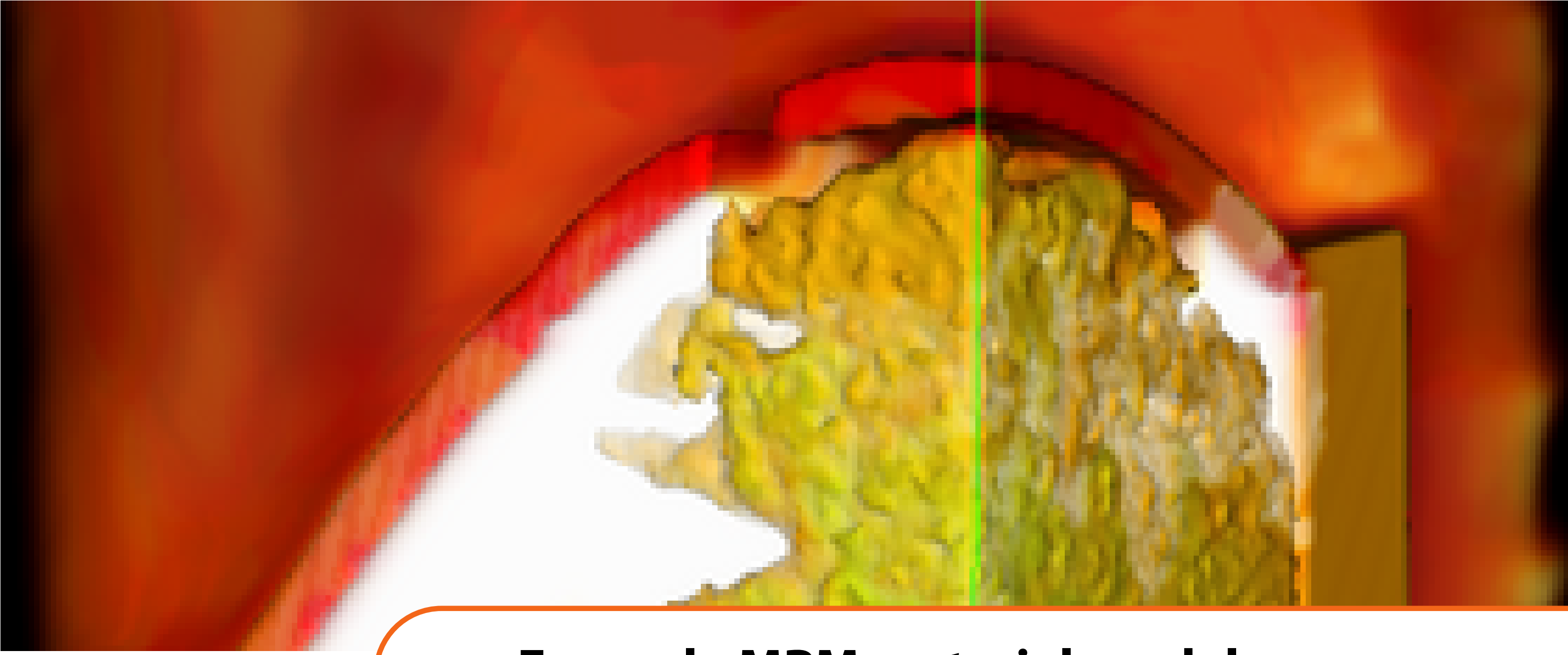
```
Require: $D_p^n$, $\chi_p^n$ materialList, mpmFlags, constitutiveModel
 1: procedure UPDATEEROSIONPARAMETER
 2:     for matl in materialList do
 3:         for part in particleList do
 4:             if matl.doBasicDamage = TRUE then
 5:                 $\chi_p^{n+1}$ ← damageModel.GETLOCALIZATIONPARAMETER()      ▷ Just get the indicator
                   ↪ parameter for particles that will be eroded.
 6:             else
 7:                 $\chi_p^{n+1}$, $D_p^{n+1}$ ← constitutiveModel[matl].GETDAMAGEPARAMETER($\chi_p^n$, $D_p^n$)
                   ↪      ▷ Update the damage parameter in the constitutive model.
 8:             end if
 9:         end for
10:     end for
11:     return $\chi_p^{n+1}$, $D_p^{n+1}$
12: end procedure
```

4.5.13 Interpolating back to the particles and update

This is the final step at which the particle velocities and positions are updated and the grid is reset. Particle that are to be removed are dealt with in a subsequent `relocation` step.

Algorithm 16 Interpolating back to the particles and position update

```
Require: $\Delta t$, $\mathbf{a}_g$, $\mathbf{v}_g^\star$, $\mathbf{x}_p^n$, $\mathbf{v}_p^n$, $\mathbf{u}_p^n$, $\boldsymbol{h}_p^n$, $\chi_p^{n+1}$, $\boldsymbol{F}_p^{n+1}$, $V_p^{n+1}$, materialList, particleList, gridNodeList,
    mpmFlags
1: procedure INTERPOLATETOPARTICLESANDUPDATE
2:     interpolator ← CREATEINTERPOLATOR(mpmFlags)
3:     for matl in materialList do
4:         $\boldsymbol{h}_p^{n+1} \leftarrow \boldsymbol{h}_p^n$
5:         for part in particleList do
6:             $n_{gp}, S_{gp} \leftarrow$ interpolator.FINDCELLSANDWEIGHTS($\mathbf{x}_p^n, \boldsymbol{h}_p^{n+1}, \boldsymbol{F}_p^{n+1}$)
7:             $\mathbf{v} \leftarrow 0, \quad \mathbf{a} \leftarrow 0,$
8:             for node in gridNodeList do
9:                 $\mathbf{v} \leftarrow \mathbf{v} + \mathbf{v}_g^\star$[node] $*$ $S_{gp}$[node]          ▷Update particle velocity
10:                $\mathbf{a} \leftarrow \mathbf{a} + \mathbf{a}_g$[node] $*$ $S_{gp}$[node]          ▷Update particle acceleration
11:            end for
12:            $\mathbf{x}_p^{n+1} \leftarrow \mathbf{x}_p^n + \mathbf{v} * \Delta t$          ▷Update position
13:            $\mathbf{u}_p^{n+1} \leftarrow \mathbf{u}_p^n + \mathbf{v} * \Delta t$          ▷Update displacement
14:            $\mathbf{v}_p^{n+1} \leftarrow \mathbf{v}_p^n + \mathbf{a} * \Delta t$          ▷Update velocity
15:        end for
16:    end for
17:    DELETEROGUEPARTICLES()          ▷Delete particles that are to be eroded.
18:    return $V_p^{n+1}, \mathbf{u}_p^{n+1}, \mathbf{v}_p^{n+1}, \mathbf{x}_p^{n+1}, m_p, \boldsymbol{h}_p^{n+1}$
19: end procedure
```



5 — Example MPM material model

In this chapter, we will examine the pseudocode for a reasonably complex material model that exercises most of the machinery for explicitly time integrated MPM material models in Vaango. The material model discussed in here is called Arena. A detailed description of the theory behind the model can be found in the Vaango Theory manual.

The main purpose of the material models is to compute the stress in a MPM particle given a state of deformation.

The stress tensor computation procedure calls the computeStressTensor routine that is specific to each constitutive model.

Algorithm 17 Computing the stress tensor

```
Require: $\Delta t$, $\mathbf{x}_p^n$, $m_p$, $V_p^{n+1}$, $\boldsymbol{h}_p^n$, $\boldsymbol{l}_p^{n+1}$, $\boldsymbol{F}_p^{n+1}$, $\boldsymbol{\sigma}_p^n$, $\boldsymbol{\eta}_p^n$, $\rho_0$, materialList, mpmFlags, constitutiveModel
1: procedure computeStressTensor
2:     for matl in materialList do
3:         $\boldsymbol{\sigma}_p^{n+1}$, $\boldsymbol{\eta}_p^{n+1}$ ← constitutiveModel[matl].computeStressTensor($\Delta t$, $\mathbf{x}_p^n$, $m_p$, $V_p^{n+1}$, $\boldsymbol{h}_p^n$,
           ↪ $\boldsymbol{l}_p^{n+1}$, $\boldsymbol{F}_p^{n+1}$, $\boldsymbol{\sigma}_p^n$, $\boldsymbol{\eta}_p^n$, $\rho_0$, mpmFlags)    ▷ Update the stress and any internal variables
           ↪ needed by the constitutive model
4:     end for
5:     return $\boldsymbol{\sigma}_p^{n+1}$, $\boldsymbol{\eta}_p^{n+1}$
6: end procedure
```

The Arena model is a high strain-rate soil plasticity model that uses a "consistency bisection" algorithm to find the plastic strain direction and to update the internal state variables. A closest-point return algorithm in transformed stress space is used to project the trial stress state on to the yield surface. Because of the nonlinearities in the material models, it is easier to solve the problem by dividing the strain increment to substeps.

The Arena model treats the porosity (ϕ) and saturation (S_w) as internal variables in addition to the hydrostatic compressive strength (X), the isotropic backstress (ζ), and the plastic strain ($\boldsymbol{\varepsilon}^p$).

5.1 Initialization of the model

The model is initialized in two steps. In the first step, the constitutive model object is created followed by initialization of the stress (and the deformation gradient if needed). Here

- ϕ_0 : The initial porosity.
- S_0 : The initial porosity.
- $n_{\max}$: The maximum number of subcycles in the plastic return algorithm.
- $\varepsilon_{v,p}^{e,n}$: The elastic volumetric strain at a particle at $t = t_n$.
- $\boldsymbol{\sigma}_p^n$: The dynamic Cauchy stress at a particle at $t = t_n$.
- $\boldsymbol{\sigma}_{qs,p}^n$: The quasistatic Cauchy stress at a particle at $t = t_n$.
- $\boldsymbol{\sigma}^0$: The initial Cauchy stress at a particle.

Algorithm 18 Creating the Arena constitutive model object

```
Require: mpmFlags, xmlProblemSpec
1: procedure CREATECONSTITUTIVEMODEL
2:     elasticityModel ← ElasticModuliModelFactory.CREATE(xmlProblemSpec)
3:     yieldCondition ← YieldConditionFactory.CREATE(xmlProblemSpec)
4:     $p_0, p_1, p_1^{sat}, p_2$ ← READCRUSHCURVEPARAMETERS(xmlProblemSpec)
5:     $\phi_0, S_0, \overline{p_0^w}$ ← READINITIALPOROSITYANDSATURATION(xmlProblemSpec)
6:     $n_{\max}$ ← READSUBCYCLINGCHARACTERISTICNUMBER(xmlProblemSpec)
7:     return elasticityModel, yieldCondition, $\phi_0, S_0, \overline{p_0^w}, n_{\max}, p_0, p_1, p_1^{sat}, p_2$
8: end procedure
```

Algorithm 19 Initializing the Arena particle variables

```
Require: $\phi_0, S_0, \overline{p_0^w}$, particleList, $V_p^0, m_p^0, \mathbf{v}_p^0$, fluidParams, elasticityModel, yieldCondition
1: procedure INITIALIZE
2:     yieldCondition.INITIALIZE(particleList, $V_p^0$)
3:     yieldParams ← yieldCondition.GETPARAMETERS()
4:     for part in particleList do
5:         $\phi_p^0$[part] ← $\phi_0$
6:         $S_{w,p}^0$[part] ← $S_0$
7:         $\chi_p^0$[part] ← 0
8:         $\varepsilon_{v,p}^{e,0}$[part] ← 0
9:         $\boldsymbol{\sigma}_p^0$[part] ← (fluidParams.$\overline{p_0^w}$) $\boldsymbol{I}$
10:        $\boldsymbol{\sigma}_{qs,p}^0$[part] ← $\boldsymbol{\sigma}^0$
11:    end for
12:    $\Delta t$ ← COMPUTESTABLETIMESTEP($V_p^0, m_p, \mathbf{v}_p^0$, elasticityModel)
13:    return $\phi_p^0, S_{w,p}^0, \chi_p^0, \varepsilon_{v,p}^{e,0}, \boldsymbol{\sigma}_{qs,p}^0, \boldsymbol{\sigma}_p^0, \Delta t$
14: end procedure
```

5.2 Computing the stress and internal variables

The COMPUTESTRESSTENSOR routine in the partially saturated Arena model assumes that the Biot coefficient is $B = 1$, and has the following form. Here we introduce the new variables

- ϕ_p^n, ϕ_p^{n+1} : The porosity at $t = t_n$ and $t = t_{n+1}$.
- $S_{w,p}^n, S_{w,p}^{n+1}$: The saturation at $t = t_n$ and $t = t_{n+1}$.
- $a_{1,p}, a_{2,p}, a_{3,p}, a_{4,p}$: Yield condition parameters at each particle.
- $p_{3,p}^n$: The particle crush curve parameter p_3 at $t = t_n$.
- X_p^n : The particle hydrostatic compressive strength at $t = t_n$.
- κ_p^n : The yield function branch point at $t = t_n$.
- $\boldsymbol{\epsilon}_p^{p,n}$: The particle plastic strain tensor at $t = t_n$.
- $\boldsymbol{\alpha}_p^n$: The particle backstress tensor at $t = t_n$.

- $\mathbf{d}^n$: The particle rate of deformation tensor at $t = t_n$.
- $\boldsymbol{R}^n, \boldsymbol{U}^n$: The particle rotation and stretch tensors at $t = t_n$.
- K^n, G^n : The particle bulk and shear modulus at $t = t_n$.

Algorithm 20 Computing the Arena stress tensor

```
Require: $\Delta t$, $\mathbf{x}_p^n$, $m_p$, $V_p^{n+1}$, $\boldsymbol{h}_p^n$, $\boldsymbol{l}_p^{n+1}$, $\boldsymbol{F}_p^n$, $\boldsymbol{F}_p^{n+1}$, $X_p^n$, $\kappa_p^n$, $\varepsilon_{v,p}^n$, $p_{3,p}^n$, $\boldsymbol{\epsilon}_p^{\text{p},n}$, $\boldsymbol{\alpha}_p^n$ $\phi_p^n$, $S_{w,p}^n$, $\chi_p^n$, $\varepsilon_{v,p}^{\text{e},n}$, $\boldsymbol{\sigma}_{qs,p}^n$, $\boldsymbol{\sigma}_p^n$, $\rho_0$,
    particleList, mpmFlags, elasticityModel, yieldCondition
 1: procedure COMPUTESTRESSTENSOR
 2:     yieldParams ← yieldCondition.GETPARAMETERS()
 3:     $a_{1,p}$, $a_{2,p}$, $a_{3,p}$, $a_{4,p}$, $I_{1,p}^{\text{peak}}$, $R_{c,p}$ ← yieldCondition.GETLOCALVARIABLES()
        ↪                                    ▷ Yield condition parameters vary at each particle.
        ↪ Get the per-particle values of these parameters.
 4:     for part in particleList do
 5:         $\chi_p^{n+1}$[part] ← $\chi_p^n$[part]                    ▷ Copy over failure indicator variable
 6:         $\boldsymbol{d}^{n+1}$ ← $[\boldsymbol{l}_p^{n+1}$[part] $+(\boldsymbol{l}_p^{n+1}$[part]$)^T]/2$      ▷ Compute rate of deformation
 7:         $\boldsymbol{R}^n$, $\boldsymbol{U}^n$ ← POLARDECOMPOSITION($\boldsymbol{F}_p^n$[part])        ▷ Compute rotation and stretch tensors
 8:         $\boldsymbol{d}_{\text{unrot}}^{n+1}$ ← $(\boldsymbol{R}^n)^T \cdot \boldsymbol{d}^{n+1} \cdot \boldsymbol{R}^n$          ▷ Unrotate the rate of deformation tensor
 9:         $\boldsymbol{\sigma}_{qs,\text{unrot}}^n$ ← $(\boldsymbol{R}^n)^T \cdot \boldsymbol{\sigma}_{qs,p}^n$[part]$\cdot\boldsymbol{R}^n$          ▷ Unrotate the quasistatic stress
10:         $\boldsymbol{\sigma}_{\text{unrot}}^n$ ← $(\boldsymbol{R}^n)^T \cdot \boldsymbol{\sigma}_p^n$[part]$\cdot\boldsymbol{R}^n$                ▷ Unrotate the total stress
11:         $K^n$, $G^n$, $\boldsymbol{s}^n$, $(\overline{p^w})^n$, $I_1^{\text{eff},n}$, $\sqrt{J_2^n}$, $r^n$, $z_{\text{eff}}^n$, $\varepsilon_v^{\text{p},n}$ ←
            ↪ COMPUTEELASTICPROPERTIES($\boldsymbol{\sigma}_{qs,\text{unrot}}^n$[part],
            ↪ $\phi_p^n$[part], $S_{w,p}^n$[part], $\boldsymbol{\epsilon}_p^{\text{p},n}$[part], $\boldsymbol{\alpha}_p^n$[part], $p_{3,p}^n$[part])
            ↪                       ▷ Compute elastic properties and stress invariants
12:         isSuccess, $\boldsymbol{\sigma}_{qs,\text{unrot}}^{n+1}$, $\phi_{qs}^{n+1}$, $S_{w,qs}^{n+1}$, $X_p^{n+1}$[part], $\boldsymbol{\alpha}_p^{n+1}$[part], $\boldsymbol{\epsilon}_p^{\text{p},n+1}$[part] ←
            ↪ RATEINDEPENDENTPLASTICUPDATE($\Delta t$, $\boldsymbol{d}_{\text{unrot}}^{n+1}$, $K^n$, $G^n$, $\boldsymbol{s}^n$, $(\overline{p^w})^n$, $I_1^{\text{eff},n}$, $\sqrt{J_2^n}$, $r^n$, $z_{\text{eff}}^n$,
            ↪ $\varepsilon_v^{\text{p},n}$, $\boldsymbol{\sigma}_{qs,\text{unrot}}^n$, $\phi_p^n$[part], $S_{w,p}^n$[part], $X_p^n$[part], $\boldsymbol{\alpha}_p^n$[part], $\boldsymbol{\epsilon}_p^{\text{p},n}$[part], $p_{3,p}^n$[part],
            ↪ $a_{1,p}$[part], $a_{2,p}$[part], $a_{3,p}$[part], $a_{4,p}$[part])
            ↪        ▷ Compute updated quasistatic state using the consistency bisection algorithm
13:         if isSuccess = FALSE then
14:             FLAGPARTICLEFORDELETION(part)
15:         end if
16:         $\boldsymbol{\sigma}_{\text{unrot}}^{n+1}$ ← RATEDEPENDENTPLASTICUPDATE($\Delta t$, $\boldsymbol{d}_{\text{unrot}}^{n+1}$, $\boldsymbol{\sigma}_{qs,\text{unrot}}^n$, $\boldsymbol{\sigma}_{qs,\text{unrot}}^{n+1}$, $\phi_p^n$[part], $\phi_{qs}^{n+1}$,
            ↪ $S_{w,p}^n$[part], $S_{w,qs}^{n+1}$, $X_p^n$[part], $X_p^{n+1}$[part], $\boldsymbol{\alpha}_p^n$[part], $\boldsymbol{\alpha}_p^{n+1}$[part],
            ↪ $\boldsymbol{\epsilon}_p^{\text{p},n}$[part], $\boldsymbol{\epsilon}_p^{\text{p},n+1}$[part], $p_{3,p}^n$[part],
            ↪ $a_{1,p}$[part], $a_{2,p}$[part], $a_{3,p}$[part], $a_{4,p}$[part])
17:         $\boldsymbol{R}^{n+1}$, $\boldsymbol{U}^{n+1}$ ← POLARDECOMPOSITION($\boldsymbol{F}_p^{n+1}$[part])   ▷ Compute rotation and stretch tensors
18:         $\boldsymbol{\sigma}_{p,qs}^{n+1}$[part] ← $\boldsymbol{R}^{n+1} \cdot \boldsymbol{\sigma}_{qs,\text{unrot}}^{n+1}\cdot(\boldsymbol{R}^{n+1})^T$          ▷ Rotate the quasistatic stress
19:         $\boldsymbol{\sigma}_p^{n+1}$[part] ← $\boldsymbol{R}^{n+1} \cdot \boldsymbol{\sigma}_{\text{unrot}}^{n+1}\cdot(\boldsymbol{R}^{n+1})^T$              ▷ Rotate the dynamic stress
20:     end for
21:     $\Delta t^{n+1}$ ← COMPUTESTABLETIMESTEP($V_p^{n+1}$, $m_p$, $\boldsymbol{\sigma}_p^{n+1}$, $\boldsymbol{\sigma}_p^n$, $\boldsymbol{l}_p^{n+1}$, elasticityModel)
22:     return $\boldsymbol{\sigma}_p^{n+1}$, $\boldsymbol{\sigma}_{p,qs}^{n+1}$, $\phi_p^{n+1}$, $S_{w,p}^{n+1}$, $\boldsymbol{\alpha}_p^{n+1}$, $X_p^{n+1}$, $\boldsymbol{\epsilon}_p^{\text{p},n+1}$, $\Delta t^{n+1}$.
23: end procedure
```

5.2.1 Compute elastic properties

The pseudocode for the generic COMPUTEELASTICPROPERTIES is listed below. The function has side effects beyond computing the elastic properties and should be used carefully. Note that the subscript p has been dropped for simplicity because all quantities are particle-based.

Algorithm 21 Computing the elastic properties

```
Require: $\boldsymbol{\sigma}$, $\phi$, $S_w$, $\boldsymbol{\epsilon}^{\mathrm{p}}$, $\boldsymbol{\alpha}$, $p_3$, elasticityModel
 1: procedure COMPUTEELASTICPROPERTIES
 2:     $\boldsymbol{s}$, $\overline{p^w}$, $I_1^{\mathrm{eff}}$, $\sqrt{J_2}$, $r$, $z_{\mathrm{eff}} \leftarrow$ UPDATESTRESSINVARIANTS($\boldsymbol{\sigma}$, $\boldsymbol{\alpha}$)   ▷Compute the deviatoric stress and the invariants of the input stress tensor.
 3:     $\varepsilon_v^{\mathrm{p}} \leftarrow$ UPDATEVOLUMETRICPLASTICSTRAIN($\boldsymbol{\epsilon}^{\mathrm{p}}$)   ▷Compute the volumetric plastic strain from the input plastic strain tensor.
 4:     $K, G \leftarrow$ elasticityModel.GETCURRENTELASTICMODULI($I_1^{\mathrm{eff}}$, $\overline{p^w}$, $\varepsilon_v^{\mathrm{p}}$, $\phi$, $S_w$)   ▷Compute the elastic moduli corresponding to the input state.
 5:     if useDisaggregationAlgorithm = TRUE then
 6:         scale = MAX(exp$[-(p_3 + \varepsilon_v^{\mathrm{p}})]$, $10^{-5}$)
 7:         $K \leftarrow K$*scale ,  $G \leftarrow G$*scale
 8:     end if
 9:     return $K$, $G$, $\boldsymbol{s}$, $\overline{p^w}$, $I_1^{\mathrm{eff}}$, $\sqrt{J_2}$, $r$, $z_{\mathrm{eff}}$, $\varepsilon_v^{\mathrm{p}}$
10: end procedure
```

Algorithm 22 Updating the stress invariants

```
 1: procedure UPDATESTRESSINVARIANTS($\boldsymbol{\sigma}$, $\boldsymbol{\alpha}$)
 2:     $I_1 \leftarrow \mathrm{tr}(\boldsymbol{\sigma})$
 3:     $\boldsymbol{s} \leftarrow \boldsymbol{\sigma} - (I_1/3)\boldsymbol{I}$                         ▷Compute deviatoric stress
 4:     $\overline{p^w} \leftarrow -\mathrm{tr}(\boldsymbol{\alpha})/3$                       ▷Compute pore pressure
 5:     $I_1^{\mathrm{eff}} \leftarrow I_1 + 3\overline{p^w}$ ,  $\sqrt{J_2} \leftarrow \sqrt{(1/2)\boldsymbol{s}:\boldsymbol{s}}$   ▷Compute invariants of the effective stress
 6:     $r \leftarrow \sqrt{2J_2}$ ,  $z_{\mathrm{eff}} \leftarrow I_1^{\mathrm{eff}}/\sqrt{3}$   ▷Compute Lode coordinates of the effective stress
 7:     return $\boldsymbol{s}$, $\overline{p^w}$, $I_1^{\mathrm{eff}}$, $\sqrt{J_2}$, $r$, $z_{\mathrm{eff}}$
 8: end procedure
```

The elastic modulus computation procedures:

The functions used to compute the moduli are listed in the pseudocode below.

Algorithm 23 Computing the current elastic moduli

```
Require: $I_1^{\mathrm{eff}}$, $\overline{p^w}$, $\varepsilon_v^{\mathrm{p}}$, $\phi$, $S_w$
 1: procedure GETCURRENTELASTICMODULI
 2:     $\bar{I}_1^{\mathrm{eff}} \leftarrow -I_1^{\mathrm{eff}}$, $\bar{\varepsilon}_v^{\mathrm{p}} \leftarrow -\varepsilon_v^{\mathrm{p}}$,
 3:     $K \leftarrow 0$, $G \leftarrow 0$
 4:     if $S_w > 0$ then
 5:         $K, G \leftarrow$ COMPUTEPARTIALSATURATEDMODULI($\bar{I}_1^{\mathrm{eff}}$, $\overline{p^w}$, $\bar{\varepsilon}_v^{\mathrm{p}}$, $\phi$, $S_w$)
 6:     else
 7:         $K, G \leftarrow$ COMPUTEDRAINEDMODULI($\bar{I}_1^{\mathrm{eff}}$, $\bar{\varepsilon}_v^{\mathrm{p}}$)
 8:     end if
 9:     return $K$, $G$
10: end procedure
```

Algorithm 24 Computing the partially saturated elastic moduli

```
Require: $K_{s0}$, $n_s$, $\bar{p}_{s0}$, $K_{w0}$, $n_w$, $\bar{p}_{w0}$, $\gamma$, $\bar{p}_r$
 1: procedure COMPUTEPARTIALSATURATEDMODULI($\bar{I}_1^{\mathrm{eff}}$, $\overline{p^w}$, $\bar{\varepsilon}_v^{\mathrm{p}}$, $\phi$, $S_w$)
 2:     if $\bar{I}_1^{\mathrm{eff}} > 0$ then
 3:         $\bar{p}^{\mathrm{eff}} \leftarrow \bar{I}_1^{\mathrm{eff}}/3$
 4:         $K_s \leftarrow K_{s0} + n_s(\bar{p}^{\mathrm{eff}} - \bar{p}_{s0})$
```

```
5:          $K_w \leftarrow K_{wo} + n_w(\overline{p^w} - \bar{p}_{wo})$
6:          $K_a \leftarrow \gamma(\overline{p^w} + \bar{p}_r)$
7:          $K_d, G \leftarrow$ computeDrainedModuli($\bar{I}_1^{\text{eff}}, \overline{\varepsilon_v^{\text{p}}}$)
8:          $K_f \leftarrow 1/[S_w/K_w + (1 - S_w)/K_a]$      ▷Bulk modulus of air + water mixture
9:          numer $\leftarrow (1 - K_d/K_s)^2$
10:         denom $\leftarrow (1/K_s)(1 - K_d/K_s) + \phi\,(1/K_f - 1/K_s)$
11:         $K \leftarrow K_d$ + numer/denom      ▷Bulk modulus of partially saturated material
              ↪ (Biot-Gassman model)
12:     else
13:         $K, G \leftarrow$ computeDrainedModuli($\bar{I}_1, \overline{\varepsilon_v^{\text{p}}}$)
14:     end if
15:     return $K, G$
16: end procedure
```

Algorithm 25 Computing the drained elastic moduli

```
Require: $K_{so}, n_s, \bar{p}_{so}, b_0, b_1, b_2, b_3, b_4, G_0, \nu_1, \nu_2$
1:  procedure computeDrainedModuli($\bar{I}_1^{\text{eff}}, \overline{\varepsilon_v^{\text{p}}}$)
2:      if $\bar{I}_1^{\text{eff}} > 0$ then
3:          $\bar{p}^{\text{eff}} \leftarrow \bar{I}_1^{\text{eff}}/3$
4:          $K_s \leftarrow K_{so} + n_s(\bar{p}^{\text{eff}} - \bar{p}_{so})$
5:          $K_s^{\text{ratio}} \leftarrow K_s/(1 - n_s\bar{p}^{\text{eff}}/K_s)$
6:          $\varepsilon_v^{\text{e}} \leftarrow$ pow$((b_3\bar{p}^{\text{eff}})/(b_1K_s - b_2\bar{p}^{\text{eff}}), (1/b_4))$;
7:          $y \leftarrow$ pow$(\varepsilon_v^{\text{e}}, b_4)$
8:          $z \leftarrow b_2 y + b_3$
9:          $K \leftarrow K_s^{\text{ratio}}[b_0 + (1/\varepsilon_v^{\text{e}})b_1b_3b_4y/z^2]$;      ▷ Compute compressive bulk modulus
10:         $\nu = \nu_1 + \nu_2 \exp(-K/K_s)$
11:         $G \leftarrow G_0$
12:         if $\nu > 0$ then
13:             $G \leftarrow 1.5K(1 - 2\nu)/(1 + \nu)$      ▷Update the shear modulus (if $\nu_1, \nu_2 > 0$)
14:         end if
15:     else
16:         $K \leftarrow b_0 K_{so}$      ▷Tensile bulk modulus = Bulk modulus at $p = 0$
17:         $G \leftarrow G_0$      ▷Tensile shear modulus
18:     end if
19:     return $K, G$
20: end procedure
```

5.2.2 Rate-independent stress update

The partially saturated soil model uses a "consistency bisection" algorithm to find the plastic strain direction and to update the internal state variables. A closest-point return algorithm in transformed stress space is used to project the trial stress state on to the yield surface. Because of the nonlinearities in the material models, it is easier to solve the problem by dividing the strain increment into substeps.

The pseudocode for the algorithm is given below. All quantities are particle-based and the subscript p has been dropped for convenience.

Algorithm 26 The rate-independent stress and internal variable update algorithm

```
Require: $\Delta t, \boldsymbol{d}^{n+1}, K^n, G^n, \boldsymbol{s}^n, (\overline{p^w})^n, I_1^{\text{eff},n}, \sqrt{J_2^n}, r^n, z_{\text{eff}}^n, \varepsilon_v^{\text{p},n}, a_1, a_2, a_3, a_4, p_3^n, \boldsymbol{\sigma}_{qs}^n, \phi^n, S_w^n, X^n, \boldsymbol{\alpha}^n, \boldsymbol{\varepsilon}^{\text{p},n}$,
    $n_{\max}, \epsilon_{\text{sub}}, \chi_{\max}$
1:  procedure rateIndependentPlasticUpdate
```

```
2:      $\boldsymbol{\sigma}_{\text{trial}} \leftarrow$ COMPUTETRIALSTRESS($\boldsymbol{\sigma}_{qs}^{n}, K^{n}, G^{n}, \boldsymbol{d}^{n+1}, \Delta t$)    ▷Compute trial stress
3:      $\boldsymbol{\alpha}^{\text{trial}} \leftarrow \boldsymbol{\alpha}^{n}, p_3^{\text{trial}} \leftarrow p_3^{n}, \phi^{\text{trial}} \leftarrow \phi^{n}, S_w^{\text{trial}} \leftarrow S_w^{n},$
          ↪ $X^{\text{trial}} \leftarrow X^{n}, \boldsymbol{\varepsilon}^{\text{p,trial}} \leftarrow \boldsymbol{\varepsilon}^{\text{p},n}$    ▷Set all other trial quantities to the values
          ↪ at the beginning of the timestep
4:      $K^{\text{trial}}, G^{\text{trial}}, \boldsymbol{s}^{\text{trial}}, (\overline{p^{w}})^{\text{trial}}, I_1^{\text{eff,trial}}, \sqrt{J_2^{\text{trial}}}, r^{\text{trial}}, z_{\text{eff}}^{\text{trial}}, \varepsilon_v^{\text{p,trial}} \leftarrow$
          ↪ COMPUTEELASTICPROPERTIES($\boldsymbol{\sigma}_{\text{trial}}, \phi^{\text{trial}}, S_w^{\text{trial}}, \boldsymbol{\varepsilon}^{\text{p,trial}}, \boldsymbol{\alpha}^{\text{trial}}, p_3^{\text{trial}}$)    ▷Update the trial
          ↪ values of the moduli and compute the invariants of the trial stress
5:      $n_{\text{sub}} \leftarrow$ COMPUTESTEPDIVISIONS($n_{\max}, \epsilon_{\text{sub}}, K^{n}, K^{\text{trial}}, I_1^{\text{peak}}, a_1, X^{n}, \boldsymbol{\sigma}_{qs}^{n}, \boldsymbol{\sigma}_{\text{trial}}$)
          ↪    ▷Compute number of substeps used by the return algorithm
6:      if $n_{\text{sub}} < 0$ then
7:          return isSuccess = FALSE
8:      end if
9:      $\delta t \leftarrow \dfrac{\Delta t}{n_{\text{sub}}}$    ▷Substep timestep
10:     $\chi \leftarrow 1, t_{\text{local}} \leftarrow 0$    ▷Initialize substep multiplier and accumulated time increment
11:     $\boldsymbol{\sigma}^{k} \leftarrow \boldsymbol{\sigma}_{qs}^{n}, \boldsymbol{\varepsilon}^{\text{p},k} \leftarrow \boldsymbol{\varepsilon}^{\text{p},n}, \phi^{k} \leftarrow \phi^{n}, S_w^{k} \leftarrow S_w^{n}, X^{k} \leftarrow X^{n}, \boldsymbol{\alpha}^{k} \leftarrow \boldsymbol{\alpha}^{n}, K^{k} \leftarrow K^{n}, G^{k} \leftarrow G^{n}, p_3^{k} \leftarrow p_3^{n},$
          ↪ $\boldsymbol{s}^{k} \leftarrow \boldsymbol{s}^{n}, (\overline{p^{w}})^{k} \leftarrow (\overline{p^{w}})^{n}, I_1^{\text{eff},k} \leftarrow I_1^{\text{eff},n}, \sqrt{J_2^{k}} \leftarrow \sqrt{J_2^{n}}, r^{k} \leftarrow r^{n}, z_{\text{eff}}^{k} \leftarrow z_{\text{eff}}^{n}, \varepsilon_v^{\text{p},k} \leftarrow \varepsilon_v^{\text{p},n}$
12:     repeat
13:         isSuccess, $\boldsymbol{\sigma}^{k+1}, \boldsymbol{\varepsilon}^{\text{p},k+1}, \phi^{k+1}, S_w^{k+1}, X^{k+1}, \boldsymbol{\alpha}^{k+1}, K^{k+1}, G^{k+1}, p_3^{k+1} \leftarrow$
          ↪ COMPUTESUBSTEP($\boldsymbol{\sigma}^{k}, \boldsymbol{\varepsilon}^{\text{p},k}, \phi^{k}, S_w^{k}, X^{k}, \boldsymbol{\alpha}^{k}, K^{k}, G^{k}, \boldsymbol{s}^{k}, (\overline{p^{w}})^{k}, I_1^{\text{eff},k}, \sqrt{J_2^{k}}, r^{k}, z_{\text{eff}}^{k},$
          ↪ $\varepsilon_v^{\text{p},k}, p_3^{k}, \boldsymbol{d}^{n+1}, \delta t$)
          ↪    ▷Compute updated stress and internal variables for the current substep
14:         if isSuccess = TRUE then
15:             $t_{\text{local}} \leftarrow t_{\text{local}} + \delta t$
16:             $\boldsymbol{\sigma}^{k} \leftarrow \boldsymbol{\sigma}^{k+1}, \boldsymbol{\varepsilon}^{\text{p},k} \leftarrow \boldsymbol{\varepsilon}^{\text{p},k+1}, \phi^{k} \leftarrow \phi^{k+1}, S_w^{k} \leftarrow S_w^{k+1}, X^{k} \leftarrow X^{k+1}, \boldsymbol{\alpha}^{k} \leftarrow \boldsymbol{\alpha}^{k+1}$
17:             $K^{k} \leftarrow K^{k+1}, G^{k} \leftarrow G^{k+1}, p_3^{k} \leftarrow p_3^{k+1}$
18:         else
19:             $\delta t \leftarrow \delta t/2$    ▷Halve the timestep
20:             $\chi \leftarrow 2\chi$    ▷Keep a count of how many times the timestep has been halved.
21:             if $\chi > \chi_{\max}$ then
22:                 return isSuccess = FALSE, $\boldsymbol{\sigma}^{k}, \phi^{k}, S_w^{k}, X^{k}, \boldsymbol{\alpha}^{k}, \boldsymbol{\varepsilon}^{\text{p},k}, K^{k}, G^{k}, p_3^{k}$
                  ↪    ▷Algorithm has failed to converge
23:             end if
24:         end if
25:     until $t_{\text{local}} \geq \Delta t$
26:     $\boldsymbol{\sigma}_{qs}^{n+1} \leftarrow \boldsymbol{\sigma}^{k+1}, \boldsymbol{\alpha}^{n+1} \leftarrow \boldsymbol{\alpha}^{k+1}, \boldsymbol{\varepsilon}^{\text{p},n+1} \leftarrow \boldsymbol{\varepsilon}^{\text{p},k+1}, \phi^{n+1} \leftarrow \phi^{k+1}, S_w^{n+1} \leftarrow S_w^{k+1}, X^{n+1} \leftarrow X^{k+1}$
27:     $K^{n+1} \leftarrow K^{k+1}, G^{n+1} \leftarrow G^{k+1}, p_3^{n+1} \leftarrow p_3^{k+1}$
28:     return isSuccess = TRUE, $\boldsymbol{\sigma}_{qs}^{n+1}, \phi^{n+1}, S_w^{n+1}, X^{n+1}, \boldsymbol{\alpha}^{n+1}, \boldsymbol{\varepsilon}^{\text{p},n+1}, K^{n+1}, G^{n+1}, p_3^{n+1}$
          ↪    ▷Algorithm has converged
29: end procedure
```

Computing the trial stress

The pseudocode of the trial stress algorithm is given below.

Algorithm 27 Computing the trial stress

```
1: procedure COMPUTETRIALSTRESS($\boldsymbol{\sigma}_{qs}^{n}, K^{n}, G^{n}, \boldsymbol{d}^{n+1}, \Delta t$)
2:      $\Delta\boldsymbol{\varepsilon} \leftarrow \boldsymbol{d}^{n+1}\Delta t$    ▷Total strain increment
3:      $\Delta\boldsymbol{\varepsilon}^{\text{iso}} \leftarrow \frac{1}{3}\text{tr}(\Delta\boldsymbol{\varepsilon})\boldsymbol{I}$
4:      $\Delta\boldsymbol{\varepsilon}^{\text{dev}} \leftarrow \Delta\boldsymbol{\varepsilon} - \Delta\boldsymbol{\varepsilon}^{\text{iso}}$
5:      $\boldsymbol{\sigma}_{\text{trial}} \leftarrow \boldsymbol{\sigma}_{qs}^{n} + 3K^{n}\Delta\boldsymbol{\varepsilon}^{\text{iso}} + 2G^{n}\Delta\boldsymbol{\varepsilon}^{\text{dev}}$
```

```
6:     return $\boldsymbol{\sigma}_{\text{trial}}$
7: end procedure
```

Computing the number of subcycles in the return algorithm

To allow for nonlinear parameter variations, the algorithm breaks a trial loading step into subcycles. The algorithm below, determines the number of substeps based on the magnitude of the trial stress increment relative to the characteristic dimensions of the yield surface. Another comparison uses the value of the pressure dependent elastic properties at $\boldsymbol{\sigma}_{qs}^n$ and $\boldsymbol{\sigma}_{\text{trial}}$ and adjusts the number of substeps if there is a large change in elastic moduli. This ensures an accurate solution for nonlinear elasticity even with fully elastic loading.

Algorithm 28 Computing the number of subcycles

```
Require: $n_{\text{max}}, \epsilon_{\text{sub}} \leftarrow 10^{-4}, K^n, K^{\text{trial}}, I_1^{\text{peak}}, a_1, X^n, \boldsymbol{\sigma}_{qs}^n, \boldsymbol{\sigma}_{\text{trial}}$
1: procedure COMPUTESTEPDIVISIONS
2:     $n_{\text{bulk}} \leftarrow \lceil |K^n - K^{\text{trial}}| / K^n \rceil$        ▷Compute change in bulk modulus
3:     $\Delta\boldsymbol{\sigma} \leftarrow \boldsymbol{\sigma}_{\text{trial}} - \boldsymbol{\sigma}_{qs}^n$
4:     $L \leftarrow \frac{1}{2}(I_1^{\text{peak}} - X^n)$
5:     if $a_1 > 0$ then
6:         $L \leftarrow \text{MIN}(L, a_1)$
7:     end if
8:     $n_{\text{yield}} \leftarrow \lceil \epsilon_{\text{sub}} \times \|\Delta\boldsymbol{\sigma}\| / L \rceil$        ▷Compute trial stress increment relative to yield surface size
9:     $n_{\text{sub}} \leftarrow \text{MAX}(n_{\text{bulk}}, n_{\text{yield}})$        ▷$n_{sub}$ is the maximum of the two values
10:    if $n_{\text{sub}} > n_{\text{max}}$ then
11:        $n_{\text{sub}} \leftarrow -1$
12:    end if
13:    return $n_{\text{sub}}$
14: end procedure
```

Updating the stress for a substep: consistency bisection

This procedure computes the updated stress state for a substep that may be either elastic, plastic, or partially elastic. It uses Homel's consistency bisection and non-hardening return concepts [HGB15].

Algorithm 29 Computing the stress and internal variable update for a substep

```
Require: $\boldsymbol{d}^{n+1}, \delta t, \boldsymbol{\sigma}^k, \boldsymbol{\varepsilon}^{p,k}, \phi^k, S_w^k, X^k, \boldsymbol{\alpha}^k, K^k, G^k, \boldsymbol{s}^k, (\overline{p^w})^k, I_1^{\text{eff},k}, \sqrt{J_2^k}, r^k, z_{\text{eff}}^k, \varepsilon_v^{p,k}, p_3^k, a_1, a_2, a_3, a_4,$
    $I_1^{\text{peak}}, R_c, \beta,$ yieldCondition
1: procedure COMPUTESUBSTEP
2:     $\delta\boldsymbol{\varepsilon} \leftarrow \boldsymbol{d}^{n+1}\delta t$        ▷Compute strain increment
3:     $\boldsymbol{\sigma}_{\text{trial}} \leftarrow \text{COMPUTETRIALSTRESS}(\boldsymbol{\sigma}^k, K^k, G^k, \boldsymbol{d}^{n+1}, \Delta t)$        ▷Compute substep trial stress
4:     $\boldsymbol{\alpha}^{\text{trial}} \leftarrow \boldsymbol{\alpha}^k, K^{\text{trial}} \leftarrow K^k, G^{\text{trial}} \leftarrow G^k, p_3^{\text{trial}} \leftarrow p_3^k, \phi^{\text{trial}} \leftarrow \phi^k, S_w^{\text{trial}} \leftarrow S_w^k,$
         ↪ $X^{\text{trial}} \leftarrow X^k, \boldsymbol{\varepsilon}^{p,\text{trial}} \leftarrow \boldsymbol{\varepsilon}^{p,k}$        ▷Set all other trial quantities to the values
         ↪ at the beginning of the substep
5:     $K^{\text{trial}}, G^{\text{trial}}, \boldsymbol{s}^{\text{trial}}, (\overline{p^w})^{\text{trial}}, I_1^{\text{eff,trial}}, \sqrt{J_2^{\text{trial}}}, r^{\text{trial}}, z_{\text{eff}}^{\text{trial}}, \varepsilon_v^{p,\text{trial}} \leftarrow$
         ↪ COMPUTEELASTICPROPERTIES($\boldsymbol{\sigma}_{\text{trial}}, \phi^{\text{trial}}, S_w^{\text{trial}}, \boldsymbol{\varepsilon}^{p,\text{trial}}, \boldsymbol{\alpha}^{\text{trial}}, p_3^{\text{trial}}$)
         ↪        ▷Compute elastic moduli and stress invariants for the trial state
6:     isElastic ← yieldCondition.EVALYIELDCONDITION($I_1^{\text{eff,trial}}, \sqrt{J_2^{\text{trial}}}, X^{\text{trial}}, (\overline{p^w})^{\text{trial}}, \phi^{\text{trial}}, S_w^{\text{trial}},$
         ↪ $a_1, a_2, a_3, a_4, I_1^{\text{peak}}, R_c, \beta$)
         ↪        ▷Determine whether the trial stress is elastic or not
7:     if isElastic = TRUE then
```

```
 8:        $\boldsymbol{\sigma}^{k+1} \leftarrow \boldsymbol{\sigma}_{\text{trial}}$, $\boldsymbol{\varepsilon}^{\text{p},k+1} \leftarrow \boldsymbol{\varepsilon}^{\text{p,trial}}$, $\phi^{k+1} \leftarrow \phi^{\text{trial}}$, $S_w^{k+1} \leftarrow S_w^{\text{trial}}$, $X^{k+1} \leftarrow X^{\text{trial}}$, $\boldsymbol{\alpha}^{k+1} \leftarrow \boldsymbol{\alpha}^{\text{trial}}$
 9:        $K^{k+1} \leftarrow K^{\text{trial}}$, $G^{k+1} \leftarrow G^{\text{trial}}$, $p_3^{k+1} \leftarrow p_3^{\text{trial}}$
             ↪                                  ▷This is an elastic substep. Update the state to the trial value.
10:        isSuccess = TRUE
11:        return isSuccess, $\boldsymbol{\sigma}^{k+1}$, $\boldsymbol{\varepsilon}^{\text{p},k+1}$ $\phi^{k+1}$, $S_w^{k+1}$, $X^{k+1}$, $\boldsymbol{\alpha}^{k+1}$, $K^{k+1}$, $G^{k+1}$, $p_3^{k+1}$
12:    end if
13:    $\boldsymbol{\sigma}_{\text{fixed}}$, $\delta\boldsymbol{\varepsilon}^{\text{p}}_{\text{fixed}} \leftarrow$ NONHARDENINGRETURN($\boldsymbol{\sigma}^k$, $\delta\boldsymbol{\varepsilon}$, $X^k$, $K^k$, $G^k$, $(\overline{p^w})^k$,
         ↪ $\boldsymbol{s}^{\text{trial}}$, $\sqrt{J_2^{\text{trial}}}$, $r^{\text{trial}}$, $z_{\text{eff}}^{\text{trial}}$, $a_1$, $a_2$, $a_3$, $a_4$, $I_1^{\text{peak}}$, $R_c$, $\beta$)
         ↪                                  ▷Compute return to updated yield surface (no hardening)
14:    isSuccess, $\boldsymbol{\sigma}^{k+1}$, $\boldsymbol{\varepsilon}^{\text{p},k+1}$, $\boldsymbol{\alpha}^{k+1}$, $(\overline{p^w})^{k+1}$, $\phi^{k+1}$, $S_w^{k+1}$, $X^{k+1}$, $K^{k+1}$, $G^{k+1}$, $\boldsymbol{s}^{k+1}$, $(\overline{p^w})^{k+1}$, $I_1^{\text{eff},k+1}$, $\sqrt{J_2^{k+1}}$,
         ↪ $r^{k+1}$, $z_{\text{eff}}^{k+1}$, $\varepsilon_v^{\text{p},k+1} \leftarrow$ CONSISTENCYBISECTION($\delta\boldsymbol{\varepsilon}$, $\boldsymbol{\varepsilon}^{\text{p},k}$, $\boldsymbol{\sigma}^k$, $K^k$, $G^k$, $(\overline{p^w})^k$, $\phi^k$, $S_w^k$, $X^k$,
         ↪ $\boldsymbol{s}^{\text{trial}}$, $I_1^{\text{eff,trial}}$, $\sqrt{J_2^{\text{trial}}}$, $r^{\text{trial}}$, $z_{\text{eff}}^{\text{trial}}$, $\varepsilon_v^{\text{p,trial}}$, $p_3^{\text{trial}}$, $a_1$, $a_2$, $a_3$, $a_4$, $I_1^{\text{peak}}$, $R_c$, $\beta$, $i_{\max}$, $j_{\max}$,
         ↪ $\boldsymbol{\sigma}_{\text{fixed}}$, $\delta\boldsymbol{\varepsilon}^{\text{p}}_{\text{fixed}}$)   ▷The bisection return algorithm to take care of yield surface hardening.
15:    if iSuccess = FALSE then
16:        return isSuccess, $\boldsymbol{\sigma}^k$, $\boldsymbol{\varepsilon}^{\text{p},k}$, $\phi^k$, $S_w^k$, $X^k$, $\boldsymbol{\alpha}^k$, $K^k$, $G^k$, $p_3^k$
17:    end if
18:    return isSuccess, $\boldsymbol{\sigma}^{k+1}$, $\boldsymbol{\varepsilon}^{\text{p},k+1}$, $\phi^{k+1}$, $S_w^{k+1}$, $X^{k+1}$, $\boldsymbol{\alpha}^{k+1}$, $K^{k+1}$, $G^{k+1}$, $p_3^{k+1}$
19: end procedure
```

The nonhardening return algorithm

The nonhardening return algorithm uses a transformed space (see [HGB15]) where the computation is carried out in special Lode coordinates (z_{eff}, r') where

$$z_{\text{eff}} := \frac{\operatorname{tr}(\boldsymbol{\sigma} - \boldsymbol{\alpha})}{\sqrt{3}} \quad \text{and} \quad r' = \beta r \sqrt{\frac{3K}{2G}}\,, \quad r := \sqrt{2J_2}\,. \tag{5.1}$$

If the flow rule is non-associative, the yield surface parameter $\beta \neq 1$.

The nonhardening return algorithm pseudocode is listed below:

Algorithm 30 Non-hardening return algorithm

```
Require: $\boldsymbol{\sigma}^k$, $\delta\boldsymbol{\varepsilon}$, $X^k$, $K^k$, $G^k$, $(\overline{p^w})^k$, $\boldsymbol{s}^{\text{trial}}$, $\sqrt{J_2^{\text{trial}}}$, $r^{\text{trial}}$, $z_{\text{eff}}^{\text{trial}}$, $a_1$, $a_2$, $a_3$, $a_4$, $I_1^{\text{peak}}$, $R_c$, $\beta$, yieldCondition
 1: procedure NONHARDENINGRETURN
 2:    $r'_{\text{trial}} \leftarrow \beta\, r^{\text{trial}} \sqrt{\frac{3K^k}{2G^k}}$                    ▷Transform the trial r coordinate
 3:    $X_{\text{eff}}^k \leftarrow X^k + 3(\overline{p^w})^k$
 4:    $z_{\text{eff}}^{\text{close}}$, $r'_{\text{close}} \leftarrow$ yieldCondition.GETCLOSESTPOINT($K^k$, $G^k$, $X_{\text{eff}}^k$, $a_1$, $a_2$, $a_3$, $a_4$, $I_1^{\text{peak}}$, $R_c$, $\beta$,
         ↪ $z_{\text{eff}}^{\text{trial}}$, $r'_{\text{trial}}$)
 5:    $I_1^{\text{close}} \leftarrow \sqrt{3} z_{\text{eff}}^{\text{close}} - 3(\overline{p^w})^k$, $\sqrt{J_2^{\text{close}}} \leftarrow \frac{1}{\beta}\sqrt{\frac{G^k}{3K^k}}\, r'_{\text{close}}$
 6:    if $\sqrt{J_2^{\text{trial}}} > 0$ then
 7:        $\boldsymbol{\sigma}^{\text{fixed}} = \frac{1}{3} I_1^{\text{close}} \boldsymbol{I} + \frac{\sqrt{J_2^{\text{close}}}}{\sqrt{J_2^{\text{trial}}}} \boldsymbol{s}^{\text{trial}}$          ▷Compute updated total stress
 8:    else
 9:        $\boldsymbol{\sigma}^{\text{fixed}} = \frac{1}{3} I_1^{\text{close}} \boldsymbol{I} + \boldsymbol{s}^{\text{trial}}$   ▷Compute updated total stress when the trial stress is hydrostatic
10:    end if
11:    $\delta\boldsymbol{\sigma}_{\text{fixed}} \leftarrow \boldsymbol{\sigma}^{\text{fixed}} - \boldsymbol{\sigma}^k$                          ▷Compute stress increment
12:    $\delta\boldsymbol{\sigma}_{\text{fixed}}^{\text{iso}} \leftarrow \frac{1}{3}\operatorname{tr}(\delta\boldsymbol{\sigma}_{\text{fixed}})\boldsymbol{I}$,    $\delta\boldsymbol{\sigma}_{\text{fixed}}^{\text{dev}} \leftarrow \delta\boldsymbol{\sigma}_{\text{fixed}} - \delta\boldsymbol{\sigma}_{\text{fixed}}^{\text{iso}}$
13:    $\delta\boldsymbol{\varepsilon}^{\text{p,fixed}} = \delta\boldsymbol{\varepsilon} - \frac{1}{3K^k}\,\delta\boldsymbol{\sigma}_{\text{fixed}}^{\text{iso}} - \frac{1}{2G^k}\,\delta\boldsymbol{\sigma}_{\text{fixed}}^{\text{dev}}$        ▷Compute plastic strain increment
```

```
14:     return $\sigma^{\text{fixed}}$, $\delta\varepsilon^{\text{p,fixed}}$
15: end procedure
```

Finding the closest point in transformed space

Algorithm 31 Compute the closest point from the trial state to transformed non-hardening yield surface

```
Require: $K^k$, $G^k$, $X^k_{\text{eff}}$, $a_1$, $a_2$, $a_3$, $a_4$, $I_1^{\text{peak}}$, $R_c$, $\beta$, $z_{\text{eff}}^{\text{trial}}$, $r'_{\text{trial}}$
 1: procedure GETCLOSESTPOINT
 2:     $n_{\text{poly}} \leftarrow 5$
 3:     Set up $I_1^{\text{max}}$, $I_1^{\text{min}}$, $I_1^{\text{mid}} = 1/2(I_1^{\text{max}} + I_1^{\text{min}})$ limits of the yield surface.
 4:     Set up bisection: $\eta_{\text{low}} \leftarrow 0$, $\eta_{\text{high}} \leftarrow 1$, $\eta_{\text{mid}} \leftarrow 1/2(\eta_{\text{low}} + \eta_{\text{high}})$
 5:     while ABS$(\eta^{\text{high}} - \eta^{\text{low}}) >$ TOLERANCE do
 6:         $\mathbf{x}_{\text{poly}} \leftarrow$ GETYIELDSURFACEPOINTSALL_RPRIMEZ$(n_{\text{poly}}, K^k, G^k, X^k_{\text{eff}},$
            ↪ $a_1, a_2, a_3, a_4, I_1^{\text{peak}}, R_c, \beta\,)$
            ↪                ▷Get the polygon that represents the yield surface in $z_{eff}$-$r'$ space.
 7:         $\mathbf{x}_{\text{close}} \leftarrow$ FINDCLOSESTPOINT$(z_{\text{eff}}^{\text{trial}}, r'_{\text{trial}}, \mathbf{x}_{\text{poly}})$
            ↪          ▷Find the closest point in the discretized segments to the trial stress state.
 8:         Compute $I_1^{\text{close}}$ from $\mathbf{x}_{\text{close}}$
 9:         if $I_1^{\text{close}} < I_1^{\text{mid}}$ then
10:             $I_1^{\text{max}} \leftarrow I_1^{\text{mid}}$, $\eta_{\text{high}} \leftarrow \eta_{\text{mid}}$                  ▷Update mid point
11:         else
12:             $I_1^{\text{min}} \leftarrow I_1^{\text{mid}}$, $\eta_{\text{low}} \leftarrow \eta_{\text{mid}}$                   ▷Update mid point
13:         end if
14:         Recompute $I_1^{\text{mid}}$, $\eta_{\text{mid}}$ and update old closest point.
15:     end while
16:     return isSuccess = TRUE, $\mathbf{x}_{\text{close}}.z_{\text{eff}}$ , $\mathbf{x}_{\text{close}}.r'$
17: end procedure
```

Finding the yield surface polygon in z_{eff}-r' space

Algorithm 32 Find points in a closed polygon that describes the yield surface in z_{eff}-r' space

```
Require: $n_{\text{poly}}$, $K^k$, $G^k$, $X^k_{\text{eff}}$, $a_1$, $a_2$, $a_3$, $a_4$, $I_1^{\text{peak}}$, $R_c$, $\beta$
 1: procedure GETYIELDSURFACEPOINTSALL_RPRIMEZ
 2:     $\kappa \leftarrow I_1^{\text{peak}} - R_c(I_1^{\text{peak}} - X^k_{\text{eff}})$                  ▷Compute $\kappa$.
 3:     $I_1^{\text{eff}} \leftarrow$ LINSPACE(from = $X^k_{\text{eff}}$, to = $I_1^{\text{peak}}$, points = $n_{\text{poly}}$)
        ↪                          ▷Create an equally spaced set of $I_1^{eff}$ values.
 4:     for $I_1$ in $I_1^{\text{eff}}$ do
 5:         $F_f = a_1 - a_3 \exp(a_2 I_1) - a_4 I_1;$                          ▷Compute $F_f$.
 6:         $F_c^2 \leftarrow 1$
 7:         if $I_1 < \kappa$ and $X^k_{\text{eff}} < I_1$ then
 8:             $F_c^2 = 1 - \left[\frac{\kappa - I_1}{\kappa - X^k_{\text{eff}}}\right]^2;$                          ▷Compute $F_c$.
 9:         end if
10:         $J_2 = F_f^2\, F_c^2$                          ▷Compute $J_2$ and push into a vector
11:     end for
12:     $z_{\text{eff}} \leftarrow I_1^{\text{eff}}/\sqrt{3}$ , $r' \leftarrow \beta\sqrt{\frac{3K^k}{2G^k}}\sqrt{2J_2}$
13:     $\mathbf{x}_{\text{poly}}.z_{\text{eff}} \leftarrow z_{\text{eff}} \cup$ REVERSE$(z_{\text{eff}})$ , $\mathbf{x}_{\text{poly}}.r' \leftarrow r' \cup$ REVERSE$(-r')$
        ↪                    ▷Add the points on the negative $r'$ side of the polygon
14:     $\mathbf{x}_{\text{poly}}[2n_{\text{poly}} + 1] \leftarrow \mathbf{x}_{\text{poly}}[1]$            ▷Add the first point to close the polygon
15:     return $\mathbf{x}_{\text{poly}}$
16: end procedure
```

Finding the closest point on yield surface in z_{eff}-r' space

Algorithm 33 Find the closest point from the trial stress state on the polyline describing the yield surface

```
Require: $\mathbf{x}_{\text{trial}}.z_{\text{eff}}$, $\mathbf{x}_{\text{trial}}.r'$, $\mathbf{x}_{\text{poly}}$
 1: procedure FINDCLOSESTPOINT
 2:     $i \leftarrow 0$
 3:     for $\{\mathbf{x}_{\text{start}}, \mathbf{x}_{\text{end}}\}$ in $\mathbf{x}_{\text{poly}}$ do
 4:         $\mathbf{x}_{\text{seg}} \leftarrow \mathbf{x}_{\text{end}} - \mathbf{x}_{\text{start}}$
 5:         $\mathbf{x}_{\text{proj}} \leftarrow \mathbf{x}_{\text{trial}} - \mathbf{x}_{\text{start}}$
 6:         $t \leftarrow \frac{\mathbf{x}_{\text{proj}} \cdot \mathbf{x}_{\text{seg}}}{\|\mathbf{x}_{\text{seg}}\|}$
 7:         $i \leftarrow i + 1$
 8:         if $t < 0$ then
 9:             $\mathbf{x}[i] \leftarrow \mathbf{x}_{\text{start}}$
10:         else if $t > 1$ then
11:             $\mathbf{x}[i] \leftarrow \mathbf{x}_{\text{end}}$
12:         else
13:             $\mathbf{x}[i] \leftarrow \mathbf{x}_{\text{start}} + t\,\mathbf{x}_{\text{seg}}$
14:         end if
15:     end for
16:     $d^2_{\text{min}} \leftarrow$ DOUBLE_MAX
17:     $\mathbf{x}_{\text{close}} \leftarrow \mathbf{0}$
18:     for $\mathbf{x}_i$ in $\mathbf{x}$ do
19:         $d^2 \leftarrow$ DISTANCESQ($\mathbf{x}_i, \mathbf{x}_{\text{trial}}$)
20:         if $d^2 < d^2_{\text{min}}$ then
21:             $d^2_{\text{min}} \leftarrow d^2$
22:             $\mathbf{x}_{\text{close}} \leftarrow \mathbf{x}_i$
23:         end if
24:     end for
25:     return $\mathbf{x}_{\text{close}}$
26: end procedure
```

Consistency bisection algorithm

Algorithm 34 The consistency bisection algorithm for partially saturated materials

```
Require: $\delta\boldsymbol{\varepsilon}$, $\boldsymbol{\varepsilon}^{\text{p},k}$, $\boldsymbol{\sigma}^k$, $K^k$, $G^k$, $(\overline{p^w})^k$, $\phi^k$, $S_w^k$, $X^k$, $\mathbf{s}^{\text{trial}}$, $I_1^{\text{eff,trial}}$, $\sqrt{J_2^{\text{trial}}}$, $r^{\text{trial}}$, $z_{\text{eff}}^{\text{trial}}$, $\varepsilon_v^{\text{p,trial}}$, $p_3^{\text{trial}}$, $a_1$, $a_2$, $a_3$,
    $a_4$, $I_1^{\text{peak}}$, $R_c$, $\beta$, $i_{\text{max}}$, $j_{\text{max}}$, $\boldsymbol{\sigma}_{\text{fixed}}$, $\delta\boldsymbol{\varepsilon}^{\text{p}}_{\text{fixed}}$, yieldCondition
 1: procedure CONSISTENCYBISECTION
 2:     $\delta\varepsilon_v^{\text{p,fixed}} \leftarrow \text{tr}(\delta\boldsymbol{\varepsilon}^{\text{p}}_{\text{fixed}})$
 3:     $i \leftarrow 1$
 4:     $\eta^{\text{in}} \leftarrow 0$, $\eta^{\text{out}} \leftarrow 1$
 5:     while ABS($\eta^{\text{out}} - \eta^{\text{in}}$) > TOLERANCE do
 6:         $j \leftarrow 1$
 7:         isElastic ← TRUE
 8:         while isElastic = TRUE do
 9:             $\eta^{\text{mid}} \leftarrow \frac{1}{2}(\eta^{\text{in}} + \eta^{\text{out}})$
10:             $\delta\varepsilon_v^{\text{p,mid}} \leftarrow \eta^{\text{mid}}\delta\varepsilon_v^{\text{p,fixed}}$
11:             $(\overline{p^w})^{\text{mid}}$, $\phi^{\text{mid}}$, $S_w^{\text{mid}}$, $X^{\text{mid}}$ ← COMPUTEINTERNALVARIABLES($K^k$, $G^k$, $(\overline{p^w})^k$, $\phi^k$, $S_w^k$,
                    ↪ $X^k$, $\delta\varepsilon_v^{\text{p,mid}}$)        ▷Update the internal variables using the bisected increment
                    ↪ of the volumetric plastic strain
```

```
12:             isElastic ← yieldCondition.EVALYIELDCONDITION($I_1^{\text{eff,trial}}$, $\sqrt{J_2^{\text{trial}}}$, $X^{\text{mid}}$, $(\overline{p^w})^{\text{mid}}$,
                    ↪ $\phi^{\text{mid}}$, $S_w^{\text{mid}}$, $a_1$, $a_2$, $a_3$, $a_4$, $I_1^{\text{peak}}$, $R_c$, $\beta$)
                    ↪                    ▷ Determine whether the trial stress is elastic or not
13:             if isElastic = TRUE then
14:                 $\eta^{\text{out}} \leftarrow \eta^{\text{mid}}$    ▷ If the local trial state is inside the updated yield surface, the yield
                    ↪ condition evaluates to "elastic". We need to reduce the size of the
                    ↪ yield surface by decreasing the plastic strain increment.
15:                 $j \leftarrow j + 1$
16:                 if $j \geq j_{\max}$ then
17:                     return isSuccess ← FALSE    ▷ The bisection algorithm failed because of
                    ↪ too many iterations.
18:                 end if
19:             end if
20:         end while
21:         $\boldsymbol{\sigma}_{\text{fixed}}^{\text{new}}$, $\delta\boldsymbol{\varepsilon}_{\text{fixed}}^{\text{p,new}}$ ← NONHARDENINGRETURN($\boldsymbol{\sigma}^k$, $\delta\boldsymbol{\varepsilon}$, $X^{\text{mid}}$, $K^k$, $G^k$, $(\overline{p^w})^{\text{mid}}$,
                ↪ $\boldsymbol{s}^{\text{trial}}$, $\sqrt{J_2^{\text{trial}}}$, $r^{\text{trial}}$, $z_{\text{eff}}^{\text{trial}}$, $a_1$, $a_2$, $a_3$, $a_4$, $I_1^{\text{peak}}$, $R_c$, $\beta$)
                ↪                    ▷ Compute return to updated yield surface (no hardening)
22:         if SIGN($\text{tr}(\boldsymbol{\sigma}_{\text{trial}} - \boldsymbol{\sigma}_{\text{fixed}}^{\text{new}})$) ≠ SIGN($\text{tr}(\boldsymbol{\sigma}_{\text{trial}} - \boldsymbol{\sigma}_{\text{fixed}})$) then
23:             $\eta^{\text{out}} \leftarrow \eta^{\text{mid}}$                    ▷ Too much plastic strain
24:             continue
25:         end if
26:         if $\left\|\delta\boldsymbol{\varepsilon}_{\text{fixed}}^{\text{p,new}}\right\| > \eta^{\text{mid}} \left\|\delta\boldsymbol{\varepsilon}_{\text{fixed}}^{\text{p}}\right\|$ then
27:             $\eta^{\text{in}} \leftarrow \eta^{\text{mid}}$                    ▷ Too little plastic strain
28:         else
29:             $\eta^{\text{out}} \leftarrow \eta^{\text{mid}}$                    ▷ Too much plastic strain
30:         end if
31:         $i \leftarrow i + 1$
32:         if $i \geq i_{\max}$ then
33:             return isSuccess ← FALSE                    ▷ Too many iterations
34:         end if
35:     end while
36:     $\delta\varepsilon_{v,\text{fixed}}^{\text{p},k+1} \leftarrow \text{tr}(\delta\boldsymbol{\varepsilon}_{\text{fixed}}^{\text{p},k+1})$
37:     $(\overline{p^w})^{k+1}$, $\phi^{k+1}$, $S_w^{k+1}$, $X^{k+1}$ ← COMPUTEINTERNALVARIABLES($K^k$, $G^k$, $(\overline{p^w})^k$, $\phi^k$, $S_w^k$,
            ↪ $X^k$, $\delta\varepsilon_{v,\text{fixed}}^{\text{p},k+1}$)    ▷ Update the internal variables using the bisected increment
            ↪ of the volumetric plastic strain
38:     $\boldsymbol{\sigma}^{k+1} \leftarrow \boldsymbol{\sigma}_{\text{fixed}}^{\text{new}}$ , $\boldsymbol{\alpha}^{k+1} \leftarrow -(\overline{p^w})^{k+1}\boldsymbol{I}$ , $p_3^{k+1} \leftarrow p_3^{\text{trial}}$
39:     $\boldsymbol{\varepsilon}^{\text{p},k+1} = \boldsymbol{\varepsilon}^{\text{p},k} + \delta\boldsymbol{\varepsilon}^{\text{p},k+1}$                    ▷ Update the plastic strain
40:     $K^{k+1}$, $G^{k+1}$, $\boldsymbol{s}^{k+1}$, $(\overline{p^w})^{k+1}$, $I_1^{\text{eff},k+1}$, $\sqrt{J_2^{k+1}}$, $r^{k+1}$, $z_{\text{eff}}^{k+1}$, $\varepsilon_v^{\text{p},k+1}$ ←
            ↪ COMPUTEELASTICPROPERTIES($\boldsymbol{\sigma}^{k+1}$, $\phi^{k+1}$, $S_w^{k+1}$, $\boldsymbol{\varepsilon}^{\text{p},k+1}$, $\boldsymbol{\alpha}^{k+1}$, $p_3^{k+1}$)
            ↪                    ▷ Compute elastic moduli and stress invariants for the new state
41:     return isSuccess ← TRUE, $\boldsymbol{\sigma}^{k+1}$, $\boldsymbol{\varepsilon}^{\text{p},k+1}$, $\boldsymbol{\alpha}^{k+1}$, $(\overline{p^w})^{k+1}$, $\phi^{k+1}$, $S_w^{k+1}$, $X^{k+1}$, $K^{k+1}$, $G^{k+1}$,
            ↪ $\boldsymbol{s}^{k+1}$, $(\overline{p^w})^{k+1}$, $I_1^{\text{eff},k+1}$, $\sqrt{J_2^{k+1}}$, $r^{k+1}$, $z_{\text{eff}}^{k+1}$, $\varepsilon_v^{\text{p},k+1}$, $p_3^{k+1}$
42: end procedure
```

Updating the internal variables

Algorithm 35 Updating the internal variables for partially saturated materials

```
Require: $\boldsymbol{\sigma}^k$, $\varepsilon_v^{\text{p},k}$, $K^k$, $G^k$, $(\overline{p^w})^k$, $\phi^k$, $S_w^k$, $X^k$, $\delta\varepsilon_v^{\text{p}}$, fluidParams, crushParams, airModel, waterModel
 1: procedure COMPUTEINTERNALVARIABLES
 2:     $\overline{\varepsilon_v^{\text{p},k}} \leftarrow -\varepsilon_v^{\text{p},k}$, $\overline{\delta\varepsilon_v^{\text{p}}} \leftarrow -\delta\varepsilon_v^{\text{p}}$
```

```
3:      $\overline{p^w_0} \leftarrow$ fluidParams.$\overline{p^w_0}$, $S_0 \leftarrow$ fluidParams.$S_0$, $\phi_0 \leftarrow$ fluidParams.$\phi_0$, $p_1^{\text{sat}} \leftarrow$ crushParams.$p_1^{\text{sat}}$
4:      $K_a \leftarrow$ airModel.COMPUTEBULKMODULUS($(\overline{p^w})^k$)
5:      $K_w \leftarrow$ waterModel.COMPUTEBULKMODULUS($(\overline{p^w})^k$)
6:      $\varepsilon_v^{a,0} \leftarrow$ airModel.COMPUTEELASTICVOLUMETRICSTRAIN($\overline{p^w_0}$)
7:      $\varepsilon_v^{a} \leftarrow$ airModel.COMPUTEELASTICVOLUMETRICSTRAIN($(\overline{p^w})^k$)
8:      $\varepsilon_v^{w} \leftarrow$ waterModel.COMPUTEELASTICVOLUMETRICSTRAIN($(\overline{p^w})^k$, $\overline{p^w_0}$)
9:      $\overline{\varepsilon_v^a} \leftarrow -(\varepsilon_v^a - \varepsilon_v^{a,0})$, $\overline{\varepsilon_v^w} \leftarrow -\varepsilon_v^w$
10:     $\mathcal{C}_p \leftarrow S_0 \exp(\overline{\varepsilon_v^a} - \overline{\varepsilon_v^w})$
11:     $\mathcal{D}_p \leftarrow \frac{1-S_0}{(1-S_0+\mathcal{C}_p)^2}$
12:     $\dfrac{d\mathcal{C}_p}{d\overline{p^w}} \leftarrow \mathcal{C}_p \left[\frac{1}{K_a} - \frac{1}{K_w}\right]$
13:     $\mathcal{G}_a \leftarrow \exp(\overline{\varepsilon_v^{p,k}} - \overline{\varepsilon_v^a})$,  $\mathcal{G}_w \leftarrow \exp(\overline{\varepsilon_v^{p,k}} - \overline{\varepsilon_v^w})$
14:     $\mathcal{B}_p \leftarrow \frac{1}{(1-S_0)\,\mathcal{G}_a + S_0\,\mathcal{G}_w}\left[-\frac{(1-\phi^k)\phi^k}{\phi_0}\left(\frac{S_w^k}{K_w} + \frac{1-S_w^k}{K_a}\right) + \frac{1-S_0}{K_a}\,\mathcal{G}_a + \frac{S_0}{K_w}\,\mathcal{G}_w\right]$
15:     $(\overline{p^w})^{k+1} \leftarrow \text{MAX}\left[(\overline{p^w})^k + \frac{1}{\mathcal{B}_p}\,\delta\varepsilon_v^{\text{p}}, 0\right]$     ▷ Update the pore pressure making sure that pressure does
            ↪ not become negative during dilatative plastic deformations.
16:     $\overline{X}_d, \dfrac{d\overline{X}_d}{d\varepsilon_v^{\text{p}}} \leftarrow$ COMPUTEDRAINEDHYDROSTATICSTRENGTHANDDERIV($\overline{\varepsilon_v^{\text{p},k}}$)
            ↪                 ▷ Compute the drained hydrostatic compressive strength and its derivative
17:     $\overline{X}^{k+1} = -X^k + \left[\left(1 - S_w^k + p_1^{\text{sat}}\,S_w^k\right)\dfrac{d\overline{X}_d}{d\varepsilon_v^{\text{p}}} + \overline{X}_d\,(p_1^{\text{sat}} - 1)\,\frac{\mathcal{D}_p}{\mathcal{B}_p}\,\dfrac{d\mathcal{C}_p}{d\overline{p^w}} + \frac{3}{\mathcal{B}_p}\right]\delta\varepsilon_v^{\text{p}}$
            ↪                 ▷ Update the hydrostatic compressive strength
18:     $X^{k+1} \leftarrow -\overline{X}^{k+1}$
19:     $\overline{\varepsilon_v^{\text{p},k+1}} \leftarrow \overline{\varepsilon_v^{\text{p},k}} + \delta\overline{\varepsilon_v^{\text{p}}}$     ▷ Compute the updated volumetric plastic strain.
20:     $\varepsilon_v^{a,k+1} \leftarrow$ airModel.COMPUTEELASTICVOLUMETRICSTRAIN($(\overline{p^w})^{k+1}$)
21:     $\varepsilon_v^{w,k+1} \leftarrow$ waterModel.COMPUTEELASTICVOLUMETRICSTRAIN($(\overline{p^w})^{k+1}$, $\overline{p^w_0}$)
22:     $\overline{\varepsilon_v^{a,k+1}} \leftarrow -(\varepsilon_v^{a,k+1} - \varepsilon_v^{a,0})$, $\overline{\varepsilon_v^{w,k+1}} \leftarrow -\varepsilon_v^{w,k+1}$     ▷ The updated strains in the fluid phases.
23:     $\mathcal{C}_p^{k+1} \leftarrow S_0 \exp(\overline{\varepsilon_v^{a,k+1}} - \overline{\varepsilon_v^{w,k+1}})$
24:     $S_w^{k+1} \leftarrow \frac{\mathcal{C}_p^{k+1}}{1-S_0+\mathcal{C}_p^{k+1}}$     ▷ Update the saturation
25:     $\mathcal{G}_a^{k+1} \leftarrow \exp(\overline{\varepsilon_v^{\text{p},k+1}} - \overline{\varepsilon_v^{a,k+1}})$,  $\mathcal{G}_w^{k+1} \leftarrow \exp(\overline{\varepsilon_v^{\text{p},k+1}} - \overline{\varepsilon_v^{w,k+1}})$
26:     $\phi^{k+1} \leftarrow (1 - S_0)\phi_0\mathcal{G}_a^{k+1} + S_0\phi_0\mathcal{G}_w^{k+1}$     ▷ Update the porosity
27:     return $(\overline{p^w})^{k+1}$, $\phi^{k+1}$, $S_w^{k+1}$, $X^{k+1}$
28: end procedure
```

Algorithm 36 Computing the drained hydrostatic strength and its derivative

```
Require: $\overline{\varepsilon_v^{\text{p},k}}$, fluidParams, crushParams
1: procedure COMPUTEDRAINEDHYDROSTATICSTRENGTHANDDERIV
2:      $\phi_0 \leftarrow$ fluidParams.$\phi_0$
3:      $p_0 \leftarrow$ crushParams.$p_0$, $p_1 \leftarrow$ crushParams.$p_1$, $p_1^{\text{sat}} \leftarrow$ crushParams.$p_1^{\text{sat}}$, $p_2 \leftarrow$ crushParams.$p_2$
4:      $p_3 \leftarrow -\log(1 - \phi_0)$
5:      $\overline{X}_d \leftarrow \text{MAX}(p_0, 1000)$;     ▷ $\overline{X}_d$ has a minimum value of 1000 pressure units
6:      $\dfrac{d\overline{X}_d}{d\varepsilon_v^{\text{p}}} \leftarrow 0$
7:      if $\overline{\varepsilon_v^{\text{p},k}} > 0$ then
8:          $\phi_{\text{temp}} \leftarrow \exp(-p_3 + \overline{\varepsilon_v^{\text{p},k}})$
9:          $\phi \leftarrow 1 - \phi_{\text{temp}}$
```

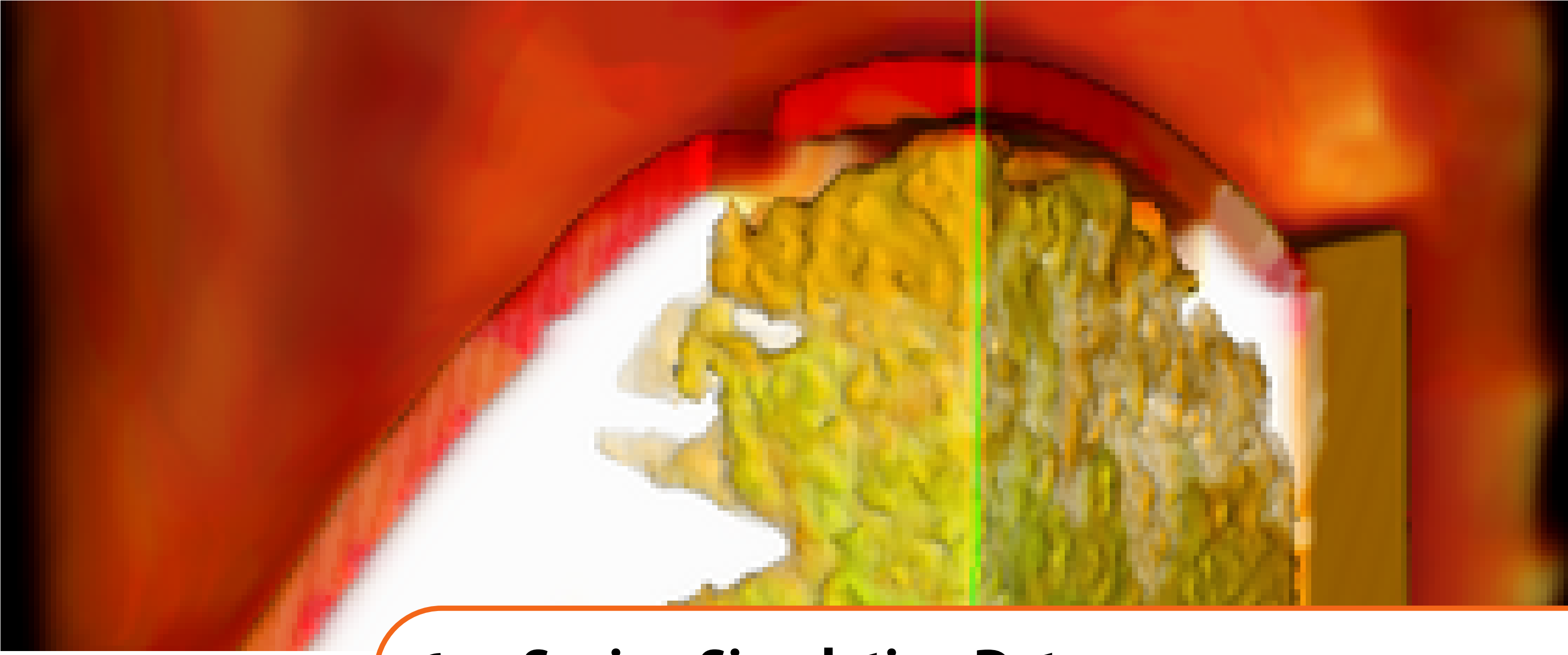
```
10:         $\bar{\xi} \leftarrow p_1 \text{ POW}\left(\frac{\phi_0}{\phi} - 1, \frac{1}{p_2}\right)$
11:         $\bar{X}_d \leftarrow \bar{X}_d + \bar{\xi}$
12:         $\frac{d\bar{X}_d}{d\varepsilon_v^{\text{p}}} \leftarrow \frac{1}{p_2}\frac{\phi_0}{\phi}\phi_{\text{temp}}\frac{\bar{\xi}}{\phi\left(\frac{\phi_0}{\phi}-1\right)}$
13:     end if
14:     return $\bar{X}_d, \frac{d\bar{X}_d}{d\varepsilon_v^{\text{p}}}$
15: end procedure
```

5.2.3 Rate-dependent plastic update

Our implementation does not consider rate-dependent updates of the internal variables. We approximate the trial stress using the average of the elastic moduli at the start and end of the step.

Algorithm 37 Computing the correction to the stress due to rate-dependent plasticity

```
Require: $\Delta t$, $\boldsymbol{d}^{n+1}$, $\boldsymbol{\sigma}^n$, $K^n$, $G^n$, $\phi^n$, $S_w^n$, $X^n$, $\boldsymbol{\alpha}^n$, $\boldsymbol{\epsilon}^{\text{p},n}$, $p_3^n$, $\boldsymbol{\sigma}_{qs}^n$, $\boldsymbol{\sigma}_{qs}^{n+1}$, $K^{n+1}$, $G^{n+1}$, $\phi^{n+1}$, $S_w^{n+1}$, $X^{n+1}$, $\boldsymbol{\alpha}^{n+1}$,
    $\boldsymbol{\epsilon}^{\text{p},n+1}$, $p_3^{n+1}$, $a_1$, $a_2$, $a_3$, $a_4$, $I_1^{\text{peak}}$, $R_c$, yieldParams
1:  procedure RATEDEPENDENTPLASTICUPDATE
2:      $T_1 \leftarrow$ yieldParams.$T_1$, $T_2 \leftarrow$ yieldParams.$T_2$
3:      if $T_1 = 0$ or $T_2 = 0$ then          ▷Check if rate-dependent plasticity has been turned on
4:          return isRateDependent ← FALSE, $\boldsymbol{\sigma}_{qs}^{n+1}$
5:      end if
6:      $K_{\text{dyn}} \leftarrow \frac{1}{2}(K^n + K^{n+1})$, $G_{\text{dyn}} \leftarrow \frac{1}{2}(G^n + G^{n+1})$          ▷Compute mid-step bulk and shear modulus
7:      $\Delta\boldsymbol{\varepsilon} \leftarrow \Delta t\, \boldsymbol{d}^{n+1}$
8:      $\boldsymbol{\sigma}_{\text{trial,dyn}} \leftarrow \text{COMPUTETRIALSTRESS}(\boldsymbol{\sigma}^n, K_{\text{dyn}}, G_{\text{dyn}}, \boldsymbol{d}^{n+1}, \Delta t)$          ▷Compute substep trial stress
9:      $\dot{\varepsilon} \leftarrow \text{MAX}(\|\boldsymbol{d}^{n+1}\|, \text{ABS\_DOUBLE\_MIN})$
10:     $\tau \leftarrow T_1 \text{ POW}(\dot{\varepsilon}, T_2)$          ▷The characteristic time is defined from the rate-dependence
            ↪ input parameters and the magnitude of the strain rate
11:     $r_h \leftarrow \exp\left(-\frac{\Delta t}{\tau}\right)$
12:     $R_H \leftarrow \frac{1-r_h}{\frac{\Delta t}{\tau}}$
13:     $\boldsymbol{\sigma}^{n+1} \leftarrow \boldsymbol{\sigma}_{qs}^{n+1} + \left[(\boldsymbol{\sigma}_{\text{trial,dyn}} - \boldsymbol{\sigma}^n) - (\boldsymbol{\sigma}_{qs}^{n+1} - \boldsymbol{\sigma}_{qs}^n)\right] R_H + (\boldsymbol{\sigma}^n - \boldsymbol{\sigma}_{qs}^n)\, r_h$          ▷Stress update
14:     return $\boldsymbol{\sigma}^{n+1}$, isRateDependent ← TRUE
15: end procedure
```



6 — Saving Simulation Data

The UCF automatically saves data as specified in the .ups file. This works for all data that is stored in the Data Warehouse. However, there are times when a component will need to save its own data. (It is preferable that the Data Warehouse be updated to manage this Component data, but sometimes this is not expedient...) In these cases, here is an outline on how to save that data:

- Create a function that will save your data:

```
void
Component::saveMyData( const PatchSubset * patches )
{
  ...
}
```

- From the last Task in your algorithm, call the function:

```
saveMyData( patches );
```

- In the function, you need to do several things: Make sure that it is an output timestep. (Your component must have saved a pointer to the Data Archiver. Most components already do this... if not, you can do it in Component::problemSetup() .)

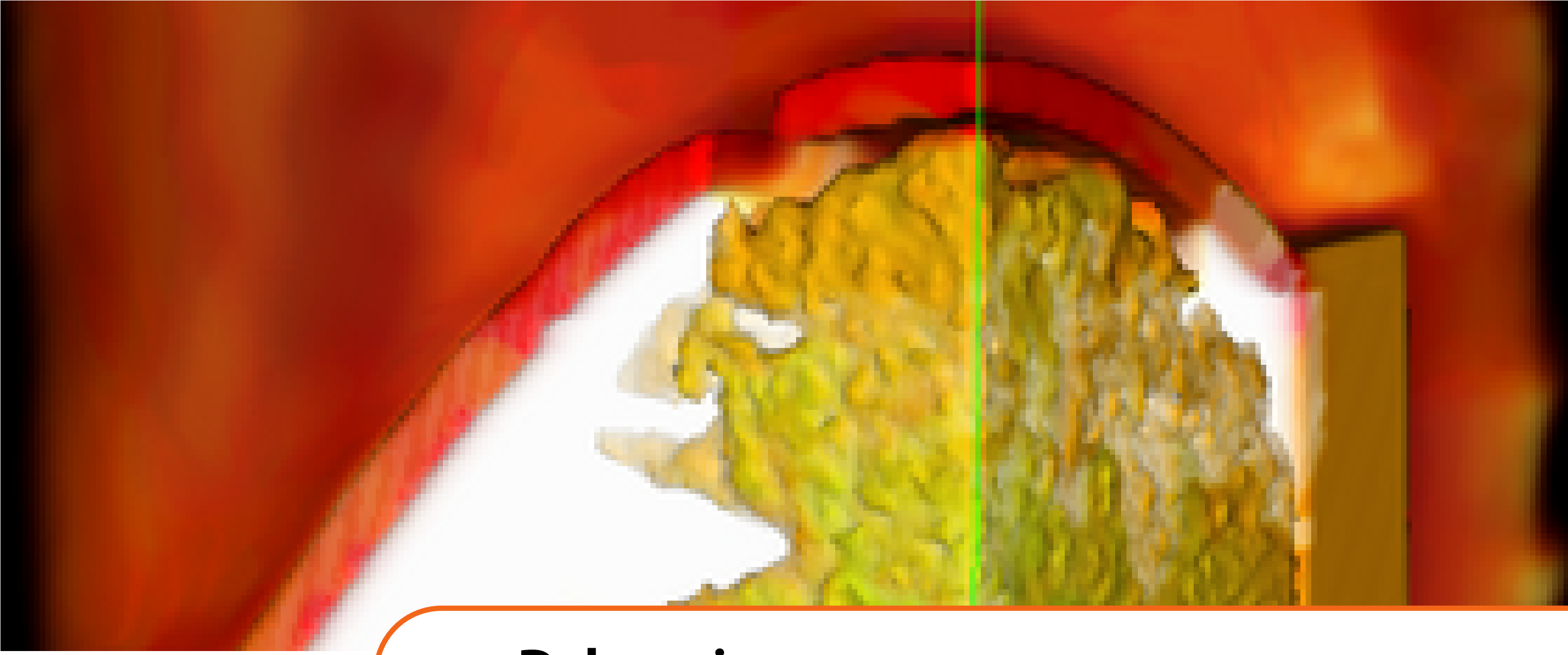
```
if( dataArchiver_->isOutputTimestep() ) {
  // Dump your data... (see below)
}
```

- You need to create a directory to put the data in:

```
const int      & timestep = d_sharedState->getCurrentTopLevelTimeStep();
ostringstream   data_dir;

// This gives you ".../simulation.uda.###/t#####/mydata/
data_dir << dataArchiver()->getOutputLocation() << "/t" << setw(5) <<
    setfill('0') << timestep
<< "/mydata";

int result = MKDIR( data_dir.c_str(), 0777 );  // Make sure you #
    include <Core/OS/Dir.h>

if( result != 0 ) {
  ostringstream error;
  error << "Component::saveMyData(): couldn't create directory '") +
      data_dir.str() + "' (" << result << ").";
    throw InternalError( error.str(), __FILE__, __LINE__);
  }
```

- Loop over the patches and save your data... (Note, depending on how much data you have, you need to determine whether you want to write the files as a binary file, or as an ascii file. Ascii files are easier for a human to look at for errors, but take a lot more time to read and space to write.)

```
  for( int pIndex = 0; pIndex < patches->size(); pIndex++ ){
    const Patch * patch = patches->get( pIndex );
    ostringstream file_name;
    // WARNING... not sure this is the correct naming convention...
       PLEASE VERIFY AND UPDATE THIS DOC.
    file_name << data_dir << "/p" << setw(5) << setfill('0') << pIndex;

    ofstream output( ceFileName, ios::out | ios::binary);
    if( !output ) {
      throw InternalError( string( "Component::saveMyData(): couldn't
          open file '") + file_name.str() + "'.",
      __FILE__, __LINE__);
    }

    // Output data
    for( loop over data ) {
      output << data...
    }

    output.close();
  } // end for pIndex
```



7 — Debugging

Debugging multi-processor software can be very difficult. Below are some hints on how to approach this.

7.1 Debugger

Use a debugger such as GDB to attach to the running program. Note, if you are an Emacs user, running GDB through Emacs' gdb-mode makes debugging even easier!

7.1.1 Serial Debugging

Whenever possible, debug vaango running in serial (ie. no MPI). This is much easier for many reasons (though may also not be possible in many situations). To debug serially, use:

```
gdb vaango <input_file>
```

7.1.2 Parallel Debugging

If it is necessary to use a parallel run of vaango to debug, a few things must take place.

1) The helpful macro `WAIT_FOR_DEBUGGER();` must be inserted into the code (vaango.cc) just before the real execution of the components begins ctl->run():

```
WAIT_FOR_DEBUGGER();
ctl->run();
```

Recompile the code (will only take a few seconds), then run through MPI as normal.

Caveats

1. Unless you know that the error occurs on a specific processor, you will need to attach a debugger to every process... so it behooves the developer to narrow the problem down to as few processors as possible.
2. If you did know the processor, then the `WAIT_FOR_DEBUGGER()` will need to be placed inside an 'if' statement to check for the correct MPI Rank.
3. Try to use only one node. If you need more 'processors', you (most likely) can double up on the same machine. If vaango must span multiple nodes, then you will need to log into each node separately to attach debuggers (see below).

7.1.3 Running Vaango

Run vaango :

```
mpirun -np 8 -m mpihosts vaango inputs/ARCHES/helium_1m.ups
```

Create a new terminal window for each processor you are using. (In the above case, 8 windows.)

You will notice that the output from vaango will include lines (one for each processor used) like this:

```
updraft2:22793 waiting for debugger
```

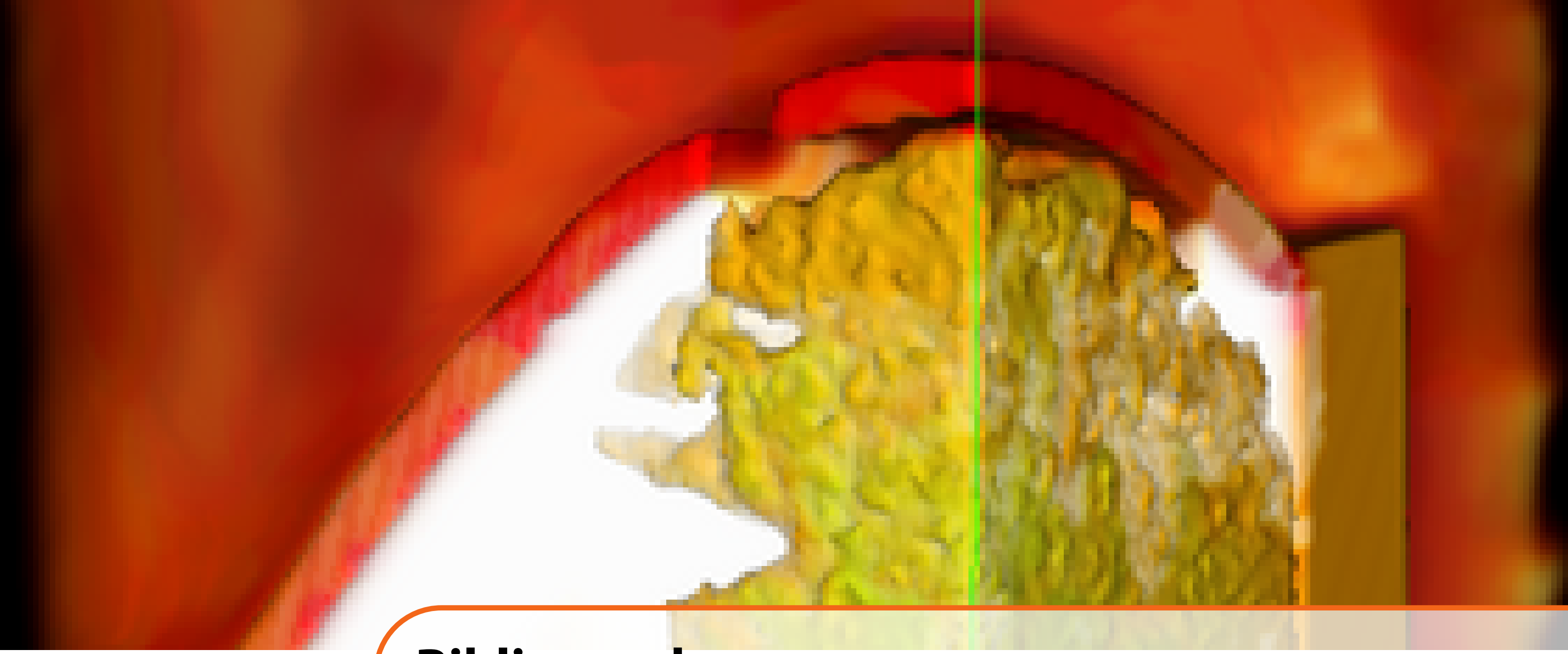
Attach to each process in the following manner (where PID is the process id number (22793 in the above case)):

```
cd Uintah/bin/StandAlone
gdb -p <PID>
```

Within GDB, you will then need to break each process out of the WAIT loop in the following manner:

```
up 2
set wait=false
cont
```

The above GDB commands must be run from each GDB session. Once all the WAITs are stopped, vaango will begin to run.



Bibliography

[HGB15] M. A. Homel, J. E. Guilkey, and R. M. Brannon. "Continuum effective-stress approach for high-rate plastic deformation of fluid-saturated geomaterials with application to shaped-charge jet penetration". In: *Acta Mechanica* (2015), pages 1–32 (cited on pages 43, 44).