A High-Level Programming Language for Modeling the Earth

Lutz Gross, Jonathan Smillie and Matt Davies

Abstract. Computational models based on the solution of partial differential equations (PDEs) play a key role in Earth systems simulations. The software implementing these models depends on the discretisation method, data structures and the computer architecture. For this reason, it is difficult for scientists to implement new models without strong software engineering skills. In this paper, we present a computational modeling language *escript* based on the object–oriented scripting language *Python*. This language, is designed to implement PDE-based models with a high degree of abstraction from the underlying discretization techniques and their implementation. The main components of *escript* are the `Data` class objects which handle data with a spatial distribution and the `linearPDE` class which define linear PDEs to be solved in each step of a time integration or non-linear iteration scheme. As an example we will discuss the solution of the Lame equation and the implementation of a quasi-static model for crustal fault systems.

Keywords. Partial Differential Equations, Parallel Computing.

1. Introduction

Many mathematical models in Earth sciences are based on the solution of systems of coupled, non-linear, time-dependent partial differential equations (PDEs). Here, the spatial and temporal scales vary from a planetary scale evolving over millions of years to the scale of a fault system evolving over several decades. In this context, various numerical techniques can be applied to manage non-linearity, including the Crank-Nicholson scheme, the Newton-Raphson method, and, for weakly coupled equations, the non-linear Gauss-Seidel scheme. In addition to these high-level techniques, spatial discretization methods such as the finite element method (FEM) or the boundary element method (BEM), can be applied to approximate the spatial derivatives on large, three-dimensional domains, usually decomposed into unstructured meshes in order to resolve geometrical complexity. For the Earth sciences,

geometrical complexity can be inherent in the physical domain, such as that found in a fault system [4], or in resolving high contrasts in the solution profile, as required in mantle convection simulations[2].

Even at this stage of our technological advancement, there remains a dearth of knowledge with respect to the dynamics of the Earth and it's constituent systems. As a consequence, the computational models that are in turn used to study Earth systems are constantly required to rapidly adapt to our current interpretations and observations. For this reason, Earth scientists and geophysicists require a high-level, interactive programming environment which allows them to rapidly implement, test and run new computational models. Established programming environments such as MATLAB [8] and PETSc [12] are built around linear algebra. They do not provide a level of abstraction sufficient to permit the user to work purely within the context of PDEs. By way of a contrast, existing PDE-oriented environments such as ELLPACK [9], VECFEM [10] and FASTFLO [11], do not provide a sufficiently-powerful programming language to handle complex and coupled problems in a tractable manner. These environments are also not bound to an object-oriented programming environment.

In this paper, we present the concepts which underpin the development of *escript*. Its objective is to provide a programming language for defining models and high-level solution algorithms for solving general, time-dependent, non-linear systems of PDEs, independently of the discretization method and underlying software technologies. This approach permits the use of a validated model, in several different contexts and with various PDE solver libraries without changes to the model implementation. In order to reduce development costs and provide a seamless integration with existing toolkits for numerical modeling, such as linear algebra tools [13] and visualization tools [14], *escript* is implemented as an extension of the interactive, object-oriented scripting language *Python* [5].

In this paper, we present *escript* from a user perspective and defer a discussion of the computational kernel's implementation and parallelization to a later paper. In the next section, we present *escript*'s concept of a PDE domain. Sections three and four discuss `Data` objects and `linearPDE` objects, respectively. In section five and six we present the application of *escript* to a simple linear-elastic material model and a quasi-static stick-slip friction model for fault systems. As it is not the purpose of the paper to discuss models and solution algorithms, we will not present numerical results.

2. The PDE Domain

A coupled system of time-dependent non-linear PDEs is solved using a time integration scheme and possibly an iterative procedure in each time step. This leads to a sequence of linear PDEs that have to be solved in each time step or iteration. It is the objective of *escript* to provide an environment in which this process can be easily implemented without actually referring to the linear PDE solver to be used

to solve the linear PDE. The domain, Ω, of a PDE is defined by a `Domain` class object. In the following, Γ will denote the boundary of the domain and $\Gamma^{contact}$ denotes a manifold within the domain where a discontinuity may occur. The two sides of the discontinuity are denoted by side 0 and side 1.

A *factory* class provided by the linear PDE solver library creates an instance of an associated `Domain`. For instance, in the case of a FEM solver, a `Domain` object holds the information about the FEM mesh described through a table of node coordinates and a table of elements. It is emphasised that the way the FEM mesh is described, stored and implemented is not dependent on *escript*. Instead *escript* is dependent on a minimum subset of the information, such as number of cells and number of data points per cell. This information must be provided by the `Domain` through callback functions. For this reason, a PDE solver library can be integrated with *escript*, requiring only a thin interface to map this core functionality.

The following snippet of *Python* script shows how to generate two `Domain` objects from the PDE solver *finley* [1]:

```
import finley
mydomain=finley.Rectangle(ne0=20,ne1=40,l0=1.,l1=2.)
myotherdomain=finley.Mesh("mymesh.file")
```

The object `mydomain` is a rectangular domain with length $l_0 = 1$ in the first spatial dimension and $l_1 = 2$ in the second spatial dimension. The domain is subdivided into a mesh with 20×40 elements. The object `myotherdomain` is created by reading the file `"mymesh.file"` which is provided by an external mesh generator and contains tables of nodes and elements. `Rectangle` and `Mesh` are both *finley* functions returning an *escript* `Domain` object.

The solution of a PDE is a function of its location in the domain of interest Ω. The solution is (piecewise) differentiable but, in general, its gradient is discontinuous. These different degrees of smoothness are reflected in different representations that are used. For instance, in the FEM context the displacement field is represented by its values at the nodes of the mesh, while the strain, which is the symmetric part of the gradient of the displacement field, is stored on the element centers. To be able to classify functions with respect to their smoothness, *escript* has the concept of the "function space" described by objects of the `FunctionSpace` class. The following statement generates the object `solution_space` which is a `FunctionSpace` object and provides access to the function space of PDE solutions on the domain `mydomain`:

```
solution_space=Solution(mydomain)
```

Notice that there is only one type of `FunctionSpace` on a given `Domain`. Any other object returned by `Solution(mydomain)` equals `solution_space`.

The following function space objects are typically used in the context of PDEs:

- `Solution(mydomain)`: solutions of PDEs.

- `ContinuousFunction(mydomain)`: continuous functions, e.g. a temperature distribution. They may have a discontinuity across $\Gamma^{contact}$.
- `Function(mydomain)`: general functions which are not necessarily continuous, e.g. a stress field.
- `FunctionOnBoundary(mydomain)`: functions on the boundary Γ of the domain, e.g. a surface pressure.
- `FunctionOnContact0(mydomain)`: functions on side 0 of the discontinuity $\Gamma^{contact}$.
- `FunctionOnContact1(mydomain)`: functions on side 1 of the discontinuity $\Gamma^{contact}$.

Figure 1 shows the dependency between function spaces. Any solution of a PDE is a continuous function. Any continuous function can be seen as a general function on the domain and can be restricted to the boundary as well as to any side of the discontinuity $\Gamma^{contact}$. The result will be different depending on which side is chosen. Functions on any side of the discontinuity can be seen as a function from the opposite side. A function on the boundary, or on one side of the discontinuity, cannot be seen as a general function on the domain as there are no values defined for the interior. For most PDE solver libraries, the space of the solution and continuous functions are identical, however in some cases, for instance when periodic boundary conditions are used, a solution fulfills periodic boundary conditions while a continuous function may not be periodic.

The concept of a function space describes the properties of functions and allows abstraction from the actual representation of the function in the context of a particular PDE solver. In the FEM context, a function in the `Function` object's function space is typically represented by its values at the element center, but, in a finite difference scheme, the edge midpoint of cells is preferred. Using the concept of function spaces allows the user to run the same script on different PDE solver libraries by just changing the creator of the `Domain` object. Changing the function space of a particular function will typically lead to a change of its representation. So, when seen as a general function, a continuous function which is typically represented by its values on the node of the FEM mesh or finite difference grid has to be interpolated to the element centers or the cell edges, respectively. The function of performing the interpolation is the responsibility of the PDE solver library and is based on that library's data structures for computational efficiency. *escript* does not require specific knowledge of the library's internal implementation.

3. The `Data` Class

Instances of the `Data` class store functions of spatial coordinates and these can be of any function space. The function is represented through its values on sample points where the sample points are chosen according to the function space of the function. The `Data` class objects are used to define coefficients of PDEs on the *Python* level and to store the PDE solutions.

The values of a function have a rank, which specifies the number of indices, and a shape defining the range of each index. The rank in *escript* is limited to the range 0 through 4 and it is assumed that the rank and shape is the same for all sample points. For instance, a stress field has rank 2 and shape (d, d) where d is the spatial dimension. The following statement creates the `Data` object `mydat` representing a continuous function of shape $(2, 3)$ and rank 2:

```
mydat=Data(1.0,(2,3),ContinuousFunction(myDomain))
```

The initial value is the constant 1.0 for all sample points and all components.

The *numarray* package [13] provides a working environment for linear algebra in *Python*, similar to that of MATLAB [8]. Matrices and tensors represented in *numarray* objects or any object, such as a list of floating point numbers, that can be converted into a *numarray* object and used to create `Data` objects. The following two statements create objects which are equivalent to `mydat`:

```
import numarray
mydat1=Data(numarray.ones((2,3)),ContinuousFunction(myDomain))
mydat2=Data([[1,1],[1,1],[1,1]],ContinuousFunction(myDomain))
```

In the first case the initial value is `numarray.ones((2,3))` which generates a 2×3 matrix as a *numarray* array filled with ones. The shape of the created `Data` object is taken from the shape of the array. In the second case, *escript* converts the initial value, which is a list of lists, into a *numarray* object before creating the actual `Data` object.

For convenience *escript* provides creators for the most common types of `Data` objects in the following forms:

- `Scalar(0.0,fSpace)` is the same as `Data(0.0,(,),fSpace)`, for instance a temperature field.
- `Vector(0.0,fSpace))`is the same as `Data(0.0,(d,),fSpace)`, for instance a velocity field.
- `Tensor(0.0,fSpace)` is the same as `Data(0.0,(d,d),fSpace)`, for instance a stress field.
- `Tensor4(0.0,fSpace)` is the same as `Data(0.0,(d,d,d,d),fSpace)` for instance a Hook tensor field.

In these statments `d` defines the spatial dimension and `fSpace` defines a function space, e.g. `fSpace=Function(myDomain)`.

Objects of the `Data` class can be manipulated by applying unitary operations, such as *cos*, *sin* and *log*, and can be combined by applying the common arithmetic binary operations such as $+$, $-$,$*$, $/$. As these operations have to be performed on a very large data sets, the actual work of these operations is not implemented in *Python* but rather in C++. In the current implementation, operations on `Data` objects are executed in parallel using the OpenMP paradigm [6] with data distributions optimized for ccNUMA architectures [7]. Support for other MPI-based parallelization [12] is not difficult to incorporate and is planned currently.

As aforementioned, *escript* itself does not handle any spatial dependencies itself and instead relies on the PDE solver library to provide appropriate functionality such as interpolation. However *escript* invokes interpolation, if required, to resolve `Data` operations. Typically, this occurs in a binary operation when both arguments belong to different function spaces or when data is provided to a PDE solver library, requiring functions to be represented in a particular way.

We now present an example to illustrate the usage of `Data` objects. Assume we have a displacement field, u, and we want to calculate the corresponding stress field, σ, using the linear–elastic isotropic material model

$$\sigma_{ij} = \lambda u_{k,k}\delta_{ij} + \mu(u_{i,j} + u_{j,i}) \tag{3.1}$$

where δ_{ij} is the Kronecker symbol and λ and μ are the Lame coefficients. The following *Python* function, `getStress`, takes the displacement, `u`, and the Lame coefficients, `lam` and `mu`, as arguments and returns the corresponding stress:

```
def getStress(u,lam,mu):
  d=u.getFunctionSpace().getDim()
  g=grad(u)
  stress=lam*trace(g)*kronecker(d)+mu*(g+transpose(g))
  return stress
```

The variable `d` gives the spatial dimension of the domain on which the displacements are defined. `kronecker` returns the Kronecker symbol with indexes i and j running from 0 to `d`-1. In a similar manner to the way that interpolation is undertaken, the gradient calculation ,`grad(u)`, in turn calls a function of the PDE solver library for which `u` is defined. Typically, `u` must be in the `Solution` or `ContinuousFunction` function space. The result, `g`, and stress are in the `Function` function space. If `u` is defined, `getStress` might be called in the following way:

```
s=getStress(u,1.,2.)
```

In the case that the values for `lam` and `mu` are calculated through expressions, such as in the case of a temperature dependency, `getStress` can also be called with `Data` objects instead of floating-point numbers as arguments. The following call is equivalent to the previous example:

```
lam=Scalar(1.,ContinuousFunction(mydomain))
mu=Scalar(2.,Function(mydomain))
s=getStress(u,lam,mu)
```

In this situation processing the `getStress` function becomes more complicated: the `lam` object belongs to the `ContinuousFunction` object's function space but the object returned by `trace(g)` is in the `Function` object's function space. Therefore the evaluation of the product `lam*trace(g)` produces a problem, as both functions are represented differently. In case of FEM, the `lam` object's values correspond to the coordinates of the mesh nodes, and `trace(g)` object's values correspond to the coordinates of the element centers. If the function spaces of arguments in a binary operation are inconsistent, *escript* interprets the arguments in the appropriate function space according to the inclusion defined in Table 1. In this

example, *escript* sees `lam` as a function of the `Function` object's function space and interpolates the `lam` object's values to the coordinates of the element centers. It is noted that this interpolation is performed independently through a function of the underlying PDE solver library.

In the Earth sciences, material parameters such as the Lame coefficients are typically dependent on rock types present in the domain of interest. A common technique to handle these kinds of material parameters, in particular in the case of complex domain geometries, is through "tagging". Figure 2 shows an example of a geological map. In this case, two rock types *white* and *grey* can be found in the domain. The domain is subdivided into triangles. Each triangle has a tag indicating the rock type found in this triangle. Here, tag 1 is used to indicate the rock type *white* and tag 2 for the rock type *grey*. The tags are assigned at the time when the triangles are generated and stored in the `Domain` class object. The following statements show how, for the example of Figure 2 and the stress calculation discussed previously, tagged values are used to define a `lam` function according to a geological map:

```
lam=Scalar(20.,Function(mydomain))
lam.setTaggedValue(1,30.)
lam.setTaggedValue(2,5000.)
s=getStress(u,lam,2.)
```

In this example, `lam` is set to 30 for those triangles with a tag of 1 and to 5000 for those triangles with a tag of 2. The initial value 20 of `lam` is used as a default value for the case when a tag is encountered which has not been linked with a value. Note that the `getStress` method is called without modification. *escript* resolves the tags when `lam*trace(g)` is calculated.

Objects of the `Data` class provide an abstraction from not only from the representation required to represent functions in a particular function space but also their possible representations as constants over the domain, piecewise constant, as represented by tagging samples, or as expanded data when each sample holds an individual value. The latter is computationally intensive and memory demanding when processed. In arithmetic expressions, `Data` objects with different representations and on different function spaces can be used in a single expression. The required transformations of representations are performed by *escript* when the expressions are evaluated. As shown in the example of a stress calculation, this allows users developing and testing algorithms for a simple case (such as with the Lame coefficients being constants) to reapply the algorithm in a more complex application without the need to modify or reimplement it.

4. `linearPDE` Class

A `linearPDE` class object defines a linear, steady, second order PDE for an unknown function u on the domain Ω. To formulate the PDE we use the generalized

flux, J, which in tensor notation is defined by

$$J_{ij} = A_{ijkl}u_{k,l} + B_{ijk}u_k - X_{ij} \tag{4.1}$$

where u_k denotes the k-th component of the function, u, and $u_{,j}$ denotes the derivative of u with respect to the j-th spatial direction. The PDE for the unknown, u, is abstracted through the `linearPDE` class and is defined by:

$$-J_{ij,j} + C_{ikl}u_{k,l} + D_{ik}u_k = Y_i \ . \tag{4.2}$$

The (natural) boundary conditions for the normal component of the flux are considered in the form

$$n_j J_{ij} = y_i - d_{ik}u_k \ . \tag{4.3}$$

where n denotes the outer normal field on the surface of the domain. Discontinuities across $\Gamma^{contact}$ are considered in the form:

$$n_j J^0_{ij} = n_j J^1_{ij} = y^{contact}_i - d^{contact}_{ik}[u]_k \ . \tag{4.4}$$

In this condition, J^0 and J^1 are the fluxes on side 0 and side 1 of the discontinuity $\Gamma^{contact}$, respectively. n is the normal field on the fault pointing away from side 0 and $[u]$ is the jump of u across $\Gamma^{contact}$. Moreover, constraints of the form

$$u_i = r_i \text{ where } q_i > 0 \tag{4.5}$$

can be considered. The constraints (4.5) override any condition set by equations (4.2), (4.3) or (4.4), where ever the characteristic function q is positive. A, B, C, D, X and Y are functions in the `Function` object's function space, y and d are in the `FunctionOnBoundary` object's function space and $y^{contact}$ and $d^{contact}$ are in the `FunctionOnContact` object's function space. The functions r and q are in the `Solution` object's function space. When the solution of a PDE is requested, *escript* passes the PDE to the solver library which is typically implemented in C/C++. As explained above the solver library is defined by the `Domain` of the PDE. The returned solution belongs to the `Solution` object's function space.

In the current version of *escript*, we have implemented an interface to the FEM library *finley* which solves the general type of PDEs defined through the `linearPDE` class, see [1]. The package *finley* is written in C and supports two- and three-dimensional isoparametric, unstructured meshes. Linear or quadratic elements can be used. It is also able to handle contact elements. It is parallelized for ccNUMA architectures, such as the SGI Altix architecture [7], using the OpenMP paradigm [6]. Test runs of *finley* and *escript* show good scalability with up to 200 processors on an SGI Altix 3700. For more details on the parallelization strategies and a discussion on performance of *finley* we refer to [1].

5. Example 1: Lame Equation

The following example demonstrates the application of the `linearPDE` class to define and solve the Lame equation, which is the basis of a large varity of models in geoscience. A more advanced example will be discussed in the next section. We

also refer to [2] which discusses the implementation of a mantle convection model with *escript*. This example requires the user to calculate the displacement field, u, of a three-dimensional block of material which is fixed at its base. The surface is loaded by a pressure, p.

The displacement is given by the equation

$$-\sigma_{ij,j} = 0 \tag{5.1}$$

where the stress tensor, σ, assumes the role of the flux J defined by (3.1). The natural boundary conditions

$$n_j \sigma_{ij} = p_i \tag{5.2}$$

are defined, and, on the bottom of the block, the constraint

$$u_i = 0 \tag{5.3}$$

is applied. To define the PDE, defined by equations (5.1)– (5.3), using the `linearPDE` class we have to set:

$$\begin{array}{ll} A_{ijkl} = \lambda\delta_{ij}\delta_{kl} + \nu(\delta_{ik}\delta_{jl} + \delta_{jk}\delta_{il}) & y_i = p_i \\ q_i = \begin{cases} 1 & x_{d-1} = 0 \\ 0 & \text{otherwise} \end{cases} & r_i = 0 \end{array} \tag{5.4}$$

where $x = (x_i)$ is a location within the domain.

The following *escript* script implements a function which returns, for given Lame coefficients, `lam` and `mu`, and a given surface load `p`, the displacement field as a solution of the PDE defined above:

```
def getDisplacement(domain,lam,mu,p):
   d=domain.getDim()
   x=domain.getX()
   hook=Tensor4(0,what=Function(mydomain))
   for i in range(d)
     for l in range(d)
       hook[i,i,l,l]+=lam
       hook[i,l,i,l]+=mu
       hook[i,l,l,i]+=mu
   myPDE=linearPDE(domain)
   myPDE.setValue(A=hook,y=p,q=whereZero(x[d-1])*numarray.ones(d))
   return myPDE.getSolution()
```

As r is not defined, it is assumed to be zero. This script can be run with the Lame coefficients input as single floating-point numbers or as any scalar `Data` objects with an appropriate function spaces. Similarly, the surface pressure, `p`, could be a vector of floating point numbers, in which case the load is acting on the entire surface, excluding the locations where the constraints are applied. Alternatively, it could be defined through "tagging" as described above. The `getStress` method, introduced in Section 3, could be used to calculate the stress from the returned displacement.

Note that alternatively one can implement `getDisplacement` through the a suitable subclass of `linearPDE` which specializes the general PDE.

6. Example 2: Quasi-Static Fault System Model

To illustrate the usage of *escript* in a more realistic scenario, we discuss the implementation of a stick–slip model applied to a crustal fault system. We adopt a quasi-static model, as proposed by [4], to calculate the displacement field, u, of the domain of interest. To simplify the presentation, we will restrict our attention to the two-dimensional case of a linear–elastic material. The domain of interest is a rectangle which is fixed at the bottom while being both compressed and sheared at the top. There is no restriction on the fault system, nor on the structure of the mesh used for the dicretization.

At any time the displacement has to fulfill the equations (5.1), with a natural boundary similar to (5.2), and constraints (5.3). At the fault $\Gamma^{contact}$ the contact condition

$$f_i = \sigma^0_{ij} n_j = \sigma^1_{ij} n_j \tag{6.1}$$

must hold at all times. The normal contact stress f_i is decomposed in its normal component f_n and its tangential component f_τ:

$$f_i = f_n n_i + f_\tau \tau_i \tag{6.2}$$

where $\tau = (-n_1, n_2)$ denotes the tangential vector on the fault.

The sides facing the fault may not penetrate, i.e. the normal component, $[u]_n$, of the jump, $[u]$, across the fault is non negative:

$$[u]_n := [u]_i n_i \geq 0. \tag{6.3}$$

The normal contact stress, f_n, is chosen to work against penetration by setting

$$f_n = min(E_n [u]_n, 0) \tag{6.4}$$

where E_n is a penalty parameter.

In the tangential direction a stick-slip friction model is used. The contact stress has to meet the yield condition

$$\Phi := |f_\tau| - \mu_d |f_n| \leq 0 \tag{6.5}$$

where μ_d is the dynamic friction coefficient yet to be defined. In the following, t_{event} is the time when the fault changes from the stick state ($\Phi < 0$) to the slip state ($\Phi = 0$) or from the slip state to the stick state. Note that t_{event} is a function of its position on the fault. The tangential dislocation, $[u]_\tau$, and the slip, s, after an event are defined by

$$[u]_\tau := [u]_i \tau_i \text{ and } s = [u]_\tau - [u]_\tau(t_{event}) \; . \tag{6.6}$$

For the stick state ($\Phi < 0$), we set

$$f_\tau = f_\tau^{elast} = E_\tau s + f_\tau(t_{event}) \tag{6.7}$$

where E_τ is a positive constant. This condition forces the fault to maintain its tangential dislocation at the value $[u]_\tau(t_{event})$ after changing from slip to stick. For the slip state ($\Phi = 0$), we set

$$f_\tau = sgn(f_\tau^{elast})\mu_d f_n \tag{6.8}$$

where $sgn(s)$ denotes the sign of argument s. Combining conditions (6.7), (6.8) and (6.5) we obtain

$$f_\tau = sign(f_\tau^{elast}) \cdot max(|f_\tau^{elast}|, \mu_d f_n) \; . \tag{6.9}$$

To define the dynamic friction coefficient μ_d, we use a slip weakening frictional relation

$$\mu_d = \mu_0 + (\mu_s - \mu_0)\left(1 - \frac{max(|s|, D_c)}{D_c}\right) \tag{6.10}$$

where μ_0 the minimum dynamic friction, μ_s the static friction coefficient and D_c is the critical slip distance. In more realistic models, slip weakening given by (6.10) has to be combined with slip-rate weakening, see [3], which is ignored here to simplify the presentation.

To calculate the displacement field, $u^{(n)}$, of the material at time $t^{(n)}$ we use the incremental approach

$$u^{(n)} = u^{(n-1)} + h^{(n)}\dot{u}^{(n)} \tag{6.11}$$

where $u^{(n-1)}$ is the displacement field at time $t^{(n-1)}$, $h^{(n)} = t^{(n)} - t^{(n-1)}$ is the step size and $\dot{u}^{(n)}$ is the velocity field. The step size is chosen to be sufficiently small. For instance, one can choose $h^{(n)}$ such that the relative size $h^{(n)}\frac{\|\dot{u}^{(n)}\|}{\|u^{(n-1)}\|}$ of the displacement increment stays below a given tolerance. We need to formulate a PDE to calculate the velocity $\dot{u}^{(n)}$. In the following the upper index (n) refers to values at time $t^{(n)}$.

By changing to rates one can immediately derive a PDE for the $\dot{u}^{(n)}$:

$$-\dot{\sigma}_{ij,j} = 0 \text{ with } \dot{\sigma}_{ij} = \lambda\dot{u}_{k,k}\delta_{ij} + \mu(\dot{u}_{i,j} + \dot{u}_{j,i}) \tag{6.12}$$

with the natural boundary conditions

$$\dot{\sigma}_{ij}n_j = p_i \tag{6.13}$$

and constraint

$$\dot{u}_i = 0 \tag{6.14}$$

The given pressure function, p, specifies the normal stress rate on the surface. Along the fault, from (6.2) we obtain

$$\dot{f}_i^{(n)} = \dot{f}_n^{(n)}n_i + \dot{f}_\tau^{(n)}\tau_i \tag{6.15}$$

with

$$\dot{f}_n^{(n)} = G^{(n-1)}[\dot{u}^{(n)}]_n \text{ with } G^{(n-1)} = \begin{cases} E_n \; , & [u^{(n-1)}]_n \leq 0 \; , \\ 0 \; , & \text{otherwise} \; . \end{cases} \tag{6.16}$$

In the stick state it is

$$\dot{f}_\tau^{(n)} = E_\tau[\dot{u}^{(n)}]_\tau \tag{6.17}$$

and for the slip state with $\mu_d^{(n-1)} \approx \mu_d^{(n)}$ and $f^{(n-1)} \approx f^{(n)}$ we obtain

$$\dot{f}_\tau^{(n)} = sgn(f_\tau^{el(n-1)} f_n^{(n-1)}) \left[\mu_d^{(n-1)} \dot{f}_n^{(n)} + f_n^{(n-1)} \dot{\mu}_d^{(n)}\right] \tag{6.18}$$

where

$$\dot{\mu}_d^{(n)} = K^{(n-1)}[\dot{u}^{(n)}]_\tau \, , K^{(n-1)} = \begin{cases} -sgn(s^{(n-1)}) \frac{\mu_s - \mu_0}{D_c} \, , & |s^{(n-1)}| < D_c \, , \\ 0 \, , & \text{otherwise.} \end{cases} \tag{6.19}$$

Combining equations (6.16), (6.17), (6.18) and (6.19) the contact condition on the fault is

$$\dot{f}_i^{(n)} = (G^{(n-1)} n_j n_i + H^{(n-1)} n_j \tau_i + J^{(n-1)} \tau_j \tau_i)[\dot{u}_j^{(n)}] \tag{6.20}$$

with $H^{(n-1)} = 0$ and $J^{(n-1)} = E_\tau$ in the stick state and

$$H^{(n-1)} = sgn(f_\tau^{el(n-1)} f_n^{(n-1)}) \mu_d^{(n-1)} G^{(n-1)} \tag{6.21}$$

$$J^{(n-1)} = sgn(f_\tau^{el(n-1)}) |f_n^{(n-1)}| K^{(n-1)} \tag{6.22}$$

in the slip state.

Equation (5.1) with boundary conditions (5.2) and contact condition (6.20) forms a PDE for the increment $\dot{u}^{(n)}$, in a similar manner to way in which the problem of section 5 was treated, except for the fact that now a contact condition has to be included. In addition to equation (5.4) we need to set:

$$d_{ij}^{contact} = G^{(n-1)} n_j n_i + H^{(n-1)} n_j \tau_i + J^{(n-1)} \tau_j \tau_i \tag{6.23}$$

The following *escript* script implements the quasi-static algorithm where some variable initializations have been omitted for brevity:

```
hook=Tensor4(0,what=Function(dom))
for i in range(dom.getDim()):
  for l in range(dom.getDim()):
     hook[i,i,l,l]+=lame_lambda
     hook[i,l,i,l]+=lame_mu
     hook[i,l,l,i]+=lame_mu
pde=LinearPDE(dom)
pde.setValue(A=hook,y=p,q=whereZero(x[d-1])*numarray.ones(d))
side0=FunctionOnContactOne(dom)
side1=FunctionOnContactZero(dom)
n=side0.getNormal()
tau=matrixmult([[0,-1],[1,0]],n)
while t<t_end:
  j=u.interpolate(side1)-u.interpolate(side0)
  j_tau,j_n=inner(j,tau),inner(j,n)
  s=j_tau-j_tau_ev
  mu_d=mu_0+(mu_s-mu_0)*(1-minimum(abs(s),D_c)/D_c)
```

```
f_tau_el=E_tau*s+f_tau_ev
f_n=minimum(E_n*j_n,0)
f_tau=sign(f_tau_el)*minimum(abs(f_tau_el),mu_d*abs(f_n))
stck,stck_old=whereNegative(abs(f_tau)-mu_d*abs(f_n)),stck
ev=abs(stck_old-stck)
j_tau_ev=j_tau*ev+j_tau_ev*(1.-ev)
f_tau_ev=f_tau*ev+f_tau_ev*(1.-ev)
K=-sign(s)*(mu_s-mu_0)/D_c*whereNegative(abs(s)-D_c)
G=E_n*whereNonPositive(j_n)
H=sign(f_tau_el*f_n)*mu_d*G*(1-stck)
J=sign(f_tau_el)*f_n*K*(1.-stck)+E_tau*stck
pde.setValue(d_contact=G*outer(n,n)+outer(H*n+J*tau,tau))
v=pde.getSolution()
h=tol*Lsup(u)/Lsup(v)
u+=h*v
t+=h
```

`mydomain`, `lam`, `mu`, `t_end`, `mu_0`, `mu_s`, `E_n` and `E_tau` constitute the input for this script. When the script is applied to real crustal fault systems, the Lame coefficients are typically defined through a geological database and the friction coefficients are fault dependent. With no modifications, the given script can deal with these cases when the corresponding variables are defined as tagged `Data` objects.

7. Conclusions

escript is a high–level language to implement mathematical models and solution algorithms. It hides the actual representation of data and the transformation between data representations from the user. This allows the development of scripts that, without subsequent modification, can be used for different data representations and discretization methods.

Arithmetic operations and the transformations upon the representations of `Data` objects are implemented in C++. As between PDE solutions operations on `Data` objects are processed on data from the processor cache, sufficient performance is achieved even if complex expressions are evaluated. Moreover, perfect scalability on parallel architectures can be observed as operations on samples are independent. For distributed shared memory architectures, such as the SGI Altix systems [7], the memory manager of *escript*, which reallocates discarded memory, is important to achieve good performance, mainly because the allocation of distributed arrays does not scale well on these systems. Profiling of simulations implemented in *escript* have shown that the compute time is predominantly spent in the PDE solver library, solving PDEs and calculating gradients and interpolations. Typically, the time spend on performing arithmetic operations on `Data` objects stays below 10% of the overall compute time, even if rather complex arithmetic operations have to be executed. In fact, the fraction of time spend in *escript* tends to become smaller

for larger meshes and more processors. We see the overhead of using a scripting rather then compiler-based language as acceptable in particular when considering the much higher productivity that can be achieved with a script based development environment.

Acknowledgment

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

References

[1] M. Davies, L. Gross, H.–B. Mühlhaus, *Scripting High Performance Earth Systems Simulations on the SGI Altix 3700* Proc. 7th Intl Conf. on High Performance Computing and Grid in Asia Pacific Region (2004), 244–251.

[2] M. Davies, H.–B. Mühlhaus, L. Gross: *Thermal Effects in the Evolutioon of Initially Layered Mantel Material*, Pure. Appl. Geophys., submitted 2004.

[3] P. Mora, D. Place, *Simulation of the Frictional Stick-slip Instability*, **143** (1994), 61–87.

[4] H. L. Xing, P. Mora, A. Makinouchi, *Finite Element Analysis of Fault Bend Influence on Stick-Slip Instability along an Intra-Plate Fault* Pure Appl. Geophys. **161** (2004), 2091–2102.

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

[6] R. Chandra, L. Dagum, D. Kohr, D. Maydan, J. McDonald, R. Menon, *Parallel Programming in OpenMP*, Academic Press (2001).

[7] M. Woodacre, D. Robb, D. Roe, K. Feind, *The SGI Altix 3000 Global Shared-Memory Architecture.*, Silicon Graphics (2003).

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

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

[10] L. Gross, *VECFEM- The Solver for Non-linear Partial Differential Equations* in E. N. Houstis, E. Gallopoulos, J.R. Rice, R. Bramley: *Enabling Technologies for Computational Science* Kluwer Academic Publishers (2000).

[11] 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).

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

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

[14] 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: gross@esscc.uq.edu.au

Jonathan Smillie
Earth Systems Science Computational Center
The University of Queensland
St. Lucia., QLD 4072
Australia
e-mail: jgs@esscc.uq.edu.au

Matt Davies
Earth Systems Science Computational Center
The University of Queensland
St. Lucia., QLD 4072
Australia
e-mail: matt@esscc.uq.edu.au

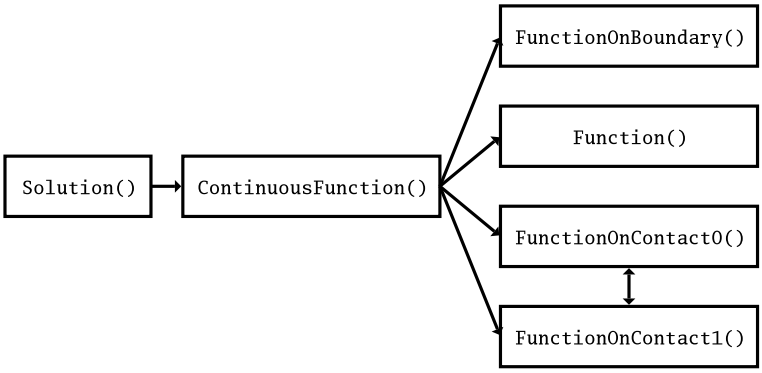

FIGURE 1. Dependency of Function Spaces. An arrow indicates that a function in the function space at the starting point can be interpreted as a function in the function space of the arrow target.

1	1	2	2	1	1	1	1	1	1
1	1	2	2	2	2	2	1	1	1
1	1	2	2	2	2	2	1	1	1
1	1	1	2	2	2	2	2	1	1
1	1	1	2	2	2	2	2	1	1
1	1	1	1	2	2	2	2	1	1
1	1	1	1	1	2	2	1	1	1

FIGURE 2. Element Tagging. A rectangular mesh over a region with two rock types *white* and *grey*. The number in each cell refers to the major rock type present in the cell (1 for *white* and 2 for *grey*).