Numerical Modelling with Python and XML

Lutz Gross, Hans Mühlhaus, Elspeth Thorne and Ken Steube

Abstract. In this paper we continue the development of our *python*-based package for the solution of partial differential equations using spatial discretization techniques such as the finite element method (FEM), but we take it to a higher level using two approaches: First we define a `Model` class object which makes it easy to break down a complex simulation into simpler sub-models, which then can be linked together into a highly efficient whole. Second, we implement an XML schema in which we can save an entire simulation. This allows implementing check-pointing and also graphical user interfaces to enable non-programmers to use models developed for their research. All this is built upon our *escript* module, which makes it easy to develop numerical models in a very abstract way while using the computational components implemented in C and C++ to achieve extreme high-performance for time-intensive calculations.

Keywords. Partial Differential Equations, Mathematical Modelling, XML schema, Drucker–Prager Flow.

1. Introduction

These days numerical simulation is a team effort combining a variety of skills. In a very simple approach we can identify four groups of people being involved: researchers using numerical simulation techniques to improve the understanding and predict phenomenas in science and engineering, modelers developing and validating mathematical models, computational scientists implementing the underlying numerical methods, software engineers implementing and optimizing algorithms for a particular architecture. Each of these skill levels uses their individual terminology: researcher is using terms such stress, temperature and constitutive laws, while the modeler is expressing his models through functions and partial differential equations. The computational scientist is working with grids, matrices. Software engineers is working with arrays and data structures. Finally, an object such as stress used by a researcher is represented as a suitable, maybe platform dependent, data structure after the modeler has interpreted as a function of spatial coordinates and the computational scientists as values at the center of elements in a finite element mesh. When moving from the software engineer's view towards the view of the researcher the data structures undergo an abstraction process finally only seeing the concept of stress but ignoring the fact that it is a function in the L^2 Sobolev space and represented by using a finite element mesh.

It is also important to point out that each of these layers has an appropriate user environment. For the researcher, this is a set input files describing the problem to be solved, typically in XML [5]. Modelers, mostly not trained as software engineers, prefer to work in script-based environments, such as *python* [7] while computational scientists and software engineers are are working in C and C++ to achieve best possible computational efficiency.

Various efforts have been made to provide tools for computational scientists to develop numerical algorithms, for instance PETSc [13] which is widely used. These tools provide linear algebra concepts such as vectors and matrices and hiding data structures from the user. The need of researchers for an easy to use environment has been addressed through various developments in problem solving environments (PSEs) [8] which is some case can be a simple graphical user interface. There were only a rather small number of activities towards environments for modelers. Two examples for partial differential equation (PDE) based modelling are ELLPACK [11] and FASTFLO [12]. Both products are using there own programming language which is not powerful enough to deal with complex and coupled problems in an easy and efficient way.

The *escript* module [1, 2] which embedded into *python* is an environment in which modelers can develop PDE based models. It is designed to solve general, coupled, time-dependent, non-linear systems of PDEs. It is a fundamental design feature of *escript* that is not tight to a particular spatial discretization technique or PDE solver library. It is seamlessly linked with other tools such as linear algebra tools [14] and visualization tools [15].

In this paper we will give an overview into the basic concepts of *escript* from a modellers point of view We will illustrate its usage for implementing the Drucker–Prager flow model. In the second part we will present the *modelframe* module within *escript*. It provides a framework to implement mathematical models as *python* objects which then can be plugged together to build simulations. We illustrate this approach for the Drucker–Prager flow and show how this model can be linked with a temperature advection–diffusion model without modifying the codes for any of the models. We will then discuss how XML can be used to set simulation parameters but also to define an entire simulation from existing models. The XML files provide an ideal method to build simulations out of PSEs or from web services. The presented implementation of the Drucker-Prager flow has been validated on some test problems but as it not the purpose of the paper to discuss models no results are presented.

2. Drucker–Prager Flow Model

2.1. The General Framework

We will illustrate the concepts for the deformation of a three-dimensional body loaded by the time-dependent volume force f_i and by the surface load g_i. If u_i and σ_{ij} defines the displacement and stress state at a given time t and dt gives a time increment the general framework for implementing a model in the updated Lagrangian framework is given in the following form:

```
0. start time integration, set $t = 0$
1. start iteration at time $t$, set $k = 0$
    1.0. calculate displacement increment $du_i$ from $\sigma^-_{ij}$ and current force $f_i$.
    1.1. update geometry
    1.2. calculate stretching $D_{ij}$ and spin $W_{ij}$ from $v_i$.
    1.3. calculate new stress $\sigma_{ij}$ from $\sigma^-_{ij}$ using $D_{ij}$ and $W_{ij}$.
    1.4. update material properties from $\sigma_{ij}$
    1.5. if not converged, $k \leftarrow k + 1$ and goto 1.0.
2. set $t \leftarrow t + dt$ and goto 1.
```

The superscript $'-'$ refers to values at the previous iteration or time step. To terminate the iteration process at a time step one can use the relative changes in the velocity $v_i = \frac{u-u^-}{dt}$:

$$\|du\|_\infty \leq \zeta_{iter} \|u\|_\infty \tag{2.1}$$

where $\|.\|_\infty$ denotes the maximum norm and ζ_{iter} is a given positive relative tolerance. Alternatively, one can check the change stress.

The keep the relative time integration error below the given tolerance ζ_{time} the time step dt for the next time step should be kept below dt_{max} given by

$$dt_{max} = \frac{dt\ \|u\|_\infty}{\|v - v^-\|_\infty} \zeta_{time} \ . \tag{2.2}$$

which controls an estimate of the local time discretization error for the total displacement u.

The stretching D_{ij} and spin W_{ij} are given as the symmetric and non-symmetric part of the the gradient of du_i:

$$D_{ij} = \frac{1}{2}(du_{i,j} + du_{j,i}) \tag{2.3}$$

$$W_{ij} = \frac{1}{2}(du_{i,j} - du_{j,i}) \tag{2.4}$$

where where for any function Z $Z_{,i}$ denotes the derivative of Z with respect to x_i.

To calculate the displacement increment du_i one has to solve a partial differential equation on the deformed domain Ω^- which takes in tensor notation the form

$$-(S_{ijkl}\ du_{k,l})_{,j} = (\sigma^-_{ij} - K\ \epsilon^{th} \delta_{ij})_{,j} + f_i \tag{2.5}$$

where K is the bulk modulus and S_{ijkl} is the tangential tensor which depends on the rheology that is used. The argument of divergence expression in the right hand side is the Cauchy stress at time $t-dt$ with the mechanical stress σ^-_{ij} at time $t-dt$. The term $K\ \epsilon^{th}\delta_{ij}$ is the thermal stress where the thermal volume strain ϵ^{th} is given as

$$\epsilon^{th} = \theta\ (T - T_{ref}) \tag{2.6}$$

with thermal expansion coefficient θ and current temperature T and the reference temperature T_{ref}. In geoscience applications, f_i typically takes the form of the gravitational force

$$f_i = -\rho\ g\delta_{id} \tag{2.7}$$

which acts oppositely to the positive d direction. The constants g is the gravity constant and ρ is the density which is often given in the form

$$\rho = \rho_0(1 - \theta(T - T_0)) \tag{2.8}$$

where ρ_0 is the density at reference temperature T_0.

The displacement increment has to fulfill the conditions

$$n_j(S_{ijkl}\ du_{k,l} + \sigma^-_{ij} - K\ \epsilon^{th}\delta_{ij}) = g_i \tag{2.9}$$

on the boundary of the domain where g_i is a possibly time-dependent surface load. Moreover, the displacement increment has to meet a constraint of the form

$$du = dt \cdot V_i \text{ on } \Gamma_i \tag{2.10}$$

where Γ_i is a subset of the boundary of the domain. At the first iteration step V_i gives the i-th velocity component acting at the subset Γ_i of the deformed domain and is set to zero for the following steps.

2.2. The Drucker-Prager Model

For the Drucker-Prager Model the stress update is done in the following way: First the stress state at time $t+dt$ due to elastic deformation $\sigma^e_{i,j}$ is calculated:

$$\sigma^e_{ij} = \sigma^-_{ij} + K\ \epsilon^{th}\delta_{ij} + dt\left(2G\ D_{ij} + (K - \frac{2}{3}G)D_{kk}\delta_{ij} + W_{ik}\sigma^-_{kj} - \sigma^-_{ik}W_{kj}\right) \tag{2.11}$$

where G is the shere modulus. Then Based on the elastic trial stress $\sigma^e_{i,j}$ the yield function F is obtained as The yield function is evaluated

$$F = \tau^e - \alpha\ p^e - \tau_Y \tag{2.12}$$

where

$$p^e = -\frac{1}{3}\sigma^e_{kk} \tag{2.13}$$

$$\tau^e = \sqrt{\frac{1}{2}(\sigma^e_{ij})'(\sigma^e_{ij})'} \tag{2.14}$$

with the deviatoric stress

$$(\sigma^e_{ij})' = \sigma^e_{ij} + p^e\delta_{ij}\ . \tag{2.15}$$

The value τ_Y is current shear length and α is the friction parameter which both are given function of the plastic shear stress γ^p.

We want to have the yield function to be non-negative. The factor χ marks when the yield condition is violated:

$$\chi = \begin{cases} 0 & \text{for } F < 0 \\ 1 & \text{else} \end{cases} \tag{2.16}$$

With current hardening modulus $h = \frac{d\tau_Y}{d\gamma^p}$ and dilatency parameter β, which is again a given function of the plastic shear stress γ^p, we set the plastic shear stress increment to

$$\lambda = \chi \frac{F}{h + G + \beta K} \tag{2.17}$$

We then can calculate a new stress as

$$\tau = \tau^e - \lambda G \tag{2.18}$$

$$\sigma_{ij} = \frac{\tau}{\tau^e}(\sigma^e_{ij})' - (p^e + \lambda\beta K)\delta_{ij} \tag{2.19}$$

Finally we can update the plastic shear stress

$$\gamma^p = \gamma^{p-} + \lambda \tag{2.20}$$

and the harding parameter

$$h = \frac{\tau_Y - \tau_Y^-}{\gamma^p - \gamma^{p-}} \tag{2.21}$$

For the Drucker-Prager model the tangential tensor is given as

$$\begin{aligned} S_{ijkl} \;=\; & G(\delta_{ik}\delta_{jl} + \delta_{jk}\delta_{il}) + (K - \tfrac{2}{3}G)\delta_{ij}\delta_{kl} \\ & +(\sigma^-_{ij}\delta_{kl} - \sigma^-_{il}\delta_{jk}) + \frac{1}{2}(\delta_{ik}\sigma^-_{lj} - \delta_{jl}\sigma^-_{ik} + \delta_{jk}\sigma^-_{il} - \delta_{il}\sigma^-_{kj}) \\ & - \frac{\chi}{h + G + \alpha\beta K}\left(\frac{G(\sigma_{ij})'}{\tau} + \beta K\delta_{ij}\right)\left(\frac{G(\sigma_{kl})'}{\tau} + \alpha K\delta_{kl}\right) \end{aligned} \tag{2.22}$$

When implementing the model we have to keep track of the total stress σ, plastic stress γ^p, the shear length τ_Y as well as displacement u and velocity v.

It is also important to keep in mind that the model is temperature-dependent through the thermal strain and through the temperature-dependent density as well as the standard advection-diffusion equation, the so called heat equation [4].

3. Implementation

The key problem in the Drucker-Prager model is to solve a linear, steady partial differential equation. The coefficients of the PDE are given by arithmetic expressions of the PDE solutions. In this chapter we outline the concept of the *python* module *escript* [1, 2] and demonstrate its usage in the Drucker-Prager model.

3.1. Spatial Functions

The solution of a PDE as well as the PDE coefficients are functions of the spatial coordinates varying within a domain. In *escript* a domain is represented in a `Domain` class object. The `Domain` class object does not only hold information about the geometry of the domain but also about a PDE solver library that is used to solve PDEs on this `Domain`. In particular, the `Domain` class object defines the discretization technique, such as finite difference method (FDM) or finite element method [3], which is used to represent spatial functions on the domain. In the current implementation *escript* is linked to FEM solver library *finley* [2] but the design is open and other libraries could be implemented.

A spatial function is represented through its values on sample points. For instance, in the context of the FEM the solution of a PDE is represented by it values on the nodes while PDE coefficients are typically represented by their values at the element centres. The appropriate representation of a function is chosen by *escript* but, if required, can be controlled by the user.

Spatial functions can be manipulated by applying unary operations, for instance cos, sin, log) and be combined through binary operations, for instance +, -,* , /. When combining spatial functions with different representations *escript* uses interpolation to make the representations of the arguments compatible for combination. If in the FEM context a function represented by it values on the element centers and a function represented by it values on the nodes is added together, interpolation from nodes to element centers will be performed before the addition takes place.

In the current version of *escript*, three internal storage schemes for spatial function representation are used. In the most general case individual values are stored for each sample point. Typically this expanded storage scheme is used for the solution of a PDE. In many cases PDE coefficients are constant. In this case, it would be a waste of memory and compute time to use the expanded storage scheme. For this case *escript* stores a single value only and refers to this value whenever the value for a sample point is requested. The third form represents piecewise constant functions which are stored as a dictionary. The keys of the dictionary refer to tags assigned to the sample points at the time of mesh generation. The value at a sample point is accessed via the tag assigned to sample point and the associated value of the tag in the dictionary. Typically, this technique is used when a spatial function is defined according to its location in certain subregions of the domain, for instance in the region of a certain rock type. When data objects using different storage schemes are combined, *escript* chooses an appropriate storage scheme in which to store the result.

The following *python* function implements stress update for the Drucker-Prager model as presented in section 2.2. It takes the last stress σ_{ij}^{-} (=`s_old`), the last plastic stress γ^{p-} (=`gamma_p_old`), displacement increment du_i (=`du`), the thermal volume strain increment ϵ^{th} (=`deps_th`) and the relevant material parameters to return the new stress and plastic stress:

```
from escript import *
def getStress(du,s,gamma_p_old,deps_th,tau_Y,G,K,alpha,beta,h):
  g=grad(du)
  D=symmetric(g)
  W=nonsymmetric(g)
  s_e=s+K*deps_th*kronecker(3)+ \
           2*G*D+(K-2./3*G)*trace(D)*kronecker(3) \
           +2*symmetric(matrix_mult(W,s)
  p_e=-1./3*trace(s_e)
  s_e_dev=s_e+p_e*kronecker(3)
  tau_e=sqrt(1./2*inner(s_e_dev,s_e_dev))
  F=tau_e-alpha*p_e-tau_Y
  chi=whereNonNegative(F)
  l=chi*F/(h+G+alpha*beta*K)
  tau=tau_e-l*G
  s=tau/tau_e*s_e_dev+(p_e+l*beta*K)*kronecker(3)
  gamma_p=gamma_p+l
  return s, gamma_p
```

Each of the arguments, with the exception of `v`, can be a *python* floating point number or an *escript* spatial function. The argument `v` must be an *escript* spatial function. The `Domain` attached to it is the `Domain` of the returned values. Their representations are determined by the `grad` call, which returns the gradient of its argument. In the FEM context, the argument of `grad` must be represented by its values on the nodes while the result of `grad` is represented by its values at the element centers. This representation is carried through to the return values. Notice that this function does not depend on whether values are stored on the nodes or at the centers of the elements, and so it may be re-used without change in any other model, even a finite difference model.

3.2. Linear PDEs

The `LinearPDE` class object provides an interface to define a a general, second order, linear PDE. The `Domain` class object attached to a `LinearPDE` class object defines the domain of the PDE but also defines the PDE solver library to be used to solve the PDE.

The general form of the PDE for an unknown vector-valued function u_i represented by the `LinearPDE` class is

$$-(A_{ijkl}u_{k,l} + B_{ijk}u_k)_{,j} + C_{ikl}u_{k,l} + D_{ik}u_k = -X_{ij,j} + Y_i \tag{3.1}$$

The coefficients A, B, C, D, X and Y are functions of their location in the domain. Moreover, natural boundary conditions of the form

$$n_j\ (A_{ijkl}u_{k,l} + B_{ijk}u_k - X_{ij,j}) = y_i \tag{3.2}$$

can be defined. In this condition, (n_j) defines the outer normal field of boundary of the domain and y is a given function. To set values of u_i to r_i on certain locations of the domains one can define constraints of the form

$$u_i = r_i \text{ where } q_i > 0 \tag{3.3}$$

where q_i is a characteristic function of the locations where the constraint is applied.

From inspecting equation 2.5 and equation 2.9 we can easily identify the value for the PDE coefficient to calculate the displacement increment du_i:

$$A_{ijkl} = S_{ijkl}, Y_i = f_i, X = \sigma^-_{ij} - K\epsilon^{th}\delta_{ij} \text{ and } y_i = g_i \ . \tag{3.4}$$

The constraints for du_i, see equation 2.10 is defined by

$$r_i = dt \cdot V_i \text{ and } q_i(x) = \begin{cases} 1 & \text{for } x \in \Gamma_i \\ 0 & \text{otherwise} \end{cases} \tag{3.5}$$

If the function `getTangentialTensor` returns the tangential operator, the following function returns a new displacement increment `du` by solving a linear PDE for the `Domain` class object `dom`:

```
from escript import LinearPDE
def getDu(dom,dt,s,dt,deps_th,tau_Y,G,K,alpha,beta,h,f,g,Gamma,V):
   pde=LinearPDE(dom)
   S=getTangentialTensor(s,tau_Y,G,K,alpha,beta,h)
   pde.setValue(A=S, X=s-K*deps_th*kronecker(3), \
                Y=f, y=g)
   pde.setValue(q=Gamma, r=dt*V)
   du=pde.getSolution()
   return du
```

It would be more efficient to create one instance of the `LinearPDE` class object and to reuse it for each new displacement increment with updated coefficients. This way information that are expensive to obtain, for instance the pattern of the stiffness matrix, can be preserved. The following function returns the tangential tensor `S`:

```
from escript import *
def getTangentialTensor(s,,tau_Y,G,K,alpha,beta,h):
   k3=kronecker(3)
   p=-1./3*trace(s)
   s_dev=s+p*k3
   tau=sqrt(1./2*inner(s_dev,s_dev))
   chi=whereNonNegative(tau-alpha*p-tau_Y)
   tmp=G*s_dev/tau
   sXk3=outer(s,k3)
   k3Xk3=outer(k3,k3)
   S=  G*(swap_axes(k3Xk3,0,3)+swap_axes(k3Xk3,1,3)) \
     + (K-2./3*G)*k3Xk3 \
     + (sXk3-swap_axes(swap_axes(sXk3,1,2),2,3))    \
     + 1./2*( swap_axes(swap_axes(sXk3,0,2),2,3)    \
             -swap_axes(swap_axes(sXk3,0,3),2,3)    \
             -swap_axes(sXk3,1,2)                  \
             +swap_axes(sXk3,1,3)                ) \
     - outer(chi/(h+G+alpha*beta*K)*(tmp+beta*K*k3),tmp+alpha*K*k3)
   return S
```

The values `chi`, `s_dev` and `tau` calculated in a call of `getStress` could used in this calculation of the new stress could be reduced in `getTangentialTensor`.

4. Modelling Framework

From the functions presented in the previous section 3 one can easily build a time integrations scheme to implement a Drucker-Prager flow model through two nested loops. The outer loop progresses in time while the inner loop iterates at time step until the stopping criterion (2.1) is fulfilled. The time step size is controlled by criterion 2.2. This straight-forward approach has the drawback that the code has to modified when the temperature dependence becomes time-dependent. Then a advection-diffusion model for the temperature has to be worked into the code which may use a different time step size. In addition, the thermal strain ϵ^{th} is changing over time but also material parameters, for instance density, see equation (2.8), may become temperature-dependent. The coding effort to implement these components is not large but has the big drawback that the submodel, such as the Drucker-Prager model, the temperature advection-diffusion model and the material property tables, loose their independence. However, from a software development point of view it is highly desirable to keep the models independent in order to test and maintain them separately and reuse models in another context.

4.1. Linkable Objects

To resolve this problem, a model, such as Drucker-Prager flow model, is implemented in a *python* class. Model parameters, such as external force F_i or state variables, such as stress σ_{ij}, are defined as object attributes. For instance if the class `DruckerPrager` implements the Drucker–Prager flow and `Gravity` the gravity force (2.7) one uses

```
g=Gravity()
g.density=1000
flow=DruckerPrager()
flow.force=g.force
```

to assign a value to the density of the gravity force and then defines the gravity force as the external force of the Drucker–Prager flow model. In the case of convection model, the density is not constant but calculated from a temperature field updated through the temperature model over time. However, the `Gravity` object `g` has to use the most recent value of the density at any time of the simulation. To accomplish this the *escript* module provides two classes `LinkableObjects` and `Link`. A `Link` class object is a callable object that points to an attribute of the target object. If `Link` class object is called the current value of the attribute of the target object is returned. If the value is callable, the value is called and the result is returned. So, similar to a pointer in C, a `Link` class object defines another name for a target value, and is updated along with the target object.

A `LinkableObjects` class object is a standard *python* object with a modified `__getattr__` method: when accessing its attributes it behaves like a usual *python* object except if the value is a `Link` class object. In this case, the link is followed, that means the value is called and the result returned. The mechanism allows to link an attribute of `LinkableObjects` class object with the attribute of another target object such that at any time the object uses the value of the target object.

The following script shows how a gravity force in flow model can be defined where the density of the gravity model is temperature-dependent:

```
mat=MaterialTable()
mat.T_0=20
temp=Temperature()
g=Gravity()
flow=DruckerPrager()
mat.T=Link(temp,"temperature")
g.density=Link(mat,"rho")
flow.force=Link(g,"force")
```

The four classes `MaterialProperties` `Temperature`, `Gravity` and `DruckerPrager` implementing a material property table, a temperature advection-diffusion model, a gravity force model and a Drucker-Prager flow model. The temperature parameter `T` of the material property table is provided by the temperature model attribute `temperature`. The density `rho` provided by the material property table is fed into the gravity model which then provides the external force for the flow model.

The model (2.8) of a temperature-dependent density can be implemented in the following way:

```
class MaterialProperties(LinkableObject):
    def rho(self):
        return self.rho_0*(1.-self.theta*(self.T-self.T_0))
```

If the temperature `T` is linked to a time-dependent temperature model via a `Link` object the method `rho` will always use the most recent value of the temperature even if it is updated by the target temperature model. On the other hand, if an attribute of another model, for instance the attribute `density` of a `Gravity` class object, is linked against the method `rho` of an instance of the `MaterialProperties` class, accessing the attribute will call the the method `rho` and use its return value calculated for the current temperature. Through the chain of links the Drucker-Prager flow model is coupled with the temperature modules. It is pointed out that if the temperature model considers advection, the velocity updated by Drucker-Prager flow model is referenced by the temperature model which produces a loop of references requiring an iteration on a time step.

4.2. Models

Implementing models as `LinkableObject` class objects solves the problem of defining the data flow of coupled models but we also need an execution scheme for individual models in order to be able to execute a set of coupled models without changing any of involved models. An appropriate execution scheme is defined through the `Model` class of the *modelframe* module. The `Model` class which is a subclass of the `LinkableObject` class provides a template for a set of methods that have to be implemented for a particular model. The `Model` class methods are as follows:

- `doInitialization`: This method is used to initialize the time integration process. For the flow model this method initializes displacements and stress for time $t = 0$ and to create an instance of the `LinearPDE` object used to update the displacement through equation (2.5).
- `getSafeTimeStepSize`: This method returns the maximum time step size that can be used to safely execute the next time step of the model. For the flow model the time step size defined by condition (2.2)) is returned.
- `doStepPreprocessing`: This method prepares the iterative process to be executed on a time step. Typically, this function will set an initial guess for the iterative process to run on a time step.
- `doStep`: This method performs a single iteration step to update state variables controlled by the model due to changes in input parameters controlled by other models. For the Drucker–Prager flow model the `doStep` perform a single iteration step by solving the PDE (2.5) and updating the stress via (2.19).
- `terminateIteration`: The method returns `True` if the iterative process on a time step can be terminated. Typically, a stopping criterion of the form (2.1) is checked.
- `doStepPostprocessing`: This method finalizes a time step, typically by shifting the current values of state variables into the buffer of the previous time step.
- `finalize`: The method returns `True` is the time integration is finalized. Typically this method checks the current time against a given end time.
- `doFinalization`: This method finalizes the whole modeling run, typically by closing files and cleaning up.

Any `Model` class object is run in the following way:

```
m=Model()
m.doInitialization()
while not finalize():
      dt=m.getSafeTimeStepSize(dt)
      m.doStepPreprocessing(dt)
      while not m.terminateIteration(): m.doStep(dt)
      m.doStepPostprocessing(dt)
m.doFinalization()
```

This scheme is executed by the `Simulation` class, which will be discussed in the next section 4.3.

The following sub-class of the `Model` class implements the Drucker–Prager flow model:

```
class DruckerPrager(Model):
   def doInitialization(self):
       self.pde=LinearPDE(self.domain)
       self.stress=Tensor(0.,Function(self.doamin))
       self.displacement=Vector(0.,Solution(self.doamin))
       self.velocity=Vector(0.,Solution(self.doamin))
       self.t=0
   def getSafeTimeStepSize(self):
       return dt*Lsup(u)/Lsup(self.velocity-self.velocity_old)*self.zeta
   def doStep(self,dt):
       S=self.getTangentialTensor()
       pde.setValue(A=S, X=self.stress,Y=self.force)
       self.du=self.pde.getSolution()
       self.displacement=self.displacement+self.du
       self.velocity=(self.displacement-self.displacement_old)/dt
       self.stress=self.getStress()
   def terminateIteration(self,dt):
       return Lsup(self.du)<=self.zeta*Lsup(self.displacement)
   def doStepPostprocessing(self,dt):
       self.velocity_old=self.velocity
       self.t+=dt
   def finalize(self):
       return self.t<self.t_end
```

Omitted `Model` class methods are empty. The functions `getStress` and `getTangentialTensor` introduced in section 3.2 are turned into class methods. Function parameters are now accessed as class attributes.

4.3. Simulations

The `Simulation` class controls the execution of a list of submodels. It is an implementation of the `Model` class, where each `Model` method executes the corresponding method of all the submodels: The `getSafeTimeStepSize` returns the minimum of all step sizes required by the submodels. If all submodels terminate their iterative process `terminateIteration` returns `True`. The simulation is terminated by the `finalize` method if any of the submodels is to be finalized. The `doStep` method iterates over all submodels until all submodel indicate convergence.

The following script outlines the usage of the `Simulation` class for constructing and running a coupled temperature diffusion and Drucker-Prager flow model:

```
mat=MaterialTable()
temp=Temperature()
```

```
g=Gravity()
flow=DruckerPrager()
s=Simulation([temp,mat,g,flow])
s.run()
```

The couplings between the models as shown in section 4.1 have been omitted. The `run` method of the `Simulation` class executes its `Model` methods as shown in the previous section 4.2. At a time step, the simulation iterates over the submodels until all submodels have converged. The execution order is defined by the order of the models in `Simulation` class argument. Alternatively, one can iterate over the flow model only and use the temperature from the previous time step. This can be achieved through

```
Simulation([temp,mat,g,Simulation([flow])).run()
```

where the convergence tolerance in the temperature model has to be chosen appropriately. Note that visualization functionality, for instance to create a movie of the velocity field, can be implemented as a `Model` class where in the `doStepPostprocessing` method the velocity field is rendered after for each time step, while the `doFinalization` method creates the movie from the frames.

It is pointed that the coupling of the models does not require any knowledge of the algorithms being used to implement the model. Furthermore, the codes implementing a model are not altered to perform the coupling. The coupling is based purely on the submodel parameters. This high-level approach is possible because of the abstraction level that is provided by the *escript* environment.

5. Simulation and XML

Concurrent to the modeling framework, an XML dialect called `EsysXML` has been developed with the objective to store simulations in a transportable form, to define the coupling of models and to allow the creation of dynamic templates of model interfaces.

5.1. XML Model Interface

The parameters used to link models are called external parameters, and are stored in a `Model` class object. These parameters may be set by the programmer or by another model. The `Model` also stores internal parameters used in the various methods implementing the model. For instance for the `Model` class `DruckerPrager` introduced in section 4.2 `domain`, `force`, `stress` , `displacement` and `velocity` are external parameters while `pde` and `t` are internal parameters that are unavailable to other models. External parameters are exposed via the `declareParameter` method called at `Model` class object initialization. For example, the `Model` class `DruckerPrager` is written in the following way:

```
class DruckerPrager(Model):
   def __init__(self):
      Model.__init__(self)
```

```
    self.declareParameters(domain=None, force=0., \
                           stress=0., displacement=0., \
                           velocity=0.)
```

This explicit declaration restricts the necessary input and output parameters that are exposed on a user-interface level. The XML representation of a `Model` class object includes all parameters and their current values, which could be `Link` class objects. The *python* implementation of a model itself is not stored in XML, only the data and the links between models.

A `Simulation` class object can be written into a XML file using the EsysXML. This includes all submodels, their external parameter and values and possibly links between the submodels. This file can be edited using any text editor to change value of external parameters, introduce new models and set or remove couplings between models. A `Simulation` class object can be build from the XML file. The parser uses the standard *python* XML tools, packaged in `PyXML` [9]. Primarily, `minidom` is used to parse and generate XML content. It is a lightweight, simple implementation of the standard Document Object Model (DOM) interface, and is fast. An important application of writing `Simulation` class objects to XML is automated check-pointing.

5.2. Technical specifications of EsysXML

A diagram of the XML schema [6] used by EsysXML is included in Figure 1 at the end of this paper.

The XML begins with an Esys tag, which includes a single `Simulation`. That `Simulation` contains one or more components, each of which may be either a `Model` or another `Simulation`. The ranks of the `Component` objects tell the order in which they are to be executed. Each `Model` includes one or more `Parameter` objects defined by a type (`float`, `string` or `Link`) and the name and value of the parameter. Whether or not a parameter is internal or external (available to other models) is determined by the use of `declareParameters` in the python script which implements the model.

The `Link` items, require that a persistent, deterministic way of preserving associations between linked `Model` class objects. This is achieved by generating a unique reference identification number for each `Model` class object which defines the `id` attribute. For a `Link` item the `Target` XML attribute specifies the reference identification number of the target `Model` item while the XML attribute `Attribute` specifies the name of `Parameter` item of the target `Model`.

6. Conclusion

The design we have chosen for *escript* simplifies the development of complex yet efficient models by breaking them down into a series of simpler steps. In the example above, we developed a Drucker-Prager flow model and a gravity model, and then later on we linked them together with a temperature adevection-diffusion model to create a more complex simulation.

By modifying the standard *python* mechanism for accessing object attributes, we allow sharing values between models without creating any burden on the modeller. The sharing of parameters between models is very efficient due to our use of links.

By adding the ability to store an entire simulation in XML format we have created a new level of control over a model. Using this technique we will be able to check-point simulations, share simulations with collaborators, and also provide a Graphical User Interface, such as in a web page, to define the parameters of a simulation and adjust models in a simulation. Once a modeller has developed a simulation in *python* it may then be used and adjusted by a researcher who is not burdened to do further programming.

Acknowledgment

Project work is supported by Australian Commonwealth Government through the Australian Computational Earth Systems Simulator Major National Research Facility and the Australian Partnership for Advanced Computing, Queensland State Government through the Smart State Research Facility Fund, The University of Queensland, the Queensland Cyberinfrastructure Foundation and SGI Ltd.

References

[1] Gross, L. and Cochrane, P. and Davies, M. and Mühlhaus, H. and Smillie J.: *Escript: numerical modelling in python.* Proceedings of the Third APAC Conference on Advanced Computing, Grid Applications and e-Research (APAC05),(2005).

[2] Davies, M. and Gross, L. and Mühlhaus, H. -B.: Scripting high-performance Earth systems simulations on the SGI Altix 3700. *Proceedings of the 7th international conference on high-performance computing and grid in the Asia Pacific region*, (2004).

[3] Zienkiewicz, O. C. and Taylor, R. L.: *The Finite Element Method.* 5th Edition, Butterworth Heinemann, (2000).

[4] R. E. Williamson: *Differential Equations and Dynamical Systems*, McGrawHall (2001).

[5] http://www.w3.org/XML/ (2006), [on-line].

[6] C. M. Sperberg-McQueen and H. Thompson: *XML Schema*, www.w3.org/XML/Schema (2006), [on-line].

[7] M. Lutz, *Programming Python, 2nd Edition* O'Reilly (2001).

[8] E. N. Houstis, E. Gallopoulos, J.R. Rice, R. Bramley: *Enabling Technologies for Computational Science* Kluwer Academic Publishers (2000).

[9] http://pyxml.sourceforge.net/ (2006), [on-line].

[10] D. J. Higham, N. J. Higham, *MATLAB Guide.*, SIAM (2000).

[11] J. R. Rice, R .F. Boisvert, *Solving Elliptic Problems Using ELLPACK.* Springer Series in Computational Software **2** (1985).

[12] X.–L. Luo, A. N. Stokes, N. G. Barton, *Turbulent flow around a car body - report of Fastflo solutions* Proc. WUA-CFD Conference, Freiburg (1996).

[13] P. Pacheco, *Parallel Programming with MPI.*, Morgan-Kaufmann (1997).

[14] P. Greenfield, J. T. Miller, J. Hsu, R. L. White. *An Array Module for Python.* in Astronomical Data Analysis Software and Systems XI (2001).

[15] The Kitware, Inc.: *Visualization Toolkit User's Guide.* Kitware, Inc publishers.

Lutz Gross
Earth Systems Science Computational Center
The University of Queensland
St. Lucia., QLD 4072
Australia
e-mail: `l.gross@uq.edu.au`

Hans Mühlhaus
Earth Systems Science Computational Center
The University of Queensland
St. Lucia., QLD 4072
Australia
e-mail: `h.muhlhaus@uq.edu.au`

Elspeth Thorne
Earth Systems Science Computational Center
The University of Queensland
St. Lucia., QLD 4072
Australia
e-mail: `e.thorne@uq.edu.au`

Ken Steube
Earth Systems Science Computational Center
The University of Queensland
St. Lucia., QLD 4072
Australia
e-mail: `k.steube@uq.edu.au`

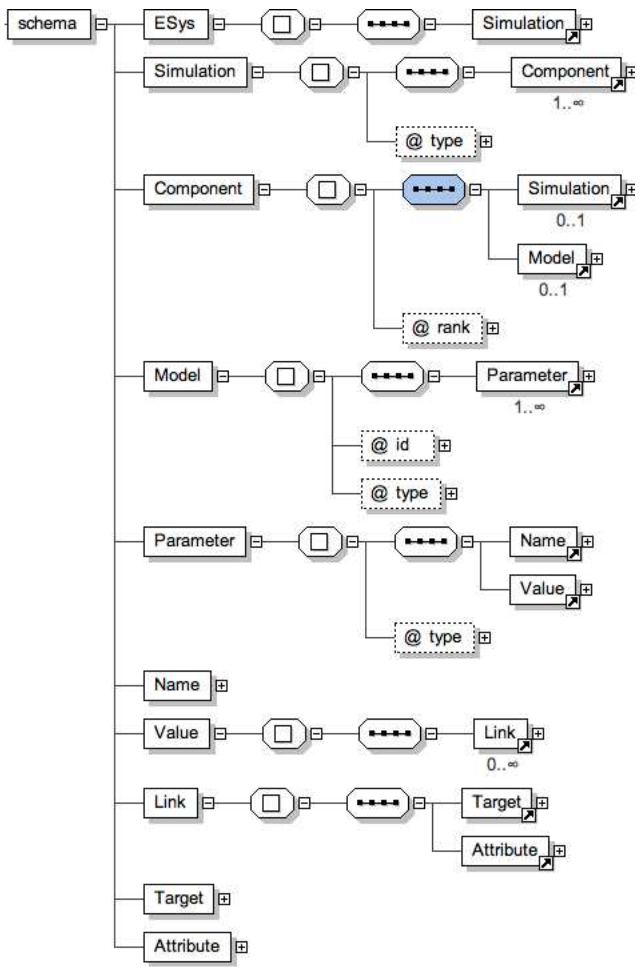

Figure 1. EsysXML Schema Tree