

Vaango Developers Manual

Version 17.09

September 25, 2017

The Utah Uintah team

and

Biswajit Banerjee



Copyright © 2015-2017 Parresia Research Limited

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



Contents

1	The Vaango framework	7
1.1	Historical information	7
1.2	Overview	7
1.3	An example	8
2	General concepts	11
2.1	Scheduler	11
2.1.1	needRecompile()	11
2.2	Tasks	11
2.3	Simulation Component Class Description	12
2.3.1	ProblemSetup	13
2.3.2	ScheduleInitialize	13
2.3.3	ScheduleComputeStableTimestep	13
2.3.4	ScheduleTimeAdvance	13
2.4	Data Storage Concepts	13
2.4.1	Data Archiver	13
2.4.2	Two Data Warehouses	13
2.4.3	Storing and Retrieving Variables	14
2.4.4	Variables	14
2.5	Grid data	15
2.5.1	Cells	15
2.5.2	Patches	15
2.5.3	Boundary Cells	15
2.5.4	Indexing grid cells	16
3	PDE Examples	17
3.1	Poisson1	17
3.1.1	Description of Scheduling Functions	18

3.1.2	Description of Computational Functions	20
3.1.3	Input file	21
3.2	Poisson2	23
3.3	Burger	25
4	The MPM component	29
4.1	The MPM algorithm	29
4.2	Reading the input file	30
4.3	Problem setup	30
4.4	Initialization	31
4.5	Time advance	31
4.5.1	Computing the body force	32
4.5.2	Applying external loads	32
4.5.3	Interpolating particles to grid	33
4.5.4	Exchanging momentum using interpolated grid values	34
4.5.5	Computing the internal force	34
4.5.6	Computing and integrating the acceleration	35
4.5.7	Exchanging momentum using integrated grid values	35
4.5.8	Setting grid boundary conditions	35
4.5.9	Computing the deformation gradient	35
4.5.10	Computing the stress tensor	36
4.5.11	Computing the basic damage parameter	36
4.5.12	Updating the particle erosion parameter	37
4.5.13	Interpolating back to the particles and update	37
5	Example MPM material model	39
5.1	Initialization of the model	39
5.2	Computing the stress and internal variables	40
5.2.1	Compute elastic properties	41
5.2.2	Rate-independent stress update	43
5.2.3	Rate-dependent plastic update	51
6	Submodels of MPM material models	53
6.1	Models in the PlasticityModels directory	53
6.1.1	Damage models	54
6.1.2	Deviatoric stress models	54
6.1.3	Flow stress models	54
6.1.4	Kinematic hardening models	54
6.1.5	Melting temperature models	54
6.1.6	Equation of state models	54
6.1.7	Shear modulus models	55
6.1.8	Specific heat models	55
6.1.9	Stability check models	55
6.1.10	Viscoplasticity models	55
6.1.11	Yield condition models	55
6.2	Using the models in PlasticityModels	55
6.3	Creating a new model in PlasticityModels	57

7	Saving Simulation Data	65
8	Debugging	67
8.1	Debugger	67
8.1.1	Serial Debugging	67
8.1.2	Parallel Debugging	67
8.1.3	Running Vaango	68
	Bibliography	69



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 original 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 course 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.

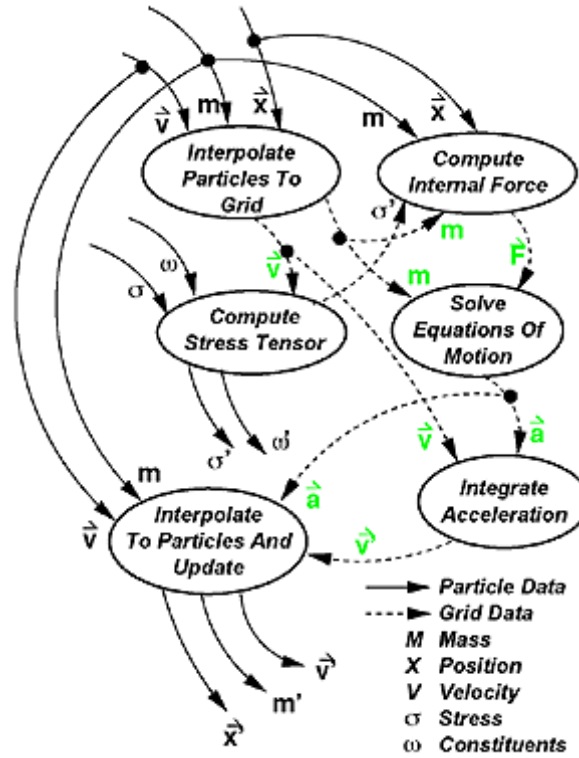


Figure 1.1: Example Task Graph

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 = scnew 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 = scineu 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 execute 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 func-

tion 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.

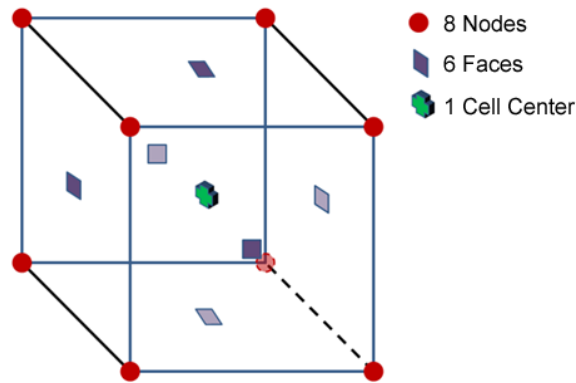


Figure 2.1: Variable locations with respect to a single cell.

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).

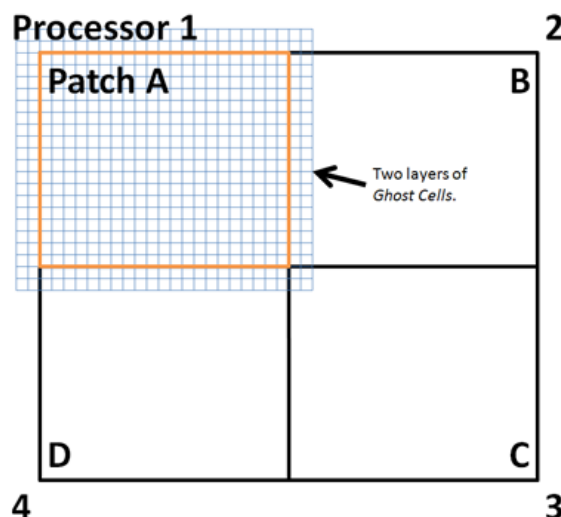


Figure 2.2: Schematic of ghost cell locations laid out with respect to patches.

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 transferred (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 instead 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, 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 readability 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 warehouse 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, `computeStableTimestep` 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 warehouse 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:

$$\frac{du}{dt} = -u \frac{du}{dx} \quad (3.1)$$

with the initial conditions:

$$u = \sin(\pi x) + \sin(2\pi y) + \sin(3\pi z) \quad (3.2)$$

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 Poisson's `timeAdvance` routine. The relevant 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 `Poisson` 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(1, 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

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

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 used 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.
- \mathbf{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 $t = t_{n+1}$.
- $\mathbf{F}_p^n, \mathbf{F}_p^{n+1}$: The particle deformation gradient at time $t = t_n$ and $t = t_{n+1}$.

Algorithm 1 The algorithm

- 1: **procedure** RUN(inputUPSTFile)
- 2: $t_{\max}, \mathbf{h}_g, \text{xmlProblemSpec}, \text{grid}, \text{globalState} \leftarrow \text{READINPUTUPSTFile}(\text{inputUPSTFile})$ ▷Parse
 ↪ the input XML file (*filename.upst*), 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:   $\mathbf{x}_p^n, \mathbf{u}_p^n, m_p, V_p^n, \mathbf{v}_p^n, \boldsymbol{\sigma}_p^n, \mathbf{F}_p^n \leftarrow \text{INITIALIZE}(\text{xmlProblemSpec})$       ▷Find the grid size and initialize
    ↪ particle variables based on geometry and other information in the input file.
6:  isSuccess ← FALSE
7:  repeat
8:     $\Delta t \leftarrow \text{COMPUTE\_STABLE\_TIME\_STEP}(\mathbf{h}_g, \mathbf{v}_p)$       ▷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,  $\mathbf{x}_p^{n+1}, \mathbf{u}_p^{n+1}, V_p^{n+1}, \mathbf{v}_p^{n+1}, \boldsymbol{\sigma}_p^{n+1}, \mathbf{F}_p^{n+1} \leftarrow \text{TIMEADVANCE}(\mathbf{h}_g, \mathbf{x}_p^n, \mathbf{u}_p^n, m_p, V_p^n,$ 
    ↪  $\mathbf{v}_p^n, \boldsymbol{\sigma}_p^n, \mathbf{F}_p^n)$       ▷Compute updated quantities
11:   OUTPUTDATA( $\mathbf{x}_p^{n+1}, \mathbf{u}_p^{n+1}, V_p^{n+1}, \mathbf{v}_p^{n+1}, \boldsymbol{\sigma}_p^{n+1}, \mathbf{F}_p^{n+1}$ )      ▷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:      $n_p[\text{matl}], \mathbf{x}_p^o[\text{matl}], \mathbf{u}_p^o[\text{matl}], m_p[\text{matl}], V_p^o[\text{matl}], \mathbf{v}_p^o[\text{matl}], \mathbf{b}_p^o[\text{matl}],$   

       ↪  $\mathbf{f}_p^{\text{ext},o}[\text{matl}] \leftarrow \text{matl.CREATEPARTICLES}()$ 
4:      $\mathbf{F}_p^o[\text{matl}] \leftarrow \text{defGradComputer.INITIALIZE}(\text{matl})$ 
5:      $\boldsymbol{\sigma}_p^o[\text{matl}] \leftarrow \text{constitutiveModel.INITIALIZE}(\text{matl})$ 
6:      $D_p^o[\text{matl}] \leftarrow \text{damageModel.INITIALIZE}(\text{matl})$ 
7:   end for
8:   if mpmFlags.initializeStressWithBodyForce = TRUE then
9:      $\mathbf{b}_p^o \leftarrow \text{INITIALIZEBODYFORCE}()$ 
10:     $\boldsymbol{\sigma}_p^o, \mathbf{F}_p^o \leftarrow \text{INITIALIZESTRESSANDDEFGRADFROMBODYFORCE}()$ 
11:   end if
12:   if mpmFlags.applyParticleBCs = TRUE then
13:      $\mathbf{f}_p^{\text{ext},o} \leftarrow \text{particleBC.INITIALIZEPRESSUREBCs}()$ 
14:   end if
15:   return  $n_p, \mathbf{x}_p^o, \mathbf{u}_p^o, m_p, V_p^o, \mathbf{v}_p^o, \mathbf{b}_p^o, \mathbf{f}_p^{\text{ext},o}, \mathbf{F}_p^o, \boldsymbol{\sigma}_p^o, D_p^o$ 
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( $\mathbf{h}_g, \mathbf{x}_p^n, \mathbf{u}_p^n, m_p, V_p^n, \mathbf{v}_p^n, \mathbf{f}_p^{\text{ext},n}, \mathbf{d}_p^n$ )
2:    $\mathbf{b}_p^n \leftarrow \text{COMPUTEPARTICLEBODYFORCE}()$  ▷ Compute the body force term
3:    $\mathbf{f}_p^{\text{ext},n+1} \leftarrow \text{APPLYEXTERNALLOADS}()$  ▷ Apply external loads to the particles
4:    $m_g, V_g, \mathbf{v}_g, \mathbf{b}_g, \mathbf{f}_g^{\text{ext}} \leftarrow \text{INTERPOLATEPARTICLES TO GRID}()$  ▷ Interpolate particle data to the grid
5:   EXCHANGEMOMENTUMINTERPOLATED() ▷ Exchange momentum between bodies on grid.  

   ↪ Not discussed in this report.
6:    $\mathbf{f}_g^{\text{int}}, \boldsymbol{\sigma}_g, \mathbf{v}_g \leftarrow \text{COMPUTEINTERNALFORCE}()$  ▷ Compute the internal force at the grid nodes
7:    $\mathbf{v}_g^*, \mathbf{a}_g \leftarrow \text{COMPUTEANDINTEGRATEACCELERATION}()$  ▷ Compute the grid velocity  

   ↪ and grid acceleration
8:   EXCHANGEMOMENTUMINTEGRATED() ▷ Exchange momentum between bodies on grid  

   ↪ using integrated values. Not discussed in this report.

```

```

9:    $\mathbf{v}_g^*, \mathbf{a}_g \leftarrow \text{SETGRIDBOUNDARYCONDITIONS}()$  ▷ Update the grid velocity and grid
    ↪ acceleration using the BCs
10:   $\mathbf{l}_p^n, \mathbf{F}_p^{n+1}, \mathbf{V}_p^{n+1} \leftarrow \text{COMPUTEDEFORMATIONGRADIENT}()$  ▷ Compute the velocity gradient
    ↪ and the deformation gradient
11:   $\boldsymbol{\sigma}_p^{n+1}, \boldsymbol{\eta}_p^{n+1} \leftarrow \text{COMPUTESTRESSTENSOR}()$  ▷ Compute the updated stress and
    ↪ internal variables (if any)
12:   $\boldsymbol{\sigma}_p^{n+1}, \boldsymbol{\eta}_p^{n+1}, \chi_p^{n+1}, D_p^{n+1} \leftarrow \text{COMPUTE BASIC DAMAGE}()$  ▷ Compute the damage parameter
    ↪ and update the stress and internal variables
13:   $\chi_p^{n+1}, D_p^{n+1} \leftarrow \text{UPDATEEROSIONPARAMETER}()$  ▷ Update the indicator variable that is used
    ↪ to delete particles at the end of a time step
14:   $\mathbf{V}_p^{n+1}, \mathbf{u}_p^{n+1}, \mathbf{v}_p^{n+1}, \mathbf{x}_p^{n+1}, m_p, \mathbf{h}_p^{n+1} \leftarrow \text{INTERPOLATE TOPARTICLES AND UPDATE}()$  ▷ 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: $\mathbf{x}_p^n, \mathbf{v}_p^n$, `materialList`, `particleList`, `mpmFlags`

```

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

- \mathbf{h}_p^n : The particle size matrix at time $t = t_n$.

Algorithm 6 Applying external loads to particles

Require: t_{n+1} , \mathbf{x}_p^n , \mathbf{h}_p^n , \mathbf{u}_p^n , $\mathbf{f}_p^{\text{ext},n}$, \mathbf{F}_p^n , `materialList`, `particleList`, `mpmFlags`, `particleBC`

```

1: procedure APPLYEXTERNALLOADS
2:    $f_p \leftarrow 0$ 
3:   if mpmFlags.useLoadCurves = TRUE then
4:      $f_p \leftarrow \text{particleBC.COMPUTEFORCEPERPARTICLE}(t_{n+1})$  ▷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:          $\mathbf{f}_p^{\text{ext},n+1}[\text{matl},\text{part}] \leftarrow \text{particleBC.GETFORCEVECTOR}(t_{n+1}, \mathbf{x}_p^n, \mathbf{h}_p^n, \mathbf{u}_p^n,$ 
           ↪  $f_p, \mathbf{F}_p^n$ ) ▷Compute the applied force vector at each particle
10:      end for
11:    else
12:       $\mathbf{f}_p^{\text{ext},n+1}[\text{matl}] \leftarrow \mathbf{f}_p^{\text{ext},n}[\text{matl}]$ 
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 , \mathbf{h}_p^n , \mathbf{b}_p^n , $\mathbf{f}_p^{\text{ext},n+1}$, \mathbf{F}_p^n , `materialList`, `particleList`, `gridNodeList`, `mpmFlags`, `particleBC`

```

1: procedure INTERPOLATEPARTICLESTOGRID
2:   interpolator  $\leftarrow \text{CREATEINTERPOLATOR}(\text{mpmFlags})$  ▷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:        $n_{gp}, S_{gp} \leftarrow \text{interpolator.FINDCELLSANDWEIGHTS}(\mathbf{x}_p^n, \mathbf{h}_p^n, \mathbf{F}_p^n)$  ▷Find the node
         ↪ indices of the cells affecting the particle and the interpolation weights
6:        $\mathbf{p}_p \leftarrow m_p[\text{matl}][\text{part}] \mathbf{v}_p^n[\text{matl}][\text{part}]$  ▷Compute particle momentum
7:       for node in  $n_{gp}$  do
8:          $m_g[\text{matl}][\text{node}] \leftarrow m_g[\text{matl}][\text{node}] + m_p[\text{matl}][\text{part}] S_{gp}[\text{node}]$ 
9:          $V_g[\text{matl}][\text{node}] \leftarrow V_g[\text{matl}][\text{node}] + V_p^n[\text{matl}][\text{part}] S_{gp}[\text{node}]$ 
10:         $\mathbf{v}_g[\text{matl}][\text{node}] \leftarrow \mathbf{v}_g[\text{matl}][\text{node}] + \mathbf{p}_p S_{gp}[\text{node}]$ 
11:         $\mathbf{f}_g^{\text{ext}}[\text{matl}][\text{node}] \leftarrow \mathbf{f}_g^{\text{ext}}[\text{matl}][\text{node}] + \mathbf{f}_p^{\text{ext},n+1}[\text{matl}][\text{part}] S_{gp}[\text{node}]$ 
12:         $\mathbf{b}_g[\text{node}] \leftarrow \mathbf{b}_g[\text{node}] + m_p[\text{matl}][\text{part}] \mathbf{b}_p^n[\text{matl}][\text{part}] S_{gp}[\text{node}]$ 
13:      end for
14:    end for

```

```

15:   for node in gridNodeList do
16:      $\mathbf{v}_g[\text{matl}][\text{node}] \leftarrow \mathbf{v}_g[\text{matl}][\text{node}]/m_g[\text{matl}][\text{node}]$ 
17:   end for
18:    $\mathbf{v}_g[\text{matl}] \leftarrow \text{APPLYSYMMETRYVELOCITYBC}(\mathbf{v}_g[\text{matl}])$  ▷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
- \mathbf{G}_{gp} : The gradient of the nodal interpolation function evaluated at a particle
- σ_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: $h_g, V_g, V_p^n, \mathbf{x}_p^n, \mathbf{h}_p^n, \sigma_p^n, \mathbf{F}_p^n, \text{materialList}, \text{particleList}, \text{gridNodeList}$ mpmFlags

```

1: procedure COMPUTEINTERNALFORCE
2:   interpolator ← CREATEINTERPOLATOR( $\text{mpmFlags}$ ) ▷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:        $n_{gp}, S_{gp}, \mathbf{G}_{gp} \leftarrow$ 
   ↪ interpolator.FINDCELLSANDWEIGHTSANDSHAPEDERIVATIVES( $\mathbf{x}_p^n, \mathbf{h}_p^n, \mathbf{F}_p^n$ )
   ↪ ▷Find the node indices of the cells affecting the particle and
   ↪ the interpolation weights and gradients
6:        $\sigma_v \leftarrow V_p[\text{matl}][\text{part}] \sigma_p^n[\text{matl}][\text{part}]$ 
7:       for node in  $n_{gp}$  do
8:          $\mathbf{f}_g^{\text{int}}[\text{matl}][\text{node}] \leftarrow \mathbf{f}_g^{\text{int}}[\text{matl}][\text{node}] - (\mathbf{G}_{gp}[\text{node}]/h_g) \cdot \sigma_p^n[\text{matl}][\text{part}] V_p^n[\text{part}]$ 
9:          $\sigma_g[\text{matl}][\text{node}] \leftarrow \sigma_g[\text{matl}][\text{node}] + \sigma_v S_{gp}[\text{node}]$ 
10:      end for
11:    end for
12:    for node in gridNodeList do
13:       $\sigma_g[\text{matl}][\text{node}] \leftarrow \sigma_g[\text{matl}][\text{node}]/V_g[\text{matl}][\text{node}]$ 
14:    end for
15:     $\mathbf{v}_g[\text{matl}] \leftarrow \text{APPLYSYMMETRYTRACTIONBC}()$  ▷Apply any symmetry tractions BCs
   ↪ that may be applicable
16:  end for
17:  return  $\mathbf{f}_g^{\text{int}}, \sigma_g, \mathbf{v}_g$ 
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^* : The integrated grid velocities.

Algorithm 9 Computing and integrating the acceleration

Require: $\Delta t, m_g, \mathbf{f}_g^{\text{int}}, \mathbf{f}_g^{\text{ext}}, \mathbf{b}_g, \mathbf{v}_g, \text{materialList}, \text{gridNodeList}, \text{mpmFlags}$

```

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

```

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

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

Algorithm 11 Computing the velocity gradient and deformation gradient

Require: $\Delta t, \mathbf{x}_p^n, m_p, V_p^n, \mathbf{h}_p^n, \mathbf{v}_p^n, \mathbf{l}_p^n, \mathbf{F}_p^n, \mathbf{h}_g, \mathbf{v}_g, \mathbf{v}_g^*, \rho_o, \text{materialList}, \text{gridNodeList}, \text{mpmFlags}, \text{velGradComputer}$

```

1: procedure COMPUTEDEFORMATIONGRADIENT
2:    $\text{interpolator} \leftarrow \text{CREATEINTERPOLATOR}(\text{mpmFlags})$ 
```

```

3:   for matl in materialList do
4:     for part in particleList do
5:        $l_p^{n+1}[\text{matl}, \text{part}] \leftarrow \text{velGradComputer.COMPUTEVELGRAD}(\text{interpolator}, h_g, x_p^n[\text{matl}, \text{part}],$ 
         $\hookrightarrow h_p^n[\text{matl}, \text{part}], F_p^n[\text{matl}, \text{part}], v_g^*[\text{matl}])$   $\triangleright$  Compute the velocity gradient
6:        $F_p^{n+1}[\text{matl}, \text{part}], \Delta F_p^{n+1} \leftarrow \text{COMPUTEDEFORMATIONGRADIENTFROMVELOCITY}(l_p^n[\text{matl}, \text{part}],$ 
         $\hookrightarrow l_p^{n+1}[\text{matl}, \text{part}], F_p^n[\text{matl}, \text{part}])$   $\triangleright$  Compute the deformation gradient
7:        $V_p^{n+1}[\text{matl}, \text{part}] \leftarrow m_p[\text{matl}, \text{part}] / \rho_0 * \det(F_p^{n+1}[\text{matl}, \text{part}])$ 
8:     end for
9:   end for
10:  return  $l_p^{n+1}, F_p^{n+1}, V_p^{n+1}$ 
11: end procedure

```

Algorithm 12 Computing the deformation gradient using the velocity gradient

Require: $\Delta t, l_p^{n+1}, F_p^n, \text{mpmFlags}$

```

1: procedure COMPUTEDEFORMATIONGRADIENTFROMVELOCITY
2:   if mpmFlags.defGradAlgorithm = "first_order" then
3:      $F_p^{n+1}, \Delta F_p^{n+1} \leftarrow \text{SERIESUPDATECONSTANTVELGRAD}(\text{numTerms} = 1, \Delta t, l_p^{n+1}, F_p^n)$ 
4:   else if mpmFlags.defGradAlgorithm = "subcycle" then
5:      $F_p^{n+1}, \Delta F_p^{n+1} \leftarrow \text{SUBCYCLEUPDATECONSTANTVELGRAD}(\Delta t, l_p^{n+1}, F_p^n)$ 
6:   else if mpmFlags.defGradAlgorithm = "taylor_series" then
7:      $F_p^{n+1}, \Delta F_p^{n+1} \leftarrow \text{SERIESUPDATECONSTANTVELGRAD}(\text{numTerms} = \text{mpmFlags.numTaylorSeriesTerms},$ 
       $\Delta t, l_p^{n+1}, F_p^n)$ 
8:   else
9:      $F_p^{n+1}, \Delta F_p^{n+1} \leftarrow \text{CAYLEYUPDATECONSTANTVELGRAD}(\Delta t, l_p^{n+1}, F_p^n)$ 
10:  end if
11:  return  $F_p^{n+1}, \Delta F_p^{n+1}$ 
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

- η_p^n, η_p^{n+1} : The internal variables needed by the constitutive model.

Algorithm 13 Computing the stress tensor

Require: $\Delta t, x_p^n, m_p, V_p^{n+1}, h_p^n, l_p^{n+1}, F_p^{n+1}, \sigma_p^n, \eta_p^n, \rho_0, \text{materialList}, \text{mpmFlags}, \text{constitutiveModel}$

```

1: procedure COMPUTESTRESSTENSOR
2:   for matl in materialList do
3:      $\sigma_p^{n+1}, \eta_p^{n+1} \leftarrow \text{constitutiveModel}[\text{matl}].\text{COMPUTESTRESSTENSOR}(\Delta t, x_p^n, m_p, V_p^{n+1}, h_p^n,$ 
       $\hookrightarrow l_p^{n+1}, F_p^{n+1}, \sigma_p^n, \eta_p^n, \rho_0, \text{mpmFlags})$   $\triangleright$  Update the stress and any
       $\hookrightarrow$  internal variables needed by the constitutive model
4:   end for
5:   return  $\sigma_p^{n+1}, \eta_p^{n+1}$ 
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

- $\epsilon_p^{f,n}, \epsilon_p^{f,n+1}$: 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: $t^{n+1}, V_p^{n+1}, F_p^{n+1}, \sigma_p^{n+1}, D_p^n, \epsilon_p^{f,n}, \chi_p^n, t_p^{\chi,n}, \text{materialList}, \text{mpmFlags}$

```

1: procedure COMPUTEDAMAGE
2:   for  $\text{matl}$  in  $\text{materialList}$  do
3:     for  $\text{part}$  in  $\text{particleList}$  do
4:       if brittleDamage = TRUE then
5:          $\sigma_p^{n+1}, \epsilon_p^{f,n+1}, \chi_p^{n+1}, t_p^{\chi,n+1}, D_p^{n+1} \leftarrow \text{UPDATEDAMAGEANDMODIFYSTRESS}(V_p^{n+1}, F_p^{n+1},$ 
            $\hookrightarrow \sigma_p^{n+1}, D_p^n, \epsilon_p^{f,n}, \chi_p^n, t_p^{\chi,n})$   $\triangleright$ Update the damage parameters and stress
6:       else
7:          $\sigma_p^{n+1}, \epsilon_p^{f,n+1}, \chi_p^{n+1}, t_p^{\chi,n+1} \leftarrow \text{UPDATEFAILEDPARTICLESANDMODIFYSTRESS}(V_p^{n+1}, F_p^{n+1},$ 
            $\hookrightarrow \sigma_p^{n+1}, \epsilon_p^{f,n}, \chi_p^n, t_p^{\chi,n}, t^{n+1})$   $\triangleright$ Update the failed particles and stress
8:       end if
9:     end for
10:  end for
11:  return  $\sigma_p^{n+1}, \epsilon_p^{f,n+1}, \chi_p^{n+1}, t_p^{\chi,n+1}, D_p^{n+1}$ 
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: $D_p^n, \chi_p^n, \text{materialList}, \text{mpmFlags}, \text{constitutiveModel}$

```

1: procedure UPDATEEROSIONPARAMETER
2:   for  $\text{matl}$  in  $\text{materialList}$  do
3:     for  $\text{part}$  in  $\text{particleList}$  do
4:       if  $\text{matl.doBasicDamage} = \text{TRUE}$  then
5:          $\chi_p^{n+1} \leftarrow \text{damageModel.GETLOCALIZATIONPARAMETER}()$   $\triangleright$ Just get the indicator
            $\hookrightarrow$  parameter for particles that will be eroded.
6:       else
7:          $\chi_p^{n+1}, D_p^{n+1} \leftarrow \text{constitutiveModel}[\text{matl}].\text{GETDAMAGEPARAMETER}(\chi_p^n, D_p^n)$ 
            $\hookrightarrow$   $\triangleright$ 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: Δt , \mathbf{a}_g , \mathbf{v}_g^* , \mathbf{x}_p^n , \mathbf{v}_p^n , \mathbf{u}_p^n , \mathbf{h}_p^n , χ_p^{n+1} , \mathbf{F}_p^{n+1} , V_p^{n+1} , `materialList`, `particleList`, `gridNodeList`, `mpmFlags`

```

1: procedure INTERPOLATEToPARTICLESANDUPDATE
2:   interpolator  $\leftarrow$  CREATEINTERPOLATOR(mpmFlags)
3:   for matl in materialList do
4:      $\mathbf{h}_p^{n+1} \leftarrow \mathbf{h}_p^n$ 
5:     for part in particleList do
6:        $n_{gp}, S_{gp} \leftarrow \text{interpolator.FINDCELLSANDWEIGHTS}(\mathbf{x}_p^n, \mathbf{h}_p^{n+1}, \mathbf{F}_p^{n+1})$ 
7:        $\mathbf{v} \leftarrow \mathbf{0}$ ,  $\mathbf{a} \leftarrow \mathbf{0}$ ,
8:       for node in gridNodeList do
9:          $\mathbf{v} \leftarrow \mathbf{v} + \mathbf{v}_g^*[\text{node}] * S_{gp}[\text{node}]$   $\triangleright$  Update particle velocity
10:         $\mathbf{a} \leftarrow \mathbf{a} + \mathbf{a}_g[\text{node}] * S_{gp}[\text{node}]$   $\triangleright$  Update particle acceleration
11:      end for
12:       $\mathbf{x}_p^{n+1} \leftarrow \mathbf{x}_p^n + \mathbf{v} * \Delta t$   $\triangleright$  Update position
13:       $\mathbf{u}_p^{n+1} \leftarrow \mathbf{u}_p^n + \mathbf{v} * \Delta t$   $\triangleright$  Update displacement
14:       $\mathbf{v}_p^{n+1} \leftarrow \mathbf{v}_p^n + \mathbf{a} * \Delta t$   $\triangleright$  Update velocity
15:    end for
16:  end for
17:  DELETEROGUEPARTICLES()  $\triangleright$  Delete particles that are to be eroded.
18:  return  $V_p^{n+1}$ ,  $\mathbf{u}_p^{n+1}$ ,  $\mathbf{v}_p^{n+1}$ ,  $\mathbf{x}_p^{n+1}$ ,  $m_p$ ,  $\mathbf{h}_p^{n+1}$ 
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 COMPUTESTRESS TENSOR routine that is specific to each constitutive model.

Algorithm 17 Computing the stress tensor

Require: $\Delta t, \mathbf{x}_p^n, m_p, V_p^{n+1}, \mathbf{h}_p^n, \mathbf{l}_p^{n+1}, \mathbf{F}_p^{n+1}, \boldsymbol{\sigma}_p^n, \boldsymbol{\eta}_p^n, \rho_o, \text{materialList}, \text{mpmFlags}, \text{constitutiveModel}$

```

1: procedure COMPUTESTRESS TENSOR
2:   for matl in materialList do
3:      $\boldsymbol{\sigma}_p^{n+1}, \boldsymbol{\eta}_p^{n+1} \leftarrow \text{constitutiveModel}[\text{matl}].\text{COMPUTESTRESS TENSOR}(\Delta t, \mathbf{x}_p^n, m_p, V_p^{n+1}, \mathbf{h}_p^n,$ 
        $\hookrightarrow \mathbf{l}_p^{n+1}, \mathbf{F}_p^{n+1}, \boldsymbol{\sigma}_p^n, \boldsymbol{\eta}_p^n, \rho_o, \text{mpmFlags})$  ▷ Update the stress and any internal variables
        $\hookrightarrow \text{needed by the constitutive model}$ 
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 ($\boldsymbol{\epsilon}^p$).

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_o : The initial porosity.
- S_o : The initial porosity.
- n_{\max} : The maximum number of subcycles in the plastic return algorithm.
- $\epsilon_{v,p}^{e,n}$: The elastic volumetric strain at a particle at $t = t_n$.
- σ_p^n : The dynamic Cauchy stress at a particle at $t = t_n$.
- $\sigma_{qs,p}^n$: The quasistatic Cauchy stress at a particle at $t = t_n$.
- σ^o : The initial Cauchy stress at a particle.

Algorithm 18 Creating the Arena constitutive model object

Require: `mpmFlags`, `xmlProblemSpec`

```

1: procedure CREATECONSTITUTIVEMODEL
2:   elasticityModel  $\leftarrow$  ElasticModuliModelFactory.CREATE(xmlProblemSpec)
3:   yieldCondition  $\leftarrow$  YieldConditionFactory.CREATE(xmlProblemSpec)
4:    $p_o, p_1, \bar{p}_1^{\text{sat}}, p_2 \leftarrow$  READCRUSHCURVEPARAMETERS(xmlProblemSpec)
5:    $\phi_o, S_o, \bar{p}_o^w \leftarrow$  READINITIALPOROSITYANDSATURATION(xmlProblemSpec)
6:    $n_{\max} \leftarrow$  READSUBCYCLINGCHARACTERISTICNUMBER(xmlProblemSpec)
7:   return elasticityModel, yieldCondition,  $\phi_o, S_o, \bar{p}_o^w, n_{\max}, p_o, p_1, \bar{p}_1^{\text{sat}}, p_2$ 
8: end procedure

```

Algorithm 19 Initializing the Arena particle variables

Require: ϕ_o, S_o, \bar{p}_o^w , `particleList`, $V_p^o, m_p^o, \mathbf{v}_p^o$, `fluidParams`, `elasticityModel`, `yieldCondition`

```

1: procedure INITIALIZE
2:   yieldCondition.INITIALIZE(particleList,  $V_p^o$ )
3:   yieldParams  $\leftarrow$  yieldCondition.GETPARAMETERS()
4:   for part in particleList do
5:      $\phi_p^o[\text{part}] \leftarrow \phi_o$ 
6:      $S_{w,p}^o[\text{part}] \leftarrow S_o$ 
7:      $\chi_p^o[\text{part}] \leftarrow 0$ 
8:      $\epsilon_{v,p}^{e,o}[\text{part}] \leftarrow 0$ 
9:      $\sigma_p^o[\text{part}] \leftarrow (\text{fluidParams}.\bar{p}_o^w) \mathbf{I}$ 
10:     $\sigma_{qs,p}^o[\text{part}] \leftarrow \sigma^o$ 
11:   end for
12:    $\Delta t \leftarrow$  COMPUTESTABLETIMESTEP( $V_p^o, m_p, \mathbf{v}_p^o$ , elasticityModel)
13:   return  $\phi_p^o, S_{w,p}^o, \chi_p^o, \epsilon_{v,p}^{e,o}, \sigma_{qs,p}^o, \sigma_p^o, \Delta t$ 
14: end procedure

```

5.2 Computing the stress and internal variables

The `COMPUTESTRESS` 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 $t = t_n$.
- $\epsilon_p^{p,n}$: The particle plastic strain tensor at $t = t_n$.
- α_p^n : The particle backstress tensor at $t = t_n$.

- \mathbf{d}^n : The particle rate of deformation tensor at $t = t_n$.
- $\mathbf{R}^n, \mathbf{U}^n$: 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: $\Delta t, \mathbf{x}_p^n, m_p, V_p^{n+1}, \mathbf{h}_p^n, \mathbf{l}_p^{n+1}, \mathbf{F}_p^n, \mathbf{F}_p^{n+1}, X_p^n, \kappa_p^n, \varepsilon_{v,p}^n, p_{3,p}^n, \epsilon_p^{p,n}, \alpha_p^n, \phi_p^n, S_{w,p}^n, \chi_p^n, \varepsilon_{v,p}^n, \sigma_{qs,p}^n, \sigma_p^n, \rho_0,$
 $\text{particleList}, \text{mpmFlags}, \text{elasticityModel}, \text{yieldCondition}$

```

1: procedure COMPUTESTRESS TENSOR
2:    $\text{yieldParams} \leftarrow \text{yieldCondition}.\text{GETPARAMETERS}()$ 
3:    $a_{1,p}, a_{2,p}, a_{3,p}, a_{4,p}, I_{1,p}^{\text{peak}}, R_{c,p} \leftarrow \text{yieldCondition}.\text{GETLOCALVARIABLES}()$ 
    $\hookrightarrow$   $\triangleright$ Yield condition parameters vary at each particle.
    $\hookrightarrow$  Get the per-particle values of these parameters.
4:   for  $\text{part}$  in  $\text{particleList}$  do
5:      $\chi_p^{n+1}[\text{part}] \leftarrow \chi_p^n[\text{part}]$   $\triangleright$ Copy over failure indicator variable
6:      $\mathbf{d}^{n+1} \leftarrow [\mathbf{l}_p^{n+1}[\text{part}] + (\mathbf{l}_p^{n+1}[\text{part}])^T]/2$   $\triangleright$ Compute rate of deformation
7:      $\mathbf{R}^n, \mathbf{U}^n \leftarrow \text{POLARDECOMPOSITION}(\mathbf{F}_p^n[\text{part}])$   $\triangleright$ Compute rotation and stretch tensors
8:      $\mathbf{d}_{\text{unrot}}^{n+1} \leftarrow (\mathbf{R}^n)^T \cdot \mathbf{d}^{n+1} \cdot \mathbf{R}^n$   $\triangleright$ Unrotate the rate of deformation tensor
9:      $\sigma_{qs,\text{unrot}}^n \leftarrow (\mathbf{R}^n)^T \cdot \sigma_{qs,p}^n[\text{part}] \cdot \mathbf{R}^n$   $\triangleright$ Unrotate the quasistatic stress
10:     $\sigma_{\text{unrot}}^n \leftarrow (\mathbf{R}^n)^T \cdot \sigma_p^n[\text{part}] \cdot \mathbf{R}^n$   $\triangleright$ Unrotate the total stress
11:     $K^n, G^n, s^n, (\bar{p}^w)^n, I_1^{\text{eff},n}, \sqrt{J_2^n}, r^n, z_{\text{eff}}^n, \varepsilon_v^{p,n} \leftarrow$ 
       $\hookrightarrow \text{COMPUTEELASTICPROPERTIES}(\sigma_{qs,\text{unrot}}^n[\text{part}],$ 
       $\hookrightarrow \phi_p^n[\text{part}], S_{w,p}^n[\text{part}], \epsilon_p^{p,n}[\text{part}], \alpha_p^n[\text{part}], p_{3,p}^n[\text{part}])$ 
       $\hookrightarrow$   $\triangleright$ Compute elastic properties and stress invariants
12:     $\text{isSuccess}, \sigma_{qs,\text{unrot}}^{n+1}, \phi_{qs}^{n+1}, S_{w,qs}^{n+1}, X_p^{n+1}[\text{part}], \alpha_p^{n+1}[\text{part}], \epsilon_p^{p,n+1}[\text{part}] \leftarrow$ 
       $\hookrightarrow \text{RATEINDEPENDENTPLASTICUPDATE}(\Delta t, \mathbf{d}_{\text{unrot}}^{n+1}, K^n, G^n, s^n, (\bar{p}^w)^n, I_1^{\text{eff},n}, \sqrt{J_2^n}, r^n, z_{\text{eff}}^n,$ 
       $\hookrightarrow \varepsilon_v^{p,n}, \sigma_{qs,\text{unrot}}^n, \phi_p^n[\text{part}], S_{w,p}^n[\text{part}], X_p^n[\text{part}], \alpha_p^n[\text{part}], \epsilon_p^{p,n}[\text{part}], p_{3,p}^n[\text{part}],$ 
       $\hookrightarrow a_{1,p}[\text{part}], a_{2,p}[\text{part}], a_{3,p}[\text{part}], a_{4,p}[\text{part}])$ 
       $\hookrightarrow$   $\triangleright$ Compute updated quasistatic state using the consistency bisection algorithm
13:    if  $\text{isSuccess} = \text{FALSE}$  then
14:       $\text{FLAGPARTICLEFORDELETION}(\text{part})$ 
15:    end if
16:     $\sigma_{\text{unrot}}^{n+1} \leftarrow \text{RATEDEPENDENTPLASTICUPDATE}(\Delta t, \mathbf{d}_{\text{unrot}}^{n+1}, \sigma_{qs,\text{unrot}}^n, \sigma_{qs,\text{unrot}}^{n+1}, \phi_p^n[\text{part}], \phi_{qs}^{n+1},$ 
       $\hookrightarrow S_{w,p}^n[\text{part}], S_{w,qs}^{n+1}, X_p^n[\text{part}], X_p^{n+1}[\text{part}], \alpha_p^n[\text{part}], \alpha_p^{n+1}[\text{part}],$ 
       $\hookrightarrow \epsilon_p^{p,n}[\text{part}], \epsilon_p^{p,n+1}[\text{part}], p_{3,p}^n[\text{part}],$ 
       $\hookrightarrow a_{1,p}[\text{part}], a_{2,p}[\text{part}], a_{3,p}[\text{part}], a_{4,p}[\text{part}])$ 
17:     $\mathbf{R}^{n+1}, \mathbf{U}^{n+1} \leftarrow \text{POLARDECOMPOSITION}(\mathbf{F}_p^{n+1}[\text{part}])$   $\triangleright$ Compute rotation and stretch tensors
18:     $\sigma_{p,qs}^{n+1}[\text{part}] \leftarrow \mathbf{R}^{n+1} \cdot \sigma_{qs,\text{unrot}}^{n+1} \cdot (\mathbf{R}^{n+1})^T$   $\triangleright$ Rotate the quasistatic stress
19:     $\sigma_p^{n+1}[\text{part}] \leftarrow \mathbf{R}^{n+1} \cdot \sigma_{\text{unrot}}^{n+1} \cdot (\mathbf{R}^{n+1})^T$   $\triangleright$ Rotate the dynamic stress
20:  end for
21:   $\Delta t^{n+1} \leftarrow \text{COMPUTESTABLETIMESTEP}(V_p^{n+1}, m_p, \sigma_p^{n+1}, \sigma_p^n, \mathbf{l}_p^{n+1}, \text{elasticityModel})$ 
22:  return  $\sigma_p^{n+1}, \sigma_{p,qs}^{n+1}, \phi_p^{n+1}, S_{w,p}^{n+1}, \alpha_p^{n+1}, X_p^{n+1}, \epsilon_p^{p,n+1}, \Delta t^{n+1}.$ 
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:** $\sigma, \phi, S_w, \epsilon^p, \alpha, p_3, \text{elasticityModel}$

```

1: procedure COMPUTEELASTICPROPERTIES
2:    $s, \bar{p}^w, I_1^{\text{eff}}, \sqrt{J_2}, r, z_{\text{eff}} \leftarrow \text{UPDATESTRESSINVARIANTS}(\sigma, \alpha)$   $\triangleright$  Compute the deviatoric stress and the
   invariants of the input stress tensor.
3:    $\epsilon_v^p \leftarrow \text{UPDATEVOLUMETRICPLASTICSTRAIN}(\epsilon^p)$   $\triangleright$  Compute the volumetric plastic strain from the
   input plastic strain tensor.
4:    $K, G \leftarrow \text{elasticityModel.GETCURRENTELASTICMODULI}(I_1^{\text{eff}}, \bar{p}^w, \epsilon_v^p, \phi, S_w)$   $\triangleright$  Compute the
   elastic moduli corresponding to the input state.
5:   if useDisaggregationAlgorithm = TRUE then
6:     scale = MAX(exp[-(p3 +  $\epsilon_v^p$ )], 10-5)
7:      $K \leftarrow K \cdot \text{scale}$ ,  $G \leftarrow G \cdot \text{scale}$ 
8:   end if
9:   return  $K, G, s, \bar{p}^w, I_1^{\text{eff}}, \sqrt{J_2}, r, z_{\text{eff}}, \epsilon_v^p$ 
10: end procedure

```

Algorithm 22 Updating the stress invariants

```

1: procedure UPDATESTRESSINVARIANTS( $\sigma, \alpha$ )
2:    $I_1 \leftarrow \text{tr}(\sigma)$ 
3:    $s \leftarrow \sigma - (I_1/3)I$   $\triangleright$  Compute deviatoric stress
4:    $\bar{p}^w \leftarrow -\text{tr}(\alpha)/3$   $\triangleright$  Compute pore pressure
5:    $I_1^{\text{eff}} \leftarrow I_1 + 3\bar{p}^w$ ,  $\sqrt{J_2} \leftarrow \sqrt{(1/2)s:s}$   $\triangleright$  Compute invariants of the effective stress
6:    $r \leftarrow \sqrt{2J_2}$ ,  $z_{\text{eff}} \leftarrow I_1^{\text{eff}}/\sqrt{3}$   $\triangleright$  Compute Lode coordinates of the effective stress
7:   return  $s, \bar{p}^w, I_1^{\text{eff}}, \sqrt{J_2}, r, z_{\text{eff}}$ 
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:** $I_1^{\text{eff}}, \bar{p}^w, \epsilon_v^p, \phi, S_w$

```

1: procedure GETCURRENTELASTICMODULI
2:    $\bar{I}_1^{\text{eff}} \leftarrow -I_1^{\text{eff}}$ ,  $\bar{\epsilon}_v^p \leftarrow -\epsilon_v^p$ ,
3:    $K \leftarrow 0$ ,  $G \leftarrow 0$ 
4:   if  $S_w > 0$  then
5:      $K, G \leftarrow \text{COMPUTEPARTIALSATURATEDMODULI}(\bar{I}_1^{\text{eff}}, \bar{p}^w, \bar{\epsilon}_v^p, \phi, S_w)$ 
6:   else
7:      $K, G \leftarrow \text{COMPUTEDRAINEDMODULI}(\bar{I}_1^{\text{eff}}, \bar{\epsilon}_v^p)$ 
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( $\bar{I}_1^{\text{eff}}, \bar{p}^w, \bar{\epsilon}_v^p, \phi, S_w$ )
2:   if  $\bar{I}_1^{\text{eff}} > 0$  then
3:      $\bar{p}^{\text{eff}} \leftarrow \bar{I}_1^{\text{eff}}/3$ 
4:      $K_s \leftarrow K_{s0} + n_s(\bar{p}^{\text{eff}} - \bar{p}_{s0})$ 

```

```

5:    $K_w \leftarrow K_{w0} + n_w(\bar{p}^w - \bar{p}_{w0})$ 
6:    $K_a \leftarrow \gamma(\bar{p}^w + \bar{p}_r)$ 
7:    $K_d, G \leftarrow \text{COMPUTEDRAINEDMODULI}(\bar{I}_1^{\text{eff}}, \bar{\epsilon}_v^{\text{p}})$ 
8:    $K_f \leftarrow 1/[S_w/K_w + (1 - S_w)/K_a]$  ▷ Bulk modulus of air + water mixture
9:    $\text{numer} \leftarrow (1 - K_d/K_s)^2$ 
10:   $\text{denom} \leftarrow (1/K_s)(1 - K_d/K_s) + \phi(1/K_f - 1/K_s)$ 
11:   $K \leftarrow K_d + \text{numer}/\text{denom}$  ▷ Bulk modulus of partially saturated material
   ↪ (Biot-Gassman model)
12:  else
13:     $K, G \leftarrow \text{COMPUTEDRAINEDMODULI}(\bar{I}_1, \bar{\epsilon}_v^{\text{p}})$ 
14:  end if
15:  return  $K, G$ 
16: end procedure

```

Algorithm 25 Computing the drained elastic moduli

Require: $K_{s0}, n_s, \bar{p}_{s0}, b_0, b_1, b_2, b_3, b_4, G_0, \nu_1, \nu_2$

```

1: procedure COMPUTEDRAINEDMODULI( $\bar{I}_1^{\text{eff}}, \bar{\epsilon}_v^{\text{p}}$ )
2:   if  $\bar{I}_1^{\text{eff}} > 0$  then
3:      $\bar{p}^{\text{eff}} \leftarrow \bar{I}_1^{\text{eff}}/3$ 
4:      $K_s \leftarrow K_{s0} + n_s(\bar{p}^{\text{eff}} - \bar{p}_{s0})$ 
5:      $K_s^{\text{ratio}} \leftarrow K_s/(1 - n_s\bar{p}^{\text{eff}}/K_s)$ 
6:      $\epsilon_v^e \leftarrow \text{POW}((b_3\bar{p}^{\text{eff}})/(b_1K_s - b_2\bar{p}^{\text{eff}}), (1/b_4));$ 
7:      $y \leftarrow \text{POW}(\epsilon_v^e, b_4)$ 
8:      $z \leftarrow b_2y + b_3$ 
9:      $K \leftarrow K_s^{\text{ratio}}[b_0 + (1/\epsilon_v^e)b_1b_3b_4y/z^2];$  ▷ 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:     $K \leftarrow b_0K_{s0}$  ▷ 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: $\Delta t, \mathbf{d}^{n+1}, K^n, G^n, \mathbf{s}^n, (\bar{p}^w)^n, I_1^{\text{eff},n}, \sqrt{J_2^n}, r^n, z_{\text{eff}}^n, \epsilon_v^{\text{p},n}, a_1, a_2, a_3, a_4, p_3^n, \sigma_{qs}^n, \phi^n, S_w^n, X^n, \alpha^n, \epsilon^{\text{p},n},$
 $n_{\text{max}}, \epsilon_{\text{sub}}, \chi_{\text{max}}$

```

1: procedure RATEINDEPENDENTPLASTICUPDATE

```

```

2:   $\sigma^{\text{trial}} \leftarrow \text{COMPUTETRIALSTRESS}(\sigma_{qs}^n, K^n, G^n, \mathbf{d}^{n+1}, \Delta t)$  ▷ Compute trial stress
3:   $\alpha^{\text{trial}} \leftarrow \alpha^n, p_3^{\text{trial}} \leftarrow p_3^n, \phi^{\text{trial}} \leftarrow \phi^n, S_w^{\text{trial}} \leftarrow S_w^n,$ 
    $\hookrightarrow X^{\text{trial}} \leftarrow X^n, \epsilon^{\text{p,trial}} \leftarrow \epsilon^{\text{p},n}$  ▷ Set all other trial quantities to the values
   ↪ at the beginning of the timestep
4:   $K^{\text{trial}}, G^{\text{trial}}, \mathbf{s}^{\text{trial}}, (\overline{p^w})^{\text{trial}}, I_1^{\text{eff,trial}}, \sqrt{J_2^{\text{trial}}}, r^{\text{trial}}, z_{\text{eff}}^{\text{trial}}, \epsilon_v^{\text{p,trial}} \leftarrow$ 
    $\hookrightarrow \text{COMPUTEELASTICPROPERTIES}(\sigma^{\text{trial}}, \phi^{\text{trial}}, S_w^{\text{trial}}, \epsilon^{\text{p,trial}}, \alpha^{\text{trial}}, p_3^{\text{trial}})$  ▷ Update the trial
   ↪ values of the moduli and compute the invariants of the trial stress
5:   $n_{\text{sub}} \leftarrow \text{COMPUTESTEPDIVISIONS}(n_{\text{max}}, \epsilon_{\text{sub}}, K^n, K^{\text{trial}}, I_1^{\text{peak}}, a_1, X^n, \sigma_{qs}^n, \sigma^{\text{trial}})$ 
    $\hookrightarrow$  ▷ Compute number of substeps used by the return algorithm
6:  if  $n_{\text{sub}} < 0$  then
7:    return isSuccess = FALSE
8:  end if
9:   $\delta t \leftarrow \frac{\Delta t}{n_{\text{sub}}}$  ▷ Substep timestep
10:  $\chi \leftarrow 1, t_{\text{local}} \leftarrow 0$  ▷ Initialize substep multiplier and accumulated time increment
11:  $\sigma^k \leftarrow \sigma_{qs}^n, \epsilon^{\text{p},k} \leftarrow \epsilon^{\text{p},n}, \phi^k \leftarrow \phi^n, S_w^k \leftarrow S_w^n, X^k \leftarrow X^n, \alpha^k \leftarrow \alpha^n, K^k \leftarrow K^n, G^k \leftarrow G^n, p_3^k \leftarrow p_3^n,$ 
    $\hookrightarrow \mathbf{s}^k \leftarrow \mathbf{s}^n, (\overline{p^w})^k \leftarrow (\overline{p^w})^n, I_1^{\text{eff},k} \leftarrow I_1^{\text{eff},n}, \sqrt{J_2^k} \leftarrow \sqrt{J_2^n}, r^k \leftarrow r^n, z_{\text{eff}}^k \leftarrow z_{\text{eff}}^n, \epsilon_v^{\text{p},k} \leftarrow \epsilon_v^{\text{p},n}$ 
12: repeat
13:   isSuccess,  $\sigma^{k+1}, \epsilon^{\text{p},k+1}, \phi^{k+1}, S_w^{k+1}, X^{k+1}, \alpha^{k+1}, K^{k+1}, G^{k+1}, p_3^{k+1} \leftarrow$ 
      $\hookrightarrow \text{COMPUTESUBSTEP}(\sigma^k, \epsilon^{\text{p},k}, \phi^k, S_w^k, X^k, \alpha^k, K^k, G^k, \mathbf{s}^k, (\overline{p^w})^k, I_1^{\text{eff},k}, \sqrt{J_2^k}, r^k, z_{\text{eff}}^k,$ 
      $\hookrightarrow \epsilon_v^{\text{p},k}, p_3^k, \mathbf{d}^{n+1}, \delta t)$ 
      $\hookrightarrow$  ▷ 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:      $\sigma^k \leftarrow \sigma^{k+1}, \epsilon^{\text{p},k} \leftarrow \epsilon^{\text{p},k+1}, \phi^k \leftarrow \phi^{k+1}, S_w^k \leftarrow S_w^{k+1}, X^k \leftarrow X^{k+1}, \alpha^k \leftarrow \alpha^{k+1}$ 
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  $\chi > \chi_{\text{max}}$  then
22:       return isSuccess = FALSE,  $\sigma^k, \phi^k, S_w^k, X^k, \alpha^k, \epsilon^{\text{p},k}, K^k, G^k, p_3^k$ 
        $\hookrightarrow$  ▷ Algorithm has failed to converge
23:     end if
24:   end if
25:   until  $t_{\text{local}} \geq \Delta t$ 
26:    $\sigma_{qs}^{n+1} \leftarrow \sigma^{k+1}, \alpha^{n+1} \leftarrow \alpha^{k+1}, \epsilon^{\text{p},n+1} \leftarrow \epsilon^{\text{p},k+1}, \phi^{n+1} \leftarrow \phi^{k+1}, S_w^{n+1} \leftarrow S_w^{k+1}, X^{n+1} \leftarrow X^{k+1}$ 
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,  $\sigma_{qs}^{n+1}, \phi^{n+1}, S_w^{n+1}, X^{n+1}, \alpha^{n+1}, \epsilon^{\text{p},n+1}, K^{n+1}, G^{n+1}, p_3^{n+1}$ 
    $\hookrightarrow$  ▷ 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( $\sigma_{qs}^n, K^n, G^n, \mathbf{d}^{n+1}, \Delta t$ )
2:    $\Delta \epsilon \leftarrow \mathbf{d}^{n+1} \Delta t$  ▷ Total strain increment
3:    $\Delta \epsilon^{\text{iso}} \leftarrow \frac{1}{3} \text{tr}(\Delta \epsilon) \mathbf{I}$ 
4:    $\Delta \epsilon^{\text{dev}} \leftarrow \Delta \epsilon - \Delta \epsilon^{\text{iso}}$ 
5:    $\sigma^{\text{trial}} \leftarrow \sigma_{qs}^n + 3K^n \Delta \epsilon^{\text{iso}} + 2G^n \Delta \epsilon^{\text{dev}}$ 

```

```

6:   return  $\sigma_{\text{trial}}$ 
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 σ_{qs}^n and σ_{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: $n_{\text{max}}, \epsilon_{\text{sub}} \leftarrow 10^{-4}, K^n, K^{\text{trial}}, I_1^{\text{peak}}, a_1, X^n, \sigma_{qs}^n, \sigma_{\text{trial}}$

```

1: procedure COMPUTESTEPDIVISIONS
2:    $n_{\text{bulk}} \leftarrow \lceil |K^n - K^{\text{trial}}| / K^n \rceil$   $\triangleright$  Compute change in bulk modulus
3:    $\Delta\sigma \leftarrow \sigma_{\text{trial}} - \sigma_{qs}^n$ 
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:    $n_{\text{yield}} \leftarrow \lceil \epsilon_{\text{sub}} \times \|\Delta\sigma\| / L \rceil$   $\triangleright$  Compute trial stress increment relative to yield surface size
9:    $n_{\text{sub}} \leftarrow \text{MAX}(n_{\text{bulk}}, n_{\text{yield}})$   $\triangleright n_{\text{sub}}$  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: $d^{n+1}, \delta t, \sigma^k, \epsilon^{p,k}, \phi^k, S_w^k, X^k, \alpha^k, K^k, G^k, s^k, (\bar{p}^w)^k, I_1^{\text{eff},k}, \sqrt{J_2^k}, r^k, z_{\text{eff}}^k, \epsilon_v^{p,k}, p_3^k, a_1, a_2, a_3, a_4, I_1^{\text{peak}}, R_c, \beta, \text{yieldCondition}$

```

1: procedure COMPUTESUBSTEP
2:    $\delta\epsilon \leftarrow d^{n+1} \delta t$   $\triangleright$  Compute strain increment
3:    $\sigma_{\text{trial}} \leftarrow \text{COMPUTETRIALSTRESS}(\sigma^k, K^k, G^k, d^{n+1}, \Delta t)$   $\triangleright$  Compute substep trial stress
4:    $\alpha^{\text{trial}} \leftarrow \alpha^k, K^{\text{trial}} \leftarrow K^k, G^{\text{trial}} \leftarrow G^k, p_3^{\text{trial}} \leftarrow p_3^k, \phi^{\text{trial}} \leftarrow \phi^k, S_w^{\text{trial}} \leftarrow S_w^k,$   

    $\hookrightarrow X^{\text{trial}} \leftarrow X^k, \epsilon^{p,\text{trial}} \leftarrow \epsilon^{p,k}$   $\triangleright$  Set all other trial quantities to the values  

    $\hookrightarrow$  at the beginning of the substep
5:    $K^{\text{trial}}, G^{\text{trial}}, s^{\text{trial}}, (\bar{p}^w)^{\text{trial}}, I_1^{\text{eff},\text{trial}}, \sqrt{J_2^{\text{trial}}}, r^{\text{trial}}, z_{\text{eff}}^{\text{trial}}, \epsilon_v^{p,\text{trial}} \leftarrow$   

    $\hookrightarrow \text{COMPUTEELASTICPROPERTIES}(\sigma_{\text{trial}}, \phi^{\text{trial}}, S_w^{\text{trial}}, \epsilon^{p,\text{trial}}, \alpha^{\text{trial}}, p_3^{\text{trial}})$   

    $\hookrightarrow$   $\triangleright$  Compute elastic moduli and stress invariants for the trial state
6:    $\text{isElastic} \leftarrow \text{yieldCondition.EVALYIELDCONDITION}(I_1^{\text{eff},\text{trial}}, \sqrt{J_2^{\text{trial}}}, X^{\text{trial}}, (\bar{p}^w)^{\text{trial}}, \phi^{\text{trial}}, S_w^{\text{trial}},$   

    $\hookrightarrow a_1, a_2, a_3, a_4, I_1^{\text{peak}}, R_c, \beta)$   

    $\hookrightarrow$   $\triangleright$  Determine whether the trial stress is elastic or not
7:   if  $\text{isElastic} = \text{TRUE}$  then

```

```

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

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

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: $\sigma^k, \delta \epsilon, X^k, K^k, G^k, (\bar{p}^w)^k, s^{\text{trial}}, \sqrt{J_2^{\text{trial}}}, r^{\text{trial}}, z_{\text{eff}}^{\text{trial}}, a_1, a_2, a_3, a_4, I_1^{\text{peak}}, R_c, \beta, \text{yieldCondition}$

```

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

```

```

14:   return  $\sigma^{\text{fixed}}, \delta \epsilon^{\text{p, fixed}}$ 
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: $K^k, G^k, X_{\text{eff}}^k, a_1, a_2, a_3, a_4, I_1^{\text{peak}}, R_c, \beta, z_{\text{eff}}^{\text{trial}}, r'_{\text{trial}}$

```

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  $\text{ABS}(\eta^{\text{high}} - \eta^{\text{low}}) > \text{TOLERANCE}$  do
6:      $\mathbf{x}_{\text{poly}} \leftarrow \text{GETYIELDSURFACEPOINTSALL\_RPRIMEZ}(n_{\text{poly}}, K^k, G^k, X_{\text{eff}}^k,$ 
        $\hookrightarrow a_1, a_2, a_3, a_4, I_1^{\text{peak}}, R_c, \beta)$ 
        $\hookrightarrow$  ▷Get the polygon that represents the yield surface in  $z_{\text{eff}}-r'$  space.
7:      $\mathbf{x}_{\text{close}} \leftarrow \text{FINDCLOSESTPOINT}(z_{\text{eff}}^{\text{trial}}, r'_{\text{trial}}, \mathbf{x}_{\text{poly}})$ 
        $\hookrightarrow$  ▷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  $\text{isSuccess} = \text{TRUE}, \mathbf{x}_{\text{close}} \cdot z_{\text{eff}}, \mathbf{x}_{\text{close}} \cdot r'$ 
17: end procedure

```

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

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

Require: $n_{\text{poly}}, K^k, G^k, X_{\text{eff}}^k, a_1, a_2, a_3, a_4, I_1^{\text{peak}}, R_c, \beta$

```

1: procedure GETYIELDSURFACEPOINTSALL\_RPRIMEZ
2:    $\kappa \leftarrow I_1^{\text{peak}} - R_c(I_1^{\text{peak}} - X_{\text{eff}}^k)$  ▷Compute  $\kappa$ .
3:    $I_1^{\text{eff}} \leftarrow \text{Linspace}(\text{from} = X_{\text{eff}}^k, \text{to} = I_1^{\text{peak}}, \text{points} = n_{\text{poly}})$ 
        $\hookrightarrow$  ▷Create an equally spaced set of  $I_1^{\text{eff}}$  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  $X_{\text{eff}}^k < I_1$  then
8:        $F_c^2 = 1 - \left[ \frac{\kappa - I_1}{\kappa - X_{\text{eff}}^k} \right]^2$  ▷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:   $\mathbf{x}_{\text{poly}} \cdot z_{\text{eff}} \leftarrow z_{\text{eff}} \cup \text{REVERSE}(z_{\text{eff}}), \mathbf{x}_{\text{poly}} \cdot r' \leftarrow r' \cup \text{REVERSE}(-r')$ 
        $\hookrightarrow$  ▷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 $z_{\text{eff}}-r'$ space

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

Require: $\mathbf{x}_{\text{trial}}, z_{\text{eff}}, \mathbf{x}_{\text{trial}} \cdot \mathbf{r}', \mathbf{x}_{\text{poly}}$

```

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:      $t \leftarrow \frac{\mathbf{x}_{\text{proj}} \cdot \mathbf{x}_{\text{seg}}}{\|\mathbf{x}_{\text{seg}}\|^2}$ 
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:   $d_{\text{min}}^2 \leftarrow \text{DOUBLE\_MAX}$ 
17:   $\mathbf{x}_{\text{close}} \leftarrow \mathbf{0}$ 
18:  for  $\mathbf{x}_i$  in  $\mathbf{x}$  do
19:     $d^2 \leftarrow \text{DISTANCESQ}(\mathbf{x}_i, \mathbf{x}_{\text{trial}})$ 
20:    if  $d^2 < d_{\text{min}}^2$  then
21:       $d_{\text{min}}^2 \leftarrow d^2$ 
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: $\delta \epsilon, \epsilon^{p,k}, \sigma^k, K^k, G^k, (\bar{p}^w)^k, \phi^k, S_w^k, X^k, \mathbf{s}^{\text{trial}}, I_1^{\text{eff,trial}}, \sqrt{J_2^{\text{trial}}}, r^{\text{trial}}, z_{\text{eff}}^{\text{trial}}, \epsilon_v^{p,\text{trial}}, p_3^{\text{trial}}, a_1, a_2, a_3,$
 $a_4, I_1^{\text{peak}}, R_c, \beta, i_{\text{max}}, j_{\text{max}}, \sigma_{\text{fixed}}, \delta \epsilon_{\text{fixed}}^p, \text{yieldCondition}$

```

1: procedure CONSISTENCYBISECTION
2:    $\delta \epsilon_v^{p,\text{fixed}} \leftarrow \text{tr}(\delta \epsilon_{\text{fixed}}^p)$ 
3:    $i \leftarrow 1$ 
4:    $\eta^{\text{in}} \leftarrow 0, \eta^{\text{out}} \leftarrow 1$ 
5:   while  $\text{ABS}(\eta^{\text{out}} - \eta^{\text{in}}) > \text{TOLERANCE}$  do
6:      $j \leftarrow 1$ 
7:     isElastic  $\leftarrow$  TRUE
8:     while isElastic = TRUE do
9:        $\eta^{\text{mid}} \leftarrow \frac{1}{2}(\eta^{\text{in}} + \eta^{\text{out}})$ 
10:       $\delta \epsilon_v^{p,\text{mid}} \leftarrow \eta^{\text{mid}} \delta \epsilon_v^{p,\text{fixed}}$ 
11:       $(\bar{p}^w)^{\text{mid}}, \phi^{\text{mid}}, S_w^{\text{mid}}, X^{\text{mid}} \leftarrow \text{COMPUTEINTERNALVARIABLES}(K^k, G^k, (\bar{p}^w)^k, \phi^k, S_w^k,$ 
         $\Rightarrow X^k, \delta \epsilon_v^{p,\text{mid}})$  ▷ Update the internal variables using the bisected increment
        ↪ of the volumetric plastic strain

```



```

12:   isElastic ← yieldCondition.EVALYIELDCONDITION( $I_1^{\text{eff,trial}}$ ,  $\sqrt{J_2^{\text{trial}}}$ ,  $X^{\text{mid}}$ ,  $(\bar{p}^w)^{\text{mid}}$ ,
    ↪  $\phi^{\text{mid}}$ ,  $S_w^{\text{mid}}$ ,  $a_1$ ,  $a_2$ ,  $a_3$ ,  $a_4$ ,  $I_1^{\text{peak}}$ ,  $R_c$ ,  $\beta$ )
    ↪ ▷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  $j \geq j_{\text{max}}$  then
17:       return isSuccess ← FALSE ▷The bisection algorithm failed because of
    ↪ too many iterations.
18:     end if
19:   end if
20: end while
21:  $\sigma_{\text{fixed}}^{\text{new}}$ ,  $\delta \epsilon_{\text{fixed}}^{\text{p,new}}$  ← NONHARDENINGRETURN( $\sigma^k$ ,  $\delta \epsilon$ ,  $X^{\text{mid}}$ ,  $K^k$ ,  $G^k$ ,  $(\bar{p}^w)^{\text{mid}}$ ,
    ↪  $s^{\text{trial}}$ ,  $\sqrt{J_2^{\text{trial}}}$ ,  $r^{\text{trial}}$ ,  $z_{\text{eff}}^{\text{trial}}$ ,  $a_1$ ,  $a_2$ ,  $a_3$ ,  $a_4$ ,  $I_1^{\text{peak}}$ ,  $R_c$ ,  $\beta$ )
    ↪ ▷Compute return to updated yield surface (no hardening)
22: if SIGN(tr( $\sigma^{\text{trial}}$  -  $\sigma_{\text{fixed}}^{\text{new}}$ )) ≠ SIGN(tr( $\sigma^{\text{trial}}$  -  $\sigma_{\text{fixed}}$ )) then
23:    $\eta^{\text{out}} \leftarrow \eta^{\text{mid}}$  ▷Too much plastic strain
24:   continue
25: end if
26: if  $\|\delta \epsilon_{\text{fixed}}^{\text{p,new}}\| > \eta^{\text{mid}} \|\delta \epsilon_{\text{fixed}}^{\text{p}}\|$  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  $i \geq i_{\text{max}}$  then
33:   return isSuccess ← FALSE ▷Too many iterations
34: end if
35: end while
36:  $\delta \epsilon_{v,\text{fixed}}^{\text{p,k+1}} \leftarrow \text{tr}(\delta \epsilon_{\text{fixed}}^{\text{p,k+1}})$ 
37:  $(\bar{p}^w)^{k+1}$ ,  $\phi^{k+1}$ ,  $S_w^{k+1}$ ,  $X^{k+1}$  ← COMPUTEINTERNALVARIABLES( $K^k$ ,  $G^k$ ,  $(\bar{p}^w)^k$ ,  $\phi^k$ ,  $S_w^k$ ,
    ↪  $X^k$ ,  $\delta \epsilon_{v,\text{fixed}}^{\text{p,k+1}}$ ) ▷Update the internal variables using the bisected increment
    ↪ of the volumetric plastic strain
38:  $\sigma^{k+1} \leftarrow \sigma_{\text{fixed}}^{\text{new}}$ ,  $\alpha^{k+1} \leftarrow -(\bar{p}^w)^{k+1} \mathbf{I}$ ,  $p_3^{k+1} \leftarrow p_3^{\text{trial}}$ 
39:  $\epsilon^{\text{p,k+1}} = \epsilon^{\text{p,k}} + \delta \epsilon^{\text{p,k+1}}$  ▷Update the plastic strain
40:  $K^{k+1}$ ,  $G^{k+1}$ ,  $s^{k+1}$ ,  $(\bar{p}^w)^{k+1}$ ,  $I_1^{\text{eff,k+1}}$ ,  $\sqrt{J_2^{k+1}}$ ,  $r^{k+1}$ ,  $z_{\text{eff}}^{k+1}$ ,  $\epsilon_v^{\text{p,k+1}} \leftarrow$ 
    ↪ COMPUTEELASTICPROPERTIES( $\sigma^{k+1}$ ,  $\phi^{k+1}$ ,  $S_w^{k+1}$ ,  $\epsilon^{\text{p,k+1}}$ ,  $\alpha^{k+1}$ ,  $p_3^{k+1}$ )
    ↪ ▷Compute elastic moduli and stress invariants for the new state
41: return isSuccess ← TRUE,  $\sigma^{k+1}$ ,  $\epsilon^{\text{p,k+1}}$ ,  $\alpha^{k+1}$ ,  $(\bar{p}^w)^{k+1}$ ,  $\phi^{k+1}$ ,  $S_w^{k+1}$ ,  $X^{k+1}$ ,  $K^{k+1}$ ,  $G^{k+1}$ ,
    ↪  $s^{k+1}$ ,  $(\bar{p}^w)^{k+1}$ ,  $I_1^{\text{eff,k+1}}$ ,  $\sqrt{J_2^{k+1}}$ ,  $r^{k+1}$ ,  $z_{\text{eff}}^{k+1}$ ,  $\epsilon_v^{\text{p,k+1}}$ ,  $p_3^{k+1}$ 
42: end procedure

```

Updating the internal variables

Algorithm 35 Updating the internal variables for partially saturated materials

Require: σ^k , $\epsilon_v^{\text{p,k}}$, K^k , G^k , $(\bar{p}^w)^k$, ϕ^k , S_w^k , X^k , $\delta \epsilon_v^{\text{p}}$, **fluidParams**, **crushParams**, **airModel**, **waterModel**

```

1: procedure COMPUTEINTERNALVARIABLES
2:    $\bar{\epsilon}_v^{\text{p,k}} \leftarrow -\epsilon_v^{\text{p,k}}$ ,  $\delta \bar{\epsilon}_v^{\text{p}} \leftarrow -\delta \epsilon_v^{\text{p}}$ 

```

```

3:  $\bar{p}_o^w \leftarrow \text{fluidParams}.\bar{p}_o^w, S_o \leftarrow \text{fluidParams}.S_o, \phi_o \leftarrow \text{fluidParams}.\phi_o, p_1^{\text{sat}} \leftarrow \text{crushParams}.p_1^{\text{sat}}$ 
4:  $K_a \leftarrow \text{airModel}.\text{COMPUTE BULK MODULUS}((\bar{p}^w)^k)$ 
5:  $K_w \leftarrow \text{waterModel}.\text{COMPUTE BULK MODULUS}((\bar{p}^w)^k)$ 
6:  $\varepsilon_v^{a,o} \leftarrow \text{airModel}.\text{COMPUTE ELASTIC VOLUMETRIC STRAIN}(\bar{p}_o^w)$ 
7:  $\varepsilon_v^a \leftarrow \text{airModel}.\text{COMPUTE ELASTIC VOLUMETRIC STRAIN}((\bar{p}^w)^k)$ 
8:  $\varepsilon_v^w \leftarrow \text{waterModel}.\text{COMPUTE ELASTIC VOLUMETRIC STRAIN}((\bar{p}^w)^k, \bar{p}_o^w)$ 
9:  $\bar{\varepsilon}_v^a \leftarrow -(\varepsilon_v^a - \varepsilon_v^{a,o}), \bar{\varepsilon}_v^w \leftarrow -\varepsilon_v^w$ 
10:  $C_p \leftarrow S_o \exp(\bar{\varepsilon}_v^a - \bar{\varepsilon}_v^w)$ 
11:  $\mathcal{D}_p \leftarrow \frac{1-S_o}{(1-S_o+C_p)^2}$ 
12:  $\frac{dC_p}{d\bar{p}^w} \leftarrow C_p \left[ \frac{1}{K_a} - \frac{1}{K_w} \right]$ 
13:  $\mathcal{G}_a \leftarrow \exp(\bar{\varepsilon}_v^{p,k} - \bar{\varepsilon}_v^a), \mathcal{G}_w \leftarrow \exp(\bar{\varepsilon}_v^{p,k} - \bar{\varepsilon}_v^w)$ 
14:  $\mathcal{B}_p \leftarrow \frac{1}{(1-S_o)} \mathcal{G}_a + S_o \mathcal{G}_w \left[ -\frac{(1-\phi^k)\phi^k}{\phi_o} \left( \frac{S_w^k}{K_w} + \frac{1-S_w^k}{K_a} \right) + \frac{1-S_o}{K_a} \mathcal{G}_a + \frac{S_o}{K_w} \mathcal{G}_w \right]$ 
15:  $(\bar{p}^w)^{k+1} \leftarrow \text{MAX} \left[ (\bar{p}^w)^k + \frac{1}{\mathcal{B}_p} \delta \varepsilon_v^p, 0 \right] \quad \triangleright \text{Update the pore pressure making sure that pressure does}$ 
    $\hookrightarrow \text{not become negative during dilatative plastic deformations.}$ 
16:  $\bar{X}_d, \frac{d\bar{X}_d}{d\varepsilon_v^p} \leftarrow \text{COMPUTE DRAINED HYDROSTATIC STRENGTH AND DERIV}(\bar{\varepsilon}_v^{p,k})$ 
    $\hookrightarrow \quad \triangleright \text{Compute the drained hydrostatic compressive strength and its derivative}$ 
17:  $\bar{X}^{k+1} = -X^k + \left[ (1-S_w^k + p_1^{\text{sat}} S_w^k) \frac{d\bar{X}_d}{d\varepsilon_v^p} + \bar{X}_d (p_1^{\text{sat}} - 1) \frac{\mathcal{D}_p}{\mathcal{B}_p} \frac{dC_p}{d\bar{p}^w} + \frac{3}{\mathcal{B}_p} \right] \delta \varepsilon_v^p$ 
    $\hookrightarrow \quad \triangleright \text{Update the hydrostatic compressive strength}$ 
18:  $\bar{X}^{k+1} \leftarrow -\bar{X}^{k+1}$ 
19:  $\varepsilon_v^{p,k+1} \leftarrow \varepsilon_v^{p,k} + \delta \varepsilon_v^p \quad \triangleright \text{Compute the updated volumetric plastic strain.}$ 
20:  $\varepsilon_v^{a,k+1} \leftarrow \text{airModel}.\text{COMPUTE ELASTIC VOLUMETRIC STRAIN}((\bar{p}^w)^{k+1})$ 
21:  $\varepsilon_v^{w,k+1} \leftarrow \text{waterModel}.\text{COMPUTE ELASTIC VOLUMETRIC STRAIN}((\bar{p}^w)^{k+1}, \bar{p}_o^w)$ 
22:  $\bar{\varepsilon}_v^{a,k+1} \leftarrow -(\varepsilon_v^{a,k+1} - \varepsilon_v^{a,o}), \bar{\varepsilon}_v^{w,k+1} \leftarrow -\varepsilon_v^{w,k+1} \quad \triangleright \text{The updated strains in the fluid phases.}$ 
23:  $C_p^{k+1} \leftarrow S_o \exp(\bar{\varepsilon}_v^{a,k+1} - \bar{\varepsilon}_v^{w,k+1})$ 
24:  $S_w^{k+1} \leftarrow \frac{C_p^{k+1}}{1-S_o+C_p^{k+1}} \quad \triangleright \text{Update the saturation}$ 
25:  $\mathcal{G}_a^{k+1} \leftarrow \exp(\bar{\varepsilon}_v^{p,k+1} - \bar{\varepsilon}_v^{a,k+1}), \mathcal{G}_w^{k+1} \leftarrow \exp(\bar{\varepsilon}_v^{p,k+1} - \bar{\varepsilon}_v^{w,k+1})$ 
26:  $\phi^{k+1} \leftarrow (1-S_o)\phi_o \mathcal{G}_a^{k+1} + S_o \phi_o \mathcal{G}_w^{k+1} \quad \triangleright \text{Update the porosity}$ 
27: return  $(\bar{p}^w)^{k+1}, \phi^{k+1}, S_w^{k+1}, X^{k+1}$ 
28: end procedure

```

Algorithm 36 Computing the drained hydrostatic strength and its derivative

Require: $\bar{\varepsilon}_v^{p,k}, \text{fluidParams}, \text{crushParams}$

```

1: procedure COMPUTEDRAINEDHYDROSTATICSTRENGTHANDDERIV
2:    $\phi_o \leftarrow \text{fluidParams}.\phi_o$ 
3:    $p_o \leftarrow \text{crushParams}.p_o, p_1 \leftarrow \text{crushParams}.p_1, p_1^{\text{sat}} \leftarrow \text{crushParams}.p_1^{\text{sat}}, p_2 \leftarrow \text{crushParams}.p_2$ 
4:    $p_3 \leftarrow -\log(1 - \phi_o)$ 
5:    $\bar{X}_d \leftarrow \text{MAX}(p_o, 1000); \quad \triangleright \bar{X}_d \text{ has a minimum value of 1000 pressure units}$ 
6:    $\frac{d\bar{X}_d}{d\varepsilon_v^p} \leftarrow 0$ 
7:   if  $\bar{\varepsilon}_v^{p,k} > 0$  then
8:      $\phi_{\text{temp}} \leftarrow \exp(-p_3 + \bar{\varepsilon}_v^{p,k})$ 
9:      $\phi \leftarrow 1 - \phi_{\text{temp}}$ 

```

```

10:    $\bar{\xi} \leftarrow p_1 \text{POW}\left(\frac{\phi_o}{\phi} - 1, \frac{1}{p_2}\right)$ 
11:    $\bar{X}_d \leftarrow \bar{X}_d + \bar{\xi}$ 
12:    $\frac{d\bar{X}_d}{d\varepsilon_v^p} \leftarrow \frac{1}{p_2} \frac{\phi_o}{\phi} \phi_{\text{temp}} \frac{\bar{\xi}}{\phi\left(\frac{\phi_o}{\phi} - 1\right)}$ 
13: end if
14: return  $\bar{X}_d, \frac{d\bar{X}_d}{d\varepsilon_v^p}$ 
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: $\Delta t, \mathbf{d}^{n+1}, \boldsymbol{\sigma}^n, K^n, G^n, \phi^n, S_w^n, X^n, \boldsymbol{\alpha}^n, \boldsymbol{\epsilon}^{p,n}, p_3^n, \boldsymbol{\sigma}_{qs}^n, \boldsymbol{\sigma}_{qs}^{n+1}, K^{n+1}, G^{n+1}, \phi^{n+1}, S_w^{n+1}, X^{n+1}, \boldsymbol{\alpha}^{n+1}, \boldsymbol{\epsilon}^{p,n+1}, p_3^{n+1}, a_1, a_2, a_3, a_4, I_1^{\text{peak}}, R_c, \text{yieldParams}$

```

1: procedure RATEDEPENDENTPLASTICUPDATE
2:    $T_1 \leftarrow \text{yieldParams}.T_1, T_2 \leftarrow \text{yieldParams}.T_2$ 
3:   if  $T_1 = 0$  or  $T_2 = 0$  then ▷Check if rate-dependent plasticity has been turned on
4:     return isRateDependent  $\leftarrow$  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:    $\Delta \boldsymbol{\epsilon} \leftarrow \Delta t \mathbf{d}^{n+1}$ 
8:    $\boldsymbol{\sigma}_{\text{trial,dyn}} \leftarrow \text{COMPUTETRIALSTRESS}(\boldsymbol{\sigma}^n, K_{\text{dyn}}, G_{\text{dyn}}, \mathbf{d}^{n+1}, \Delta t)$  ▷Compute substep trial stress
9:    $\dot{\boldsymbol{\epsilon}} \leftarrow \text{MAX}(\|\mathbf{d}^{n+1}\|, \text{ABS\_DOUBLE\_MIN})$ 
10:   $\tau \leftarrow T_1 \text{POW}(\dot{\boldsymbol{\epsilon}}, T_2)$  ▷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:   $\boldsymbol{\sigma}^{n+1} \leftarrow \boldsymbol{\sigma}_{qs}^{n+1} + [(\boldsymbol{\sigma}_{\text{trial,dyn}} - \boldsymbol{\sigma}^n) - (\boldsymbol{\sigma}_{qs}^{n+1} - \boldsymbol{\sigma}_{qs}^n)] R_H + (\boldsymbol{\sigma}^n - \boldsymbol{\sigma}_{qs}^n) r_h$  ▷Stress update
14:  return  $\boldsymbol{\sigma}^{n+1}, \text{isRateDependent} \leftarrow$  TRUE
15: end procedure

```



6 — Submodels of MPM material models

The previous chapter on the ARENA model showed that MPM constitutive models can rapidly become unmanageable without sufficient object granularity. To allow for simpler model development and testing processes to be used in VAANGO, constitutive models are sometime broken up into submodels.

In VAANGO, we have the main constitutive models located in the directory `src/CCA/Components/MPM/ConstitutiveModel`. Some of the models in this directory depend on one of two sets of submodels. The first set of submodels were designed circa 2004-2005 and are located in the directory `...ConstitutiveModel/PlasticityModels`. These models have been retained largely for backward compatibility with older plasticity models. A more recent set of models (designed circa 2012) can be found in `...ConstitutiveModel/Models`.

6.1 Models in the PlasticityModels directory

`PlasticityModels` contains several submodels that are primarily applicable to high strain-rate metal plasticity. The models in this folder are derived from the following base classes:

```
PlasticityModels/  
|-- DamageModel.h  
|-- DevStressModel.h  
|-- FlowModel.h  
|-- KinematicHardeningModel.h  
|-- MeltingTempModel.h  
|-- MPMEquationOfState.h  
|-- ShearModulusModel.h  
|-- SpecificHeatModel.h  
|-- StabilityCheck.h  
|-- ViscoPlasticityModel.h  
|-- YieldCondition.h
```

These models are created inside a `ConstitutiveModel` using the factory idiom. The following factories are available:

```
PlasticityModels/  
|-- DamageModelFactory.cc  
|-- DevStressModelFactory.cc  
|-- FlowStressModelFactory.cc  
|-- KinematicHardeningModelFactory.cc  
|-- MeltingTempModelFactory.cc
```

```

|-- MPMEquationOfStateFactory.cc
|-- ShearModulusModelFactory.cc
|-- SpecificHeatModelFactory.cc
|-- StabilityCheckFactory.cc
|-- ViscoPlasticityModelFactory.cc
|-- YieldConditionFactory.cc

```

The material state is communicated to the submodels through the two structs:

```

PlasticityModels/
|-- DeformationState.h
|-- PlasticityState.h

```

Each model factory can produce several types of submodel objects. The VAANGO implementation contains the following specialized submodels.

6.1.1 Damage models

```

PlasticityModels/
    HancockMacKenzieDamage.cc
    JohnsonCookDamage.cc
    NullDamage.cc

```

6.1.2 Deviatoric stress models

```

PlasticityModels/
    HypoElasticDevStress.cc
    HypoViscoElasticDevStress.cc

```

6.1.3 Flow stress models

```

PlasticityModels/
    IsoHardeningFlow.cc
    JohnsonCookFlow.cc
    MTSFlow.cc
    PTWFlow.cc
    SCGFlow.cc
    ZAFlow.cc
    ZAPolymerFlow.cc

```

6.1.4 Kinematic hardening models

```

PlasticityModels/
    ArmstrongFrederickKinematicHardening.cc
    NoKinematicHardening.cc
    PragerKinematicHardening.cc

```

6.1.5 Melting temperature models

```

PlasticityModels/
    BPSMeltTemp.cc
    ConstantMeltTemp.cc
    LinearMeltTemp.cc
    SCGMeltTemp.cc

```

6.1.6 Equation of state models

```

PlasticityModels/
    DefaultHypoElasticEOS.cc
    HyperElasticEOS.cc
    MieGruneisenEOS.cc
    MieGruneisenEOSEnergy.cc

```

6.1.7 Shear modulus models

```
PlasticityModels/
    ConstantShear.cc
    MTSShear.cc
    NPSShear.cc
    PTWSShear.cc
    SCGSShear.cc
```

6.1.8 Specific heat models

```
PlasticityModels/
    ConstantCp.cc
    CopperCp.cc
    CubicCp.cc
    SteelCp.cc
```

6.1.9 Stability check models

```
PlasticityModels/
    BeckerCheck.cc
    DruckerBeckerCheck.cc
    DruckerCheck.cc
    NoneCheck.cc
```

6.1.10 Viscoplasticity models

```
PlasticityModels/
    SuvicI.cc
```

6.1.11 Yield condition models

```
PlasticityModels/
    GursonYield.cc
    VonMisesYield.cc
```

6.2 Using the models in PlasticityModels

Suppose that you want to design a new constitutive model `MyModel` but want to reuse some of the flow stress models in the `PlasticityModels` directory. The following steps are needed to integrate the submodels into your new model.

1. In your header file, `MyModel.h`, create a private pointer to the model:

```
#ifndef __MPM_CM_MyModel_H__
#define __MPM_CM_MyModel_H__
#include <CCA/Components/MPM/ConstitutiveModel/PlasticityModels/FlowModel.h>
namespace Vaango {
    class MyModel {
    public:
        //.....
    private:
        Uintah::FlowModel* d_flow;
    };
}
#endif
```

2. In your implementation file, `MyModel.cc`, create a copy of the model in the constructor and delete the copy in the destructor:

```
#include <CCA/Components/MPM/ConstitutiveModel/MyModel.h>
#include <CCA/Components/MPM/ConstitutiveModel/PlasticityModels/
    FlowStressModelFactory.h>
```

```

// Constructor
MyModel::MyModel(Uintah::ProblemSpecP& ps, Uintah::MPMFlags* flags) :
    ConstitutiveModel(flags)
{
    d_flow = FlowStressModelFactory::create(ps);
    if (!d_flow) {
        std::ostringstream err;
        err << "An error occurred in the FlowModelFactory that has \n"
            << "slipped through the existing bullet proofing. \n";
        throw Uintah::ProblemSetupException(err.str(), __FILE__, __LINE__);
    }
}

// Copy constructor
MyModel::MyModel(const MyModel* model) : ConstitutiveModel(model)
{
    d_flow = FlowStressModelFactory::createCopy(model->d_flow);
}

// Destructor
MyModel::~MyModel()
{
    delete d_flow;
}

```

3. To make sure that the details of the flow stress model are added to the output for restarting the simulation from a checkpoint, you will have to add the following to `MyModel.cc`:

```

void
MyModel::outputProblemSpec(Uintah::ProblemSpecP& ps, bool output_cm_tag)
{
    Uintah::ProblemSpecP model_ps = ps;
    if (output_cm_tag) {
        model_ps = ps->appendChild("constitutive_model");
        model_ps->setAttribute("type", "my_model_tag");
    }
    d_flow->outputProblemSpec(model_ps);
}

```

4. Some flow stress models have their own associated internal variables. You will have to make sure that these are initialized, even if you don't plan to use a model with submodel internal variables.

```

// Set up particle state
void
MyModel::addParticleState(std::vector<const Uintah::VarLabel*>& from, std::vector<const Uintah::VarLabel*>& to)
{
    d_flow->addParticleState(from, to);
}

// Set up initialization task
void
MyModel::addInitialComputesAndRequires(Uintah::Task* task, const Uintah::MPMMaterial* matl, const Uintah::PatchSet* patch) const
{
    const Uintah::MaterialSubset* matlset = matl->thisMaterial();
    d_flow->addInitialComputesAndRequires(task, matl, patch);
}

// Do the actual initialization
void
MyModel::initializeCMDData(const Uintah::Patch* patch, const Uintah::MPMMaterial* matl, Uintah::DataWarehouse* new_dw)
{
    Uintah::ParticleSubset* pset = new_dw->getParticleSubset(matl->getDWIndex(), patch);
    d_flow->initializeInternalVars(pset, new_dw);
}

```


- Now you are almost ready to use the flow stress model in the stress computation logic. To complete the process, you will have to add a task that makes sure any submodel internal variables are updated correctly during the stress computation process:

```
void
MyModel::addComputesAndRequires(Uintah::Task* task, const Uintah::
    MPMMaterial* matl, const Uintah::PatchSet* patches) const
{
    d_flow->addComputesAndRequires(task, matl, patches);
}
```

- Finally, you can use the flow stress model object for stress computation:

```
void
MyModel::computeStressTensor(const Uintah::PatchSubset* patches, const
    Uintah::MPMMaterial* matl, Uintah::DataWarehouse* old_dw, Uintah::
    DataWarehouse* new_dw)
{
    // Set up initial state
    Uintah::PlasticityState state;
    state.initialTemperature = ...;
    state.initialDensity = ...;
    state.initialVolume = ...;
    state.initialBulkModulus = ...;
    state.initialShearModulus = ...;
    state.initialMeltTemp = ...;
    state.energy = ...;
    // Loop through patches
    for (int patchIndex = 0; patchIndex < patches->size(); patchIndex++) {
        const Patch* patch = patches->get(patchIndex);
        int dwi = matl->getDWIndex();
        // Get the particle set and do gets and allocations
        ParticleSubset* pset = old_dw->getParticleSubset(dwi, patch);
        d_flow->getInternalVars(pset, old_dw);
        d_flow->allocateAndPutInternalVars(pset, new_dw);
        // Loop through particles in a patch
        for (auto idx : *pset) {
            // Update the state
            state.strainRate = ...;
            state.plasticStrainRate = ...;
            state.plasticStrain = ...;
            state.pressure = ...;
            state.temperature = ...;
            state.density = ...;
            state.volume = ...;
            state.bulkModulus = ...;
            state.shearModulus = ...;
            state->meltingTemp = ...;
            state->specificHeat = ...;
            // Calculate the flow stress using the flow stress model
            state->yieldStress = d_flow->computeFlowStress(state, delT, d_tol,
                matl, idx);
            // .....
        } // End particle loop
    } // End patch loop
}
```

6.3 Creating a new model in PlasticityModels

Let us now suppose that you want to add a new flow stress model called **MyFlow** to VAANGO. You will have to use the following process to create the model so that it can be used by the existing constitutive

models as well as the new model that you created in the previous section.

1. First create a header file `MyFlow.h` in the `PlasticityModels` directory:

```
#ifndef __MPM_CM_PM_MyFlow_H__
#define __MPM_CM_PM_MyFlow_H__
#include <CCA/Components/MPM/ConstitutiveModels/PlasticityModels/FlowModel.h>
....
namespace Vaango {
class MyFlow : public FlowModel
{
public:
    // Model parameters
    struct MyFlowParameters {
        double parameter1;
        double parameter2;
        ....
    };

    // Internal variables
    constParticleVariable<double> pInternalVar1, pInternalVar2;
    ParticleVariable<double> pInternalVar1_new, pInternalVar2_new;

    // Internal variable labels
    const Uintah::VarLabel* pInternalVar1Label, pInternalVar2Label;
    const Uintah::VarLabel* pInternalVar1Label_preReloc,
        pInternalVar2Label_preReloc;

    // Constructors and destructor
    MyFlow(Uintah::ProblemSpecP& ps);
    MyFlow(const MyFlow* flow);
    ~MyFlow() override;

    // Delete assignment operator
    MyFlow& operator=(const MyFlow& flow) = delete;

    // Compute the flow stress
    double computeFlowStress(const Uintah::PlasticityState* state, const
        double& delT, const double& tolerance, const Uintah::MPMMaterial*
        matl, const Uintah::particleIndex idx) override;

    // Calculate the plastic strain rate [epdot(tau,ep,T)]
    double computeEpdot(const Uintah::PlasticityState* state, const double&
        delT, const double& tolerance, const Uintah::MPMMaterial* matl,
        const Uintah::particleIndex idx) override;

    // Evaluate derivative of flow stress with respect to scalar and
    // internal variables.
    void evalDerivativeWRTScalarVars(const Uintah::PlasticityState* state,
        const Uintah::particleIndex idx, Uintah::Vector& derivs) override;

    // Evaluate derivative of flow stress with respect to plastic strain
    double evalDerivativeWRTPlasticStrain(const Uintah::PlasticityState*
        state, const Uintah::particleIndex idx) override;

    // Evaluate derivative of flow stress with respect to strain rate.
    double evalDerivativeWRTStrainRate(const Uintah::PlasticityState* state
        , const Uintah::particleIndex idx) override;

    //Compute the elastic-plastic tangent modulus
    void computeTangentModulus(const Uintah::Matrix3& stress, const Uintah
        ::PlasticityState* state, const double& delT, const Uintah::
        MPMMaterial* matl, const Uintah::particleIndex idx, Uintah::
```

```

    TangentModulusTensor& Ce, Uintah::TangentModulusTensor& Cep)
    override;

    // Flow stress update methods
    void updateElastic(const Uintah::particleIndex idx) override;
    void updatePlastic(const Uintah::particleIndex idx, const double&
        delGamma) override;

    // Shear modulus and melting temperature computations
    double computeShearModulus(const Uintah::PlasticityState* state)
        override;
    double computeMeltingTemp(const Uintah::PlasticityState* state)
        override;

    // Boilerplate methods used by the computational framework
    void addInitialComputesAndRequires(Uintah::Task* task, const Uintah::
        MPMMaterial* matl, const Uintah::PatchSet* patches) override;
    void addComputesAndRequires(Uintah::Task* task, const Uintah::
        MPMMaterial* matl, const Uintah::PatchSet* patches) override;
    void addComputesAndRequires(Uintah::Task* task, const Uintah::
        MPMMaterial* matl, const Uintah::PatchSet* patches, bool recurse,
        bool schedParent) override;
    void allocateCMDDataAddRequires(Uintah::Task* task, const Uintah::
        MPMMaterial* matl, const Uintah::PatchSet* patch, Uintah::MPMLabel*
        lb) override;
    void allocateCMDDataAdd(Uintah::DataWarehouse* new_dw, Uintah::
        ParticleSubset* addset, Uintah::ParticleLabelVariableMap* newState,
        Uintah::ParticleSubset* delset, Uintah::DataWarehouse* old_dw)
        override;
    void allocateAndPutRigid(Uintah::ParticleSubset* pset, Uintah::
        DataWarehouse* new_dw) override;

    // Methods needed if the model has its own internal variables
    void addParticleState(std::vector<const VarLabel*>& from, std::vector<
        const VarLabel*>& to) override;
    void initializeInternalVars(Uintah::ParticleSubset* pset, Uintah::
        DataWarehouse* new_dw) override;
    void getInternalVars(Uintah::ParticleSubset* pset, Uintah::
        DataWarehouse* old_dw) override;
    void allocateAndPutInternalVars(Uintah::ParticleSubset* pset, Uintah::
        DataWarehouse* new_dw) override;

private:
    MyFlowParameters d_flow;
    void createInternalVarLabels();
}
#endif

```

2. In the implementation file `MyFlow.cc`, create the constructors and make sure that the model input parameters are copied to output files:

```

#include <CCA/Components/MPM/ConstitutiveModels/PlasticityModels/FlowModel.h>

MyFlow::MyFlow(Uintah::ProblemSpecP& ps)
{
    ps->require("my_flow_parameter1", d_flow.parameter1);
    ps->require("my_flow_parameter2", d_flow.parameter2);
    createInternalVarLabels();
}

MyFlow::MyFlow(const MyFlow* flow)
{
    d_flow.parameter1 = flow->d_flow.parameter1;
}

```

```

    d_flow.parameter2 = flow->d_flow.parameter2;
    createInternalVarLabels();
}

void
MyFlow::createInternalVarLabels()
{
    pInternalVar1Label = Uintah::VarLabel::create("p.my_flow_var1", Uintah::
        ParticleVariable<double>::getTypeDescription());
    pInternalVar2Label = Uintah::VarLabel::create("p.my_flow_var2", Uintah::
        ParticleVariable<double>::getTypeDescription());
    pInternalVar1Label_preReloc = Uintah::VarLabel::create("p.my_flow_var1+",
        Uintah::ParticleVariable<double>::getTypeDescription());
    pInternalVar2Label_preReloc = Uintah::VarLabel::create("p.my_flow_var2+",
        Uintah::ParticleVariable<double>::getTypeDescription());
}

void
MyFlow::outputProblemSpec(Uintah::ProblemSpecP& ps)
{
    ProblemSpecP flow_ps = ps->appendChild("flow_model");
    flow_ps->setAttribute("type", "my_flow_model");
    flow_ps->appendElement("my_flow_parameter1", d_flow.parameter1);
    flow_ps->appendElement("my_flow_parameter2", d_flow.parameter2);
}

```

3. We can now implement the flow stress computation code:

```

double
MyFlow::computeFlowStress(const Uintah::PlasticityState* state, const
    double&, const double&, const Uintah::MPMMaterial*, const Uintah::
    particleIndex idx)
{
    // Get the internal variables
    double internalVar1 = pInternalVar1[idx];
    double internalVar2 = pInternalVar2[idx];

    // Compute the flow stress
    double flowStress = doSomeComputation(internalVar1, internalVar2);

    // Update the internal variables if needed
    pInternalVar1_new[idx] = internalVar1;
    pInternalVar2_new[idx] = internalVar2;

    return flowStress;
}

```

4. Compute the plastic strain rate if possible:

```

double
MyFlow::computeEpdot(const Uintah::PlasticityState* state, const double&
    delT, const double&, const Uintah::MPMMaterial*, const Uintah::
    particleIndex pidx)
{
    double epdot = doSomComputation();
    return epdot;
}

```

5. Compute the derivative of the flow stress function with respect to the scalar plastic strain and strain rate:

```

double
MyFlow::evalDerivativeWRTPlasticStrain(const Uintah::PlasticityState* state
    , const Uintah::particleIndex idx)

```

```

{
    // Do the calculation
    return derivative;
}
double
MyFlow::evalDerivativeWRTStrainRate(const Uintah::PlasticityState* state,
    const Uintah::particleIndex idx)
{
    // Do the calculation
    return derivative;
}

```

6. Compute the derivative of the flow stress function with respect to some scalar quantities if needed:

```

void
MyFlow::evalDerivativeWRTScalarVars(const Uintah::PlasticityState* state,
    const Uintah::particleIndex idx, Uintah::Vector& derivs)
{
    derivs[0] = evalDerivativeWRTStrainRate(state, idx);
    derivs[1] = evalDerivativeWRTTemperature(state, idx);
    derivs[2] = evalDerivativeWRTPlasticStrain(state, idx);
}

```

7. Compute the tangent modulus for implicit calculations. This is typically quite involved and usually avoided by the plasticity models in VAANGO.
8. Compute the shear modulus and melting temperature if needed. We usually use the separate shear modulus and melting temperature models to accomplish this task.
9. At this stage we should set up the boilerplate code for communicating the internal variables:

```

void
MyFlow::addInitialComputesAndRequires(Uintah::Task* task, const Uintah::
    MPMMaterial* matl, const PatchSet*)
{
    const Uintah::MaterialSubset* matlset = matl->thisMaterial();
    task->computes(pInternalVar1Label, matlset);
    task->computes(pInternalVar2Label, matlset);
}

void
MyFlow::addComputesAndRequires(Uintah::Task* task, const Uintah::
    MPMMaterial* matl, const Uintah::PatchSet*)
{
    const Uintah::MaterialSubset* matlset = matl->thisMaterial();
    task->requires(Task::OldDW, pInternalVar1Label, matlset, Ghost::None);
    task->requires(Task::OldDW, pInternalVar2Label, matlset, Ghost::None);
    task->computes(pInternalVar1Label_preReloc, matlset);
    task->computes(pInternalVar2Label_preReloc, matlset);
}

void
MyFlow::addComputesAndRequires(Uintah::Task* task, const Uintah::
    MPMMaterial*, const Uintah::PatchSet*, bool, bool)
{
    const Uintah::MaterialSubset* matlset = matl->thisMaterial();
    task->requires(Task::ParentOldDW, pInternalVar1Label, matlset, Ghost::
        None);
    task->requires(Task::ParentOldDW, pInternalVar2Label, matlset, Ghost::
        None);
}

void
MyFlow::addParticleState(std::vector<const Uintah::VarLabel*>&, std::vector
    <const Uintah::VarLabel*>&)

```

```

{
    from.push_back(pInternalVar1Label);
    from.push_back(pInternalVar2Label);
    to.push_back(pInternalVar1Label_preReloc);
    to.push_back(pInternalVar2Label_preReloc);
}

void
MyFlow::initializeInternalVars(Uintah::ParticleSubset* pset, Uintah::
    DataWarehouse* new_dw)
{
    new_dw->allocateAndPut(pInternalVar1_new, pInternalVar1Label, pset);
    new_dw->allocateAndPut(pInternalVar2_new, pInternalVar2Label, pset);
    for (auto pidx : *pset) {
        pInternalVar1_new[pidx] = 0.0;
        pInternalVar2_new[pidx] = 0.0;
    }
}

void
MyFlow::getInternalVars(Uintah::ParticleSubset* pset, Uintah::DataWarehouse
    * old_dw)
{
    old_dw->get(pInternalVar1, pInternalVar1Label, pset);
    old_dw->get(pInternalVar2, pInternalVar2Label, pset);
}

void
MyFlow::allocateAndPutInternalVars(Uintah::ParticleSubset* pset, Uintah::
    DataWarehouse* new_dw)
{
    new_dw->allocateAndPut(pInternalVar1_new, pInternalVar1Label_preReloc,
        pset);
    new_dw->allocateAndPut(pInternalVar2_new, pInternalVar2Label_preReloc,
        pset);
}

void
MyFlow::allocateAndPutRigid(Uintah::ParticleSubset* pset, Uintah::
    DataWarehouse* new_dw)
{
    allocateAndPutInternalVars(pset, new_dw);
    for (auto pidx : *pset) {
        pInternalVar1_new[pidx] = 0.0;
        pInternalVar2_new[pidx] = 0.0;
    }
}

void
MyFlow::allocateCMDDataAddRequires(Uintah::Task* task, const Uintah::
    MPMMaterial* matl, const Uintah::PatchSet*, Uintah::MPMLabel*)
{
    const Uintah::MaterialSubset* matlset = matl->thisMaterial();
    task->requires(Task::NewDW, pInternalVar1Label_preReloc, matlset, Ghost::
        None);
    task->requires(Task::NewDW, pInternalVar2Label_preReloc, matlset, Ghost::
        None);
}

void
MyFlow::allocateCMDDataAdd(Uintah::DataWarehouse* new_dw, Uintah::
    ParticleSubset* addset, Uintah::ParticleLabelVariableMap* newState,
    Uintah::ParticleSubset* delset, Uintah::DataWarehouse* old_dw)

```

```

{
    Uintah::ParticleVariable<double> n_internalVar1, n_internalVar2;
    Uintah::constParticleVariable<double> o_internalVar1, o_internalVar2;

    new_dw->allocateTemporary(n_internalVar1, addset);
    new_dw->allocateTemporary(n_internalVar2, addset);

    new_dw->get(o_internalVar1, pInternalVar1Label_preReloc, delset);
    new_dw->get(o_internalVar2, pInternalVar2Label_preReloc, delset);

    ParticleSubset::iterator o,n = addset->begin();
    for (o = delset->begin(); o != delset->end(); o++, n++) {
        n_internalVar1[*n] = o_internalVar1[*o];
        n_internalVar2[*n] = o_internalVar2[*o];
    }

    (*newState)[pInternalVar1Label] = n_internalVar1.clone();
    (*newState)[pInternalVar2Label] = n_internalVar2.clone();
}

void
MyFlow::updateElastic(const Uintah::particleIndex pidx)
{
    pInternalVar1_new[idx] = pInternalVar1[idx];
    pInternalVar2_new[idx] = pInternalVar2[idx];
}

void
MyFlow::updatePlastic(const particleIndex pidx, const double& someValue)
{
    pInternalVar1_new[idx] = pInternalVar1_new[idx];
    pInternalVar2_new[idx] = pInternalVar2_new[idx] + someValue;
}

```

10. We now add this model to the [FlowStressModelFactory](#) as follows:

```

#include <CCA/Components/MPM/ConstitutiveModel/PlasticityModels/MyFlow.h>
using namespace Uintah;
FlowModel*
FlowStressModelFactory::create(ProblemSpecP& ps)
{
    ProblemSpecP child = ps->findBlock("flow_model");
    if (!child)
        throw ProblemSetupException("Cannot find flow_model tag", __FILE__,
                                     __LINE__);

    string mat_type;
    if (!child->getAttribute("type", mat_type))
        throw ProblemSetupException("No type for flow_model", __FILE__,
                                     __LINE__);

    .....
    else if (mat_type == "my_flow_model") {
        return (scinew Vaango::MyFlow(child));
    }
    .....
}

FlowModel*
FlowStressModelFactory::createCopy(const FlowModel* pm)
{
    .....
    else if (dynamic_cast<const Vaango::MyFlow*>(pm))
        return (scinew Vaango::MyFlow(dynamic_cast<const Vaango::MyFlow*>(pm)));
    ;
    .....
}

```

```
}
```

11. Next we add the new file to the compilation list in `CMakeLists.txt` in the `PlasticityModels` directory:

```
SET(MPM_ConstitutiveModel_PlasticityModels_SRCS
    .....
    MyModel.cc
)
```

12. Finally, we add the new model tags in the `constitutive_models.xml` file in the directory `src/StandAlone/inputs/UPS_SPEC`:

```
<flow_model spec="OPTIONAL NO_DATA" attribute1="type REQUIRED STRING '
    isotropic_hardening, johnson_cook, mts_model, .... ,
    zerilli_armstrong_polymer, my_flow_model'" >
    .....
    <my_flow_parameter1 spec="REQUIRED DOUBLE" need_applies_to "type
        my_flow_model"/>
    <my_flow_parameter2 spec="REQUIRED DOUBLE" need_applies_to "type
        my_flow_model"/>
</flow_model>
```


7 — 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() << "/" << timestep
        << 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
```



8 — Debugging

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

8.1 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!

8.1.1 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>
```

8.1.2 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).

8.1.3 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 [45](#), [46](#)).