Interface Modeling in Incompressible Media using Level Sets in *Escript*

L. Gross [a,*] L. Bourgouin [a] A. J. Hale [a] H.–B. Mühlhaus [a,*]

[a] *Earth Systems Science Computational Centre (ESSCC), The University of Queensland, St Lucia, QLD 4072, Australia.*

Abstract

We use a finite element (FEM) formulation of the level set method to model geological fluid flow problems involving interface propagation. Interface problems are ubiquitous in geophysics. Here we focus on a Rayleigh-Taylor instability, namely mantel plumes evolution, and the growth of lava domes. Both problems require the accurate description of the propagation of an interface between heavy and light materials (plume) or between high viscous lava and low viscous air (lava dome), respectively. The implementation of the models is based on *Escript* which is a *Python* module for the solution of partial differential equations (PDEs) using spatial discretization techniques such as FEM. It is designed to describe numerical models in the language of PDEs while using computational components implemented in C and C++ to achieve high performance for time–intensive, numerical calculations. A critical step in the solution geological flow problems is the solution of the velocity-pressure problem. We describe how the *Escript* module can be used for a high–level implementation of an efficient variant of the well–known Uzawa scheme (Arrow et al., 1958). We begin with a brief outline of the *Escript* modules and then present illustrations of its usage for the numerical solutions of the problems mentioned above.

Key words: Partial differential equations, numerical software, level set method, Uzawa scheme, Rayleigh-Taylor instability, mantle plumes, lava dome modeling

* Corresponding author.
Email addresses: l.gross@uq.edu.au (L. Gross), laurent@esscc.uq.edu.au (L. Bourgouin), alinah@esscc.uq.edu.au (A. J. Hale), h.muhlhaus@uq.edu.au (H.–B. Mühlhaus).

1 Introduction

Computational simulations are now firmly established as an effective tool to model and explore virtually all areas of geological fluid mechanics. This coincides with an unprecedented increase in the computational capacity to solve very large and complex problems. Accordingly the expectations in the community in regard to sophistication and model sizes have risen significantly. There is a desire to go beyond the Stokes equations and the heat equation and to consider complex constitutive relationships, fracture, damage, chemical reactions, melting, etc., all translating into additional partial differential equations.

This led to a recent trend in the development of software tools for scientific computing towards high–level user interfaces for lower level numerical libraries, see for instance Drummond et al. (to appear). The objective is to make these libraries easier to use for computational scientists and to provide an environment which is platform independent. The programming language *Python* (Lutz, 2001) is the preferred environment to implement these high–level user interfaces. The main reasons for choosing *Python* are the facts that it is very easy to learn, even for people with very little programming background, it is widely available across all platform and a wast number of *Python* modules have been developed ready to use when developing new applications. Moreover, *Python* is designed as a language to build interfaces to existing libraries developed in C or C++. In fact, there are variety of tools available which allow to make libraries callable for *Python* almost automatically. Although the usage of high–level, interpretive language is bearing the risk of producing inefficient code the practice has shown that, in particular if the reduced development time is taken in consideration, the losses are acceptable, mainly as the computational intensive tasks are still implemented in C/C++ and *Python* is used to steer the calculations only.

High–level user interfaces to numerical libraries still require the user to translate a mathematical model into the language of numerics. For instance, when solving partial differential equations (PDEs) using finite elements (FEM) the user still has to deal with a FEM mesh, sparse matrices and arrays to store the right hand side of a linear system, the solution and PDE coefficients. On the other hand, the user formulates the mathematical model using the terminology of domain, PDE and functions of spatial variables. It is the objective of *Escript* (Davies et al., 2004; Gross et al., to appear) to provide an environment in *Python* that maps the terminology used to describe PDEs–based mathematical models onto a standardised interface into which a PDE solver library written is C or C++ can be linked. The actual spatial discretization technique and its implementation which is provided by the library is hidden from the user. In particular, this allows running an implementation of a model

with different discretization methods. The abstraction from numerical techniques and their implementations dramatically simplifies the development of simulation codes but blocks the user from accessing individual discretization objects such as elements in the FEM mesh. However, this restriction is consistent with the approach of describing models using the PDE terminology which is reasonable only if there is a mesh independence in the sense of convergence towards a mesh independent solution for decreasing mesh size.

In this paper we will discuss the application of *Escript* to model material interfaces between incompressible media with temperature dependent viscosity. This class of problems are the core for many computational models in geosciences. Here we will discuss two applications, namely the formation of plumes in the Earth's mantel and the growth of lava domes. In the case of plume formation, the interface between the heavy material in the Earth's mantel and the light material in the deeper mantel is tracked. When modelling the growth of lava domes the surface of the dome forms a free surface which is treated as an interface between lava and air. These modelling scenarios both contain three algorithmic challenges: the incompressible flow solver, the interface tracking and the advection–diffusion solver for temperature. As we want to apply *Escript* we are restricted in the techniques that can be used to address these challenges, in particular we cannot use special element types and element–based up-winding. In this paper we are proposing to use the inexact Uzawa scheme as an incompressible flow solver, the level set method to track interfaces, and the two–step Taylor–Galerkin scheme to solve advection–diffusion problems. We will show how these three methods can be implemented in *Escript* and are successfully used to model plumes and lava domes.

In the next section we will present the governing equations and the basic idea of the level set method. Section 3 will discuss the solution algorithms that have been chosen, in particular the level set method and the inexact Uzawa scheme used as the incompressible flow solver. In Section 4 we then give a brief overview on *Escript* and discuss some aspects of the implementation of the solution algorithms. Section 5 will show how the model is applied to the plume formation and lava dome growth. In the final section we give a summary and outline some further *Escript* developments.

2 The Model

2.1 Governing Equations

The applications presented in this paper are governed by the Navier-Stoke's equations for velocity v_i and pressure p. We assume a linear relationship be-

tween the deviatoric stress σ'_{ij} and the stretching $D_{ij} = \frac{1}{2}(v_{i,j} + v_{j,i})$ in the form

$$\sigma'_{ij} = 2\eta \, D'_{ij} \ , \tag{1}$$

where η denotes the viscosity and the deviatoric stretching D'_{ij} is defined as

$$D'_{ij} = D_{ij} - \frac{1}{3} D_{kk} \delta_{ij} \ . \tag{2}$$

In this definition δ_{ij} is the Kronecker δ-symbol and Einstein summation tensor notation is used. We restrict ourselves to incompressible flows and neglect the inertia terms.

For gravity acting in the x_3-direction the governing equations are given as

$$-(\eta(v_{i,j} + v_{j,i}))_{,j} - p_{,i} = -g \ \rho \delta_{i3} \ , \tag{3}$$

with the incompressibility condition

$$-v_{i,i} = 0 \ . \tag{4}$$

The coefficients ρ and g denote the density and the gravity constant, respectively. In equations (3) and (4) $f_{,i}$ denotes the derivative of the function f with respect to x_i.

The temperature field T is calculated solving the heat equation

$$\rho c_p (T_{,t} + v_i T_{,j}) - (\kappa T_{,i})_{,i} = Q \ , \tag{5}$$

where c_p is the heat capacity and κ is the thermal diffusivity. The heat source Q is given from viscous dissipation:

$$Q = \tau \, \dot{\gamma} \text{ with } \tau = \sqrt{\frac{1}{2} \sigma'_{ij} \sigma'_{ij}} \text{ and } \dot{\gamma} = \sqrt{2 D'_{ij} D'_{ij}} \ . \tag{6}$$

In general, the viscosity may depend on temperature and pressure but for the following discussion we assume that the temperature dependence takes the form

$$\ln \frac{\eta}{\eta_{min}} = \frac{E_{act} \, T_{melt}}{T} \ , \tag{7}$$

where E_{act} is the activation energy, T_{melt} is the melting temperature and η_{min} is the minimal viscosity.

A more complex, imperical form is used in Section 5.2. We also neglect the temperature–dependency of the density but this effect can be included easily into the framework we will present.

The solution of the temperature equation (5) as well as the velocity-pressure equations (3) and (4) requires appropriate boundary conditions which for the the sake of a simpler presentation are not discussed here.

2.2 The Level Set Method

We assume that the two sub-domains Ω_0 and Ω_1 of the domain are filled with two different material with distinct values for material parameter such as viscosity η and density ρ. To describe the interface Γ between the two sub-domains we use the level set method. It is based upon an implicit representation of the interface Γ by a smooth, scalar function ϕ which is called the level set function. The function usually takes the form of a signed distance to the interface, whereby the zero level surface $\phi(x) = 0$ represents the points x on the actual interface Γ between the two materials. Points x in Ω_0 can be characterised by $\phi(x) < 0$ while points x in Ω_1 can be characterised by $\phi(x) > 0$, or vice versa. At a given location in the domain the value of any parameter can be set depending upon the sign of ϕ at that location.

In the presence of a velocity field v_i the interface Γ is transformed over time. In the Eulerian framework this is described by the advection equation:

$$\phi_{,t} + v_i \phi_{,i} = 0 \ . \tag{8}$$

It is desirable that over time the function ϕ maintains its initial character as a distance function. This can be expressed by the normalisation condition

$$\phi_{,i} \phi_{,i} = 1 \ . \tag{9}$$

In the next section we will discuss appropriate algorithms to deal with the time-depended problems (5) and (8), the normalization condition (9) and the saddle point problem (3)-(4).

3 Algorithms

3.1 Two–Step Taylor–Galerkin Scheme

The equations (5) and (8) are written as the advection-diffusion equation

$$c \cdot u_{,t} = G(u) = f - v_i u_{,j} + (\kappa u_{,i})_{,i}, \tag{10}$$

where for the temperature equation (5) we set $u \leftarrow T$, $c \leftarrow \rho c_p$, $v \leftarrow \rho c_p v$, $f \leftarrow Q$ and $\kappa \leftarrow \kappa$, and for the advection of the level set function (8) we set $u \leftarrow \phi$, $c \leftarrow 1$, $v \leftarrow v$, $f \leftarrow 0$ and $\kappa \leftarrow 0$.

For time integration we use the Taylor-Galerkin scheme (Zienkiewicz & Taylor, 2000) which is of third order accuracy in time. If u^- and u^+ are the solution u at the current and next time step and dt denotes the time-step size the scheme can be written in the following form: First for the given solution u^- at time t^- one solves

$$c \cdot (u^{1/2} - u^-) = \frac{dt}{2} G(u^-). \tag{11}$$

to calculate the solution $u^{1/2}$ at the mid point $t^{1/2} = \frac{1}{2}(t^+ + t^-)$. Then, using $u^{1/2}$, one calculates the solution u^+ at the next time step t^+ by solving

$$c \cdot (u^+ - u^-) = dt\ G(u^{1/2}). \tag{12}$$

In a practical implementation it is assumed that the velocity is constant for the entire time step, this means the velocity field is not updated when $G(u^{1/2})$ is calculated.

In the case of a divergence free velocity field v_i, this scheme is equivalent to the classical one–step formulation for the pure advective case:

$$c \cdot (u^+ - u^-) = dt \left(\hat{f} - \frac{dt}{2c} (v_i \hat{f})_{,i} \right) \text{ with } \hat{f} = f - v_i u^-_{,i} \ . \tag{13}$$

The one–step scheme (13) and each step of the two–steps scheme (11)-(12) requires the same type of update operation but with a slightly different expression for the second order spatial differential term.

To maintain stability for the Taylor–Galerkin scheme the time step size has to fulfill the Courant condition which can be written in the form

$$dt \leq \zeta_{dt} \frac{dx^2\ c}{dx\ \sqrt{v_i v_i} + |\kappa|}\ . \tag{14}$$

This condition must be met over the entire problem domain. The value dx is the local length scale, for instance the local element size in a finite element mesh. The factor ζ_{dt} is a safety factor which is dependent on the spatial discretization method only and is typically determined experimentally.

The one–step scheme is preferred over the two–step scheme since typically it is less expensive. However, the preferred scheme ultimately depends upon the spatial discretization method. When using the finite element method the calculations at each step require the solution of a system of linear of equations with the mass matrix. The computational costs can be minimised by using lumping of the mass matrix. This however can lead to instabilities. In the presence of a diffusion term, i.e. $\kappa \neq 0$, the two step scheme is the better option since the corresponding one–step scheme requires a fourth order spatial derivative which is difficult to construct in most spatial discretization schemes. In the following we will use the two–step scheme to solve the advection–diffusion problems for temperature and the advection of the level set function, but use one–step scheme for the reinitialisation of the level set function.

3.2 Reinitialisation

It is clear that in general the normalisation condition (9) of the level set function ϕ is not preserved during the advection process. It has however been shown, see Sussman et al. (1994), that, in order to obtain acceptable conservation of mass, it is critical that ϕ remains a distance function in regions close to any interface. Therefore, a reinitialiseialisation procedure is applied that transforms ϕ back into a distance function ψ but maintains the location of the interface.

We are using the following, computationally very light approach, see Sussman et al. (1994): With the artificial time t' we are solving the initial value problem

$$\psi_{,t'} = \text{sign}(\phi)(1 - \sqrt{\psi_{,i}\psi_{,i}}) \text{ and } \psi(0) = \phi, \tag{15}$$

until the steady state is reached. The solution at the steady state will have the same zero level set as ϕ and meet the normalisation condition (9). Equa-

tion (15) can be rewritten as, see Tornberg & Engquist (2000):

$$\psi_{,t'} + w_i \psi_{,i} = \text{sign}(\phi) \text{ with } w_i = \text{sign}(\phi) \frac{\psi_{,i}}{\sqrt{\psi_{,i}\psi_{,i}}} \ . \tag{16}$$

Physically, equations (16) can be interpreted as the propagation of information away from the interface at the speed of w_i which is a unit vector normal to the interface and is pointing away from it.

Equation (16) is solved using the one–step Taylor–Galerkin scheme (13) where one sets

$$\hat{f} = \text{sign}(\phi)(1 - \sqrt{\psi_{,i}\psi_{,i}}) \ , \tag{17}$$

If a suitable norm of $\hat{f}$ is sufficiently small the steady state is reached. In practise, it not required to update the velocity w_i as it is only used to stabilise the time integration scheme. Note that the Courant condition (14) takes the simple form

$$dt' \leq \zeta_{dt} \ dx. \tag{18}$$

The reinitialiseialisation only needs to be performed when the level set function ϕ starts losing its distance function property typically, after five time steps.

When the new distance function is found, the physical parameters are updated using the sign of the level set function ϕ. In practise, if the values of a parameter show a large contrast, the jump across the interface must be smoothed. The following procedure is used, to smooth the model parameter χ:

$$\chi = \begin{cases} \chi_0 & \text{where } \psi < -dx \\ \chi_1 & \text{where } \psi > dx \\ \dfrac{\chi_0 - \chi_1}{2\ dx}\ \psi + \dfrac{\chi_0 + \chi_1}{2} & \text{where } |\psi| < dx \end{cases} \ . \tag{19}$$

This has the effect that the physical parameters χ is smoothed across the interface on a band of width $2 \cdot dx$. In the case of a finite element discretization this corresponds to a layer on one elements around the interface. The smoothing procedure prevents numerical instabilities due to high material parameter gradients across a single element.

The Uzawa scheme (Arrow et al., 1958) is used to solve the momentum equation (3) with the secondary condition (4) of incompressibility. The scheme is iterative and is based on the idea that for a given pressure p^- the momentum equation (3) can be used to calculate a velocity v^+. Initially v^+ does not meet the incompressibility condition and its divergence is used to calculate an increment Δp for the pressure as (Cahouet & Chabard, 1988)

$$\frac{1}{\eta}\Delta p = v^+_{i,i} \ . \tag{20}$$

The new pressure p^+ given as

$$p^+ = p^- + \Delta p \ , \tag{21}$$

is now fed back into the momentum equation (3) to calculate a new, improved velocity approximation. The iteration is completed if the relative size of the pressure increment in the L^2–norm is smaller than a given, positive tolerance `TOL`:

$$\|\Delta p\|_2 \leq \ \texttt{TOL} \ \|p^+\|_2 \text{ with } \|p\|_2^2 = \int p^2 \, dx. \tag{22}$$

This criterion can be problematic in practise because slow convergence triggers termination. However, for the purpose of the paper this criterion is sufficient.

A fundamental drawback of the Uzawa scheme as presented above is the fact that the new velocity v^+ is calculated accurately although one can expect that the incompressibility condition will not be fulfilled. It is therefore reasonable to calculate an update increment Δv of the current velocity approximation v^-. The increment is given as the solution of

$$-(\eta(\Delta v_{i,j} + \Delta v_{j,i}))_{,j} = F_i + (\eta(v^-_{i,j} + v^-_{j,i}))_{,j} + p^-_{,i} \ , \tag{23}$$

with external force

$$F_i = -g \cdot \rho \delta_{i3} \ . \tag{24}$$

Then one sets

$$v^+_i = v^-_i + \Delta v_i \ , \tag{25}$$

before updating the pressure. This scheme is equivalent to the classical Uzawa scheme if the incremental momentum equation is solved exactly. However, in the inexact Uzawa scheme (Bramble et al., 1997), it is sufficient to solve the incremental momentum equation (23) inexactly in the sense that a low convergence tolerance is applied in the iterative solver and/or a simplified but robust form of the left hand side operator is used. This scheme still produces fast convergence in pressure and velocity.

There are various options to modify improve efficiency and robustness of the inexact Uzawa, for instance by simplifying the left hand side operator in the equation for the velocity increment as mentioned above, and by introducing relaxation (Hu & Zou, 2001). For this paper we will restrict ourselves to the simple version presented here but want to point out that these modification are useful for the problem class discussed in this paper and that they can be implemented in the environment presented in the next section.

3.4 Work Flow

The algorithm to implement the temperature–dependent, incompressible flow with interface described by a level set function can be outlined as follows:

0. until time integration is completed:
0.0. reinitialise level set function ϕ from (15) by one–step scheme (13).
0.1. Update model parameter using level set function ϕ and smoothing (19).
0.2. Calculate new time step size dt from (14).
0.3. Until stopping criterion (22) is met:
  0.3.0. Update velocity v by solving equation (23).
  0.3.1. Update pressure p by solving equation (20).
0.4. Update temperature T from (5) by two–step scheme (11)–(12).
0.5. Update level set function ϕ from (8) by two–step scheme (11)–(12).
0.6. go to next time step.

Note that the time step size control needs to take in consideration the fact that two advection–diffusion problems, namely for the temperature and the level set function, is integrated over time. Consequently, the minimum of the upper time step size bounds required by each of the problems has to be used when performing the time step.

4 Implementation

In this section we discuss how *Escript* (Gross et al., to appear) is used to implement the models and solution algorithms presented above. Embedded into *Python* (Lutz, 2001) it provides an environment to implement mathematical models that are based on partial differential equations (PDEs). The functionality of *Escript* does not include PDE solver capabilities as such but provides an interface to PDE solver libraries. This approach achieves a high degree of reusabilty of mathematical model implementation because the code can be run with various spatial discretization techniques as well as different implementation approaches without modifications to the code.

In *Escript* the domain of a PDE is described by a `Domain` class object. The following *Python* script creates the `Domain` class object `dom` which is the unit cube discretized by a $10 \times 10 \times 10$ grid for the PDE solver library *Finley* (Davies et al., 2004):

```
from finley import Brick
dom=Brick(10,10,10)
```

From this code it becomes clear that a `Domain` class object does not only contain information about the geometry of the domain but also about the library that will be used to solve the PDEs. Implicitly this also sets the spatial discretization method. In the case of *Finley* this is the finite element method (FEM).

The `LinearPDE` class object defines a general, second order, linear PDE over a domain represented by a `Domain` class object. The general form of the PDE for an unknown vector-valued function u_i represented by the `LinearPDE` class is

$$-(A_{ijkl}u_{k,l} + B_{ijk}u_k)_{,j} + C_{ikl}u_{k,l} + D_{ik}u_k = -X_{ij,j} + Y_i \ . \tag{26}$$

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

$$n_j \ (A_{ijkl}u_{k,l} + B_{ijk}u_k) + d_{ik}u_k = n_j \ X_{ij} + y_i \ , \tag{27}$$

can be defined where n_j defines the outer normal field of the boundary of the domain and y and d are given functions. Notice that A, B and X are already used in the PDE (26). To set values of u_i to r_i on certain locations of the domain one can define constraints of the form

$$u_i = r_i \text{ where } q_i > 0 \ , \tag{28}$$

where q_i is a given function defining through a positive value the locations where the constraint is applied.

In case of a scalar solution u the PDE takes the form:

$$-(A_{jl}u_{,l} + B_j u)_{,j} + C_l u_{,l} + D\ u = -X_{j,j} + Y\ , \tag{29}$$

with natural boundary conditions of the form

$$n_j\ (A_{jl}u_{,l} + B_j u) + d\ u = n_j\ X_j + y\ , \tag{30}$$

and constraints of the form

$$u = r \text{ where } q > 0\ . \tag{31}$$

The usage of is `LinearPDE` class object is illustrated by solving the Helmholtz problem

$$-u_{,ll} + u = 1 \text{ with } n_j\ u_{,j} = 0\ . \tag{32}$$

The following script defines and solves the Helmholtz problem:

```
from escript import LinearPDE, kronecker
pde=LinearPDE(dom)
pde.setValue(A=kronecker(dom),D=1.,Y=1.)
u=pde.getSolution()
```

The function `kronecker` returns the Kronecker δ–symbol. Notice, that no special settings for the boundary conditions is required. The object `u` which holds the solution is an *Escript* `Data` object. In *Escript* `Data` object are used to represent spatial function defined on a `Domain`.

A closer inspection of the algorithms of Section 3 show that each time or iteration step requires the solution of a linear PDE. However, unlike in the simple Helmholtz problem the coefficients are no longer constants but become expressions involving functions of spatial coordinates. The *Escript* core library provides the necessary functionality to evaluate the expressions defining the PDE coefficients in *Python* and to prepare the coefficients such that they can be handed over to the PDE solver library. This will discussed in more details in the following. It is pointed out that for the sake of a simpler presentation boundary conditions are ignored and in some example scripts not all variables are initialised.

First we look into the implementation of Uzawa scheme as presented in Section 3.3. The following function solves the PDE (23) for the velocity increment Δv and returns the result:

```
def getdV(v,p,eta,F):
  v_pde=LinearPDE(v.getDomain())
  k=kronecker(v.getDomain())
  kxk=outer(k,k)
  v_pde.setValue(A=eta*(swap_axes(kXk,1,3)+swap_axes(kXk,0,3), \
                 X=-eta*symmetric(grad(v))-p*kronecker(dom), \
                 Y=F)
  v_pde.setTolerance(1.e-2)
  dv=v_pde.getSolution()
  return dv
```

The arguments are the current velocity `v` and the current pressure `p`, the viscosity `eta` and the external force `F`. It is assumed that `v` and `p` are `Data` objects. The `getDomain` method allows extracting their `Domain`.

Some care has to be taken with the selection of the smoothness of velocity and pressure. In fact, the representations have to meet the LBB condition (Girault & Raviart, 1986) which requires to use a lower approximation order for the pressure. In *Escript* this is reflected through the fact that although defined on the same `Domain` the initial value for `v` and `p` are created with two different smoothness attributes:

```
v=Vector(0.,Solution(dom))
p=Scalar(0.,ReducedSolution(dom))
```

The argument `Solution` indicates a PDE solution with full approximation order while argument `ReducedSolution` a PDE solution with reduced approximation order. In the context of second order FEM, the velocity is approximated by continuous, piecewise quadratic splines while the pressure is approximated by continuous, piecewise linear splines on the same elements. Typically the pressure is represented by its values on element vertices while for velocity additionally the values at edge midpoints are used.

In the light of the concept of smoothness the calculation of the PDE coefficient `X` needs special attention. When using finite elements the result of a gradient calculation is stored by its values on the quadrature points in the interior of each element as the gradient may be discontinuous. In *Escript* this is reflected by the fact that the `Data` class object returned by the gradient function `grad` is defined on the same `Domain` but with the smoothness attribute `Function`

to indicate the fact that a representation of the function is used which is different from the representation used for PDE solutions. The smoothness attribute of the gradient is passed through to the result of the expression `-eta*symmetric(grad(v))` (we assume that `eta` is just a real scalar). Then there is aconflict with the smoothness attribute `ReducedSolution` of the term `-p*kronecker(dom)` when finally calculating the PDE coefficient `X`. For the FEM the first term is stored on quadrature point while the second on element vertices. Consequently, the addition cannot performed directly but interpolation has to be applied before hand. In fact, *Escript* is detecting this mismatch of smoothness attributes and calls the interpolation functions provided by the `Domain` of the arguments.

As temperature is a solution of a PDE it has the `Solution` smoothness attribute, the temperature-depending viscosity `eta` defined by equation (7) will also have the smoothness attribute `Solution`. As discussed before *Escript* will preform an interpolation of the viscosity `eta` when the PDE coefficient `X` is calculated. Similarly, as the PDE solver library will expect the coefficient `Y` with smoothness attribute `Function` *Escript* will invoke, if required, the interpolation of the external force `F` before the coefficient is passed on to the PDE solver. It is pointed out that the function `getdV` can be used with a viscosity `eta` being a floating point number as well as an *Escript* `Data` class object with an arbitrary smoothness attribute. The external force `F` can be given as a *Python* list of floating point numbers, a `numarray` object (Greenfield et al., 2002) or an *Escript* `Data` class object. Behind the scenes *Escript* performs the necessary conversions.

Typically, the PDE solver library will use an iterative method to solve a discrete version of the PDE. The method `setTolerance` of the `LinearPDE` class sets the tolerance for the solver. This is the tolerance for a given discretization and does not consider the discretization error. In the given example of the inexact Uzawa scheme we need the velocity increment with low accuracy only and therefore we require the solution of the PDE to be returned with a relative accuracy of 0.01. It requires only a few iteration steps in the PDE solver to meet this criterion.

Similar to the calculation of the velocity increment we can now easily implement the function `getdP` which returns the pressure increment by solving (20):

```
def getdP(v, eta):
  p_pde=LinearPDE(v.getDomain())
  p_pde.setReducedOn()
  p_pde.setTolerance(1.e-2)
  p_pde.setValue(D=1./eta,Y=div(v))
  dp=p_pde.getSolution()
```

```
  return dp
```

Although we do not solve a PDE in the strict sense, we are using the `LinearPDE` class to project the discontinuous divergence of the velocity onto a continuous function. The call of the method `setReducedOn` makes sure that the PDE returns a solution with the smoothness attribute `ReducedSolution` as required to meet the LBB condition. Similar to the velocity increment the PDE is solved with an accuracy of 0.01 only.

Finally the inexact Uzawa scheme can be implemented with following script which returns a new velocity `v` and new pressure `p` with tolerance `TOL` for the pressure:

```
def incompressibleFlow(v, p, eta, F, TOL=1.e-5):
  while L2(dp) > TOL*L2(p):
    v+=getdV(v, p, eta, F)
    dp=getdP(v, eta)
    p+=dp
  return v,p
```

The *Escript* function `L2` is calculating the L^2–norm.

4.2 Reinitialisation

With the one–step Taylor-Galerkin scheme (13) and (17) it is simple to implement the reinitialiseialization of the level set function. The following function `reinitialise` implements the algorithm:

```
def reinitialise(phi, TOL=1.e-5):
  sgn_phi=sign(phi)
  v=normalize(grad(phi))
  dt=0.5*inf(phi.getDomain().getSize())
  pde=LinearPDE(phi.getDomain())
  pde.setValue(D=1.)
  pde.setSolverMethod(pde.LUMPING)
  while L2(f_hat)>TOL:
      pde.setValue(X=dt/2*inner(v,f_hat),Y=f_hat)
      phi+=pde.getSolution()*dt
      f_hat=sgn_phi*(1-length(grad(phi)))
  return phi
```

The argument `phi` gives the current level set function while the function returns the reinitialised level set function describing the same interface then the input. The pseudo–time integration stops if L^2-norm of the defect of the

normalization condition (9) is less than the given tolerance `TOL`.

The method `getSize` of the `Domain` class returns the local length scale, that is the element diameter in case of the finite element method. The pseudo–time step size is determined from condition (18) with $\zeta_{dt} = 0.5$. Similar to the pressure update calculation we are solving a PDE to calculate the level set function increment. As we are interested in the steady state solution lumping of the stiffness matrix is switched on by the call of the `setSolverMethod` call. An instance of the `LinearPDE` is created outside the pseudo–time integration loop and all constant coefficients are set before the loop is entered. This way computational work, for instance the lumping of the stiffness matrix, has to be done once only and values can be reused during iteration.

Equation (19) defines a method to set a model parameter using a level set function. The following *Python* function implements this method where the argument `phi` is a a level set function and `chi0` and `chi1` are the parameter value in regions of negative values and positive values of the level set function:

```
def getParameter(phi, chi0, chi1):
    dx = phi.getDomain().getSize()
    mask0  = whereNonNegative(-phi-dx)
    mask1  = whereNonNegative( phi-dx)
    mask01 = whereNegative(abs(phi)-dx)
    chi = chi0*mask0 + \
          chi1*mask1 + \
          ((chi0-chi1)/(2*dx)*phi+(chi0+chi1)/2)*mask01
    return chi
```

This implementation illustrate the technique of masking regions of the domain. The `whereNonNegative` returns a *Escript* `Data` class object which has the same `Domain` and smoothness attribute as its argument. The value is 1 in regions where the argument has a non–negative value and the value 0 elsewhere. In the context of finite elements, the local length scale `dx` will be represented at quadrature points, that means it has the smoothness attribute `Function`. Therefore as `phi` will be interpolated the masks `mask0`, `mask1` and `mask01` inherit this attribute and consequently the returned parameter `chi` has the smoothness attribute `Function`.

4.3 Advection–Diffusion Problem

Suppose we have an implementation of the function G the two–step scheme (11)-(12) is implemented in the following way:

```
def stepAdvectionDiffusion(dt, u, kappa, v, c=1, Q=0):
```

```
  u_half=u+dt/2.*G(u, kappa, v, c, Q)
  u=u+dt*G(u_half, kappa, v, Q)
  return u
```

To implement the function G the `LinearPDE` is used:

```
def G(u, kappa, v, c, Q):
  pde=LinearPDE(u.getDomain())
  g=grad(u)
  pde.setValue(D=c,X=kappa*g,Y=Q-inner(v,g))
  return pde.getSolution()
```

To speed–up the calculation one can use lumping in the evaluation of G.

4.4 Simulation

Finally the components which have been discussed are put together where the temperature–dependent viscosity and the external force are calculated using (7) and (24). Again dropping the initialisation phase the scripts takes the following form where as example we assume that the level set function defines two region with different melting temperatures T_{melt}:

```
while t<t_end:
    phi=reinitialise(phi)
    T_melt=getParameter(phi,T_melt0, T_melt1)
    eta=eta_min*exp(E_act*T_melt/T)
    v,p=incompressibleFlow(v, p, eta, -g*rho*[0,0,1])
    D=symmetric(grad(v))
    dev_D=D-trace(D)/3*kronecker(3)
    dt=0.5*inf(dx**2*/(dx*length(v)+abs(kappa)/rho/c_p))
    T=stepAdvectionDiffusion(dt,T,kappa, rho*c_p*v, rho*c_p, \
                             2*eta*length(dev_D)**2)
    phi=stepAdvectionDiffusion(dt,phi,0,v)
    t+=dt
```

The implementation presented here is not the best possible as each of the subtasks creates a new instance of the `LinearPDE` class in every time or iteration step. As a `LinearPDE` class objects manages various structures such as the stiffness matrix pattern, the matrix entries and preconditioners it is more efficient to keep instances of the `LinearPDE` class and to reset modified PDE coefficient. Implementing subtasks as classes rather then functions is an elegant way to achieve this. In this case the `LinearPDE` class objects can be stored as instance variables and an update step is implemented as a class method. As the object driven implementation requires more lines of program code we

have restricted the presentation to the simpler but less efficient approach of using function calls.

5 Applications

In this section we show how the model implementation presented in the previous section is applied to the formation of plumes and the growth of lave domes. The results that are shown are from simulation runs using the OpenMP version of *Escript* and *Finley* on an SGI Altix 3700 system.

5.1 Plumes in the Earth Mantle

Plumes are defined as upwelling in the mantle driven by their own buoyancy and having the form of a large spherical head roughly 1000 km in diameter and with a narrower tail 100–200 km in diameter. They arise from the mode of convection that cools the Earth's interior and drives plate tectonics. Plumes are active columnar upwelling, originating at hot boundary layers. They move laterally an order of magnitude slower than tectonic plates, thus generating an age-progressive chain of volcanism on the overriding plate. The basic thermal plume model has been extended to explore, for example, the effects of mantle viscosity stratification (Leicht et al., 2001), compositional layering at the source (Farnetani, 1997), mixing within a plume (Farnetani et al., 2002), entrained compositional buoyancy and melting at the top of the mantle (Leicht et al., 2001, 1998).

Figure 1 shows the evolution of a hot plume rising from the mantle in a box of dimension 2000 km $\times$2000 km $\times$3000 km . Initially the domain is layered by two materials: The top layer representing the Earth mantle and the bottom layer representing the hot, deeper mantle. The density and viscosity contrasts between the two layers manifest the chemical discontinuity. The temperature dependence of density and viscosity is neglected. The level set method tracks the interface between the two layers. To initiate the plume growth, an initial perturbation is prescribed in the center of the bottom layer. The model is able to reproduce realistic plume shapes. As observed in the Earth's mantel the head of the plume is roughly 1000 km in diameter and the tail about a few hundreds kilometers.

The techniques described in this paper can be applied to model lava dome growth. Here we briefly outline the usage of the presented techniques and refer to Hale et al. (in print) for more details and comparisons to the arbitrary Lagrangian–Eulerian method (ALE) used by Hale & Wadge (2003).

Figure 2 shows the geometrical set–up for an axisymmetric lava dome growth model. The lava is extruded from a conduit inlet onto the volcano surface. The conduit inlet acts as the inlet for magma flow to drive the growth of the lava dome either by prescribing a velocity or pressure boundary condition. The free–surface of the lava dome is defined as the interface between the lava and the embedding–medium air. The two subregions are modelled using the same set of equations but with different values for viscosity and density. The level set method is used to describe this interface.

The viscosity η is given in the form

$$\eta = \theta \cdot \eta_m \ , \tag{33}$$

where the factor θ considers the effects of a constant crystallinity. The melt viscosity η_m of the magma is calculated using an empirical equation, see Hess & Dingwell (1996):

$$\ln \eta = -3.545 + 0.417 \cdot \ln\left(C_f^2 p\right) + \frac{9601 - 1184 \cdot \ln\left(C_f^2 p\right)}{T - 195.7 + 16.13 \cdot \ln\left(C_f^2 p\right)} \ , \tag{34}$$

where $C_f = 4.11\ 10^{-6} Pa^{-1}$ is the solubility coefficient.

Figure 3 shows an evolving lava dome in three-dimensions. The cut–away region shows the interior temperature of the lava dome. It shows an increase in temperature from shear heating above the conduit exit. The level-set method has been proven to be a technique robust enough to model the free-surface of a growing lava dome. This technique is also computationally very light. Moreover, it does not require an initial above-ground free-surface from which the dome can be grown. As observed in Hale & Wadge (2003) assuming an initial lava free-surface shape can influence the final shape and evolution of the dome, whereas the application of the level set method avoids this complication of finding a suitable initial configuration.

6 Conclusion

The *Escript* module provides an environment that allows scientists to quickly implement complex, coupled PDE–driven mathematical models. We have illustrated this for the example of an temperature dependent, flow of two incompressible media. The sample script presented in Section 4.4 show how the different, independent sub–models are put together as a coupled model. In practice, the script can be easily extended by additional sub–models, for instance the advection of chemical species and chemical reactions. Coupling between sub–models is introduced through the fact the output of some models is used as the input of other models.

To simplify the coupling simple model components *Escript* provides the `modelframe` environment. The basic idea is to implement each model as *Python* class which follows a prescribed work flow. Together with a mechanism to link parameters between models the fact that all models of a simulation apply the same work flow allows coupling models without modifying any of the model codes. If the models classes are available in libraries, a simulation can now be described by its models, the values of model parameters, the links between model parameters and the order in which the models are executed. This information can be captured in an XML file which then can be used to build the *Python* script for the simulation. The simulation description file can be created from a graphical user interface. The necessary infrastructure is currently under development.

Acknowledgment

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

References

Gross, L., Cumming, B., Steube, K. & Weatherley, D., A Python Module for PDE-based Numerical Modelling in Kagstrom, B. & Elmroth, E., *PARA'06 Proceedings*, Springer–Verlag (Berlin, to appear).

Sussman, M., Smereka, P. & Osher, S., A Level Set Approach for Computing Solutions to Incompressible Two-Phase Flow, 1994, J. of Comput. Physics 114, 146–159.

Tornberg, A.-K. & Engquist, B., A finite element based level-set method for multiphase flow applications, 2000, Comput. and Visual. in Sci. 3, 93–101.

Arrow, K., Hurwicz, L., & Uzawa, H., *Studies in Nonlinear Programming*, Stanford University Press (Stanford, 1958).

Bramble, J.H., Pasciak, J.E. & Vassilev, A.T., Analysis of the inexact Uzawa Algorithm for saddle point problems, 1997, SIAM J. Numer. Anal. 34, 1072–1092.

Cahouet, J. & Chabard, J.P., Some fast 3D finite element solver for the generalized Stokes problem, 1988, Int. J. Numer. Meth. Fluids 8, 869–895.

M. Lutz, *Programming Python*, 2nd edition, O'Reilly (2001).

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

Hu, Q. & Zou, J., An iterative method with variable relaxation parameter for saddle–point problems, 2001, SIAM J. Matrix Anal. Appl. 23. 317–338.

Zienkiewicz, O.C. & Taylor, R.L., *The Finite Element Method; Volume 3: Fluid Mechanics*, 5th edition, Butterworth-Heinemann (2000).

Girault, V., & Raviart, P.–A., *Finite Element Methods for Navier–Stokes Equations*, Springer–Verlag, (Berlin, 1986).

Greenfield, P., Miller, J.T., Hsu, J., & White., R.L., An Array Module for Python, in Bohlender, D.H., Durand, D., & Handley, T.H., *Astronomical Data Analysis Software and Systems XI*, Astronomical Society of the Pacific (San Francisco, 2002).

Leicht, A.M., Davies, G.F., Wells, M., A plume head melting under a rifting margin, 1998, Earth Planet. Sci. Lett. 161, 161–177.

Leicht, A.M. & Davies, G.F., Mantle plumes from flood basalts: Enhanced melting from plume ascent and an eclogite component, 2001, J. Geoph. Res. 106, 2047–2059.

Farnetani, C.G., Excess temperature of mantle plumes: the role of chemical stratification across D", 1997, Geoph. Res. Lett. 24, 1583–1586.

Farnetani, C.G., Legras, B. & Tackley, P.J., Mixing and deformations in mantle plumesi, 2002, Earth Planet. Sci. Lett. 196, 1–15.

Hale, A.J. & Wadge, G., Numerical modeling of the growth dynamics of a simple silicic lava dome, 2003, Geophys. Res. Letts. 30(19).

Hale H.J., Bourgouin, L., & Mühlhaus, H.–B., Using the Level-Set Method to Model Endogenous Lava Dome Growth, in print, J. Geophys. Res.

Hess K.U. & Dingwell, D.B., Viscosities of hydrous leucogranitic melts: A non-Arrhenian model, 1996, Am. Mineral. 81, 1297–1300.

Drummond, L., Galiano, V., Migallon, V. & Penades, J., High-level User Interfaces for the DOE ACTS Collection, in Kagstrom, B. & Elmroth, E., *PARA'06 Proceedings*, Springer–Verlag (Berlin, to appear).

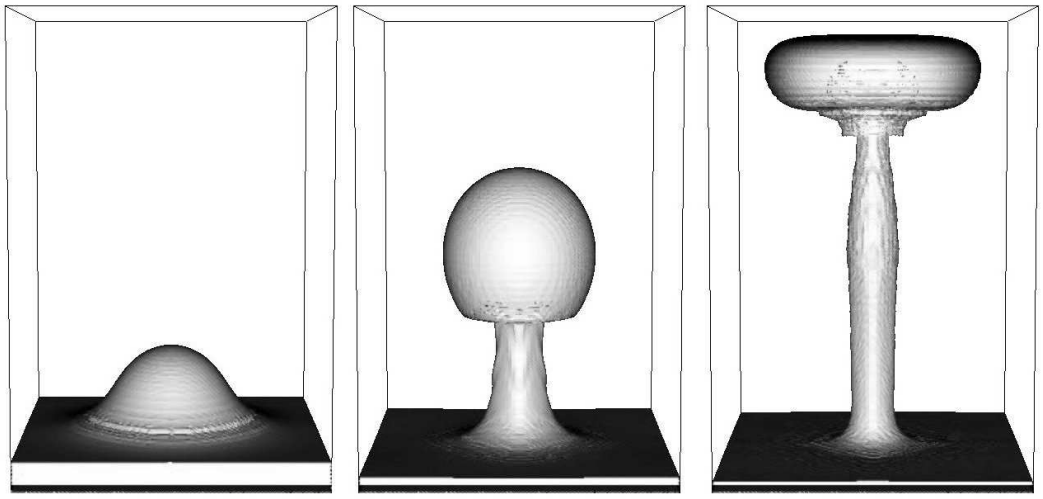

Fig. 1. Plume raising for the deeper Earth's mantel. Surface shows the interface between the Earth mantel and deep mantel material defined by the level set function. The resolution is 25 km.

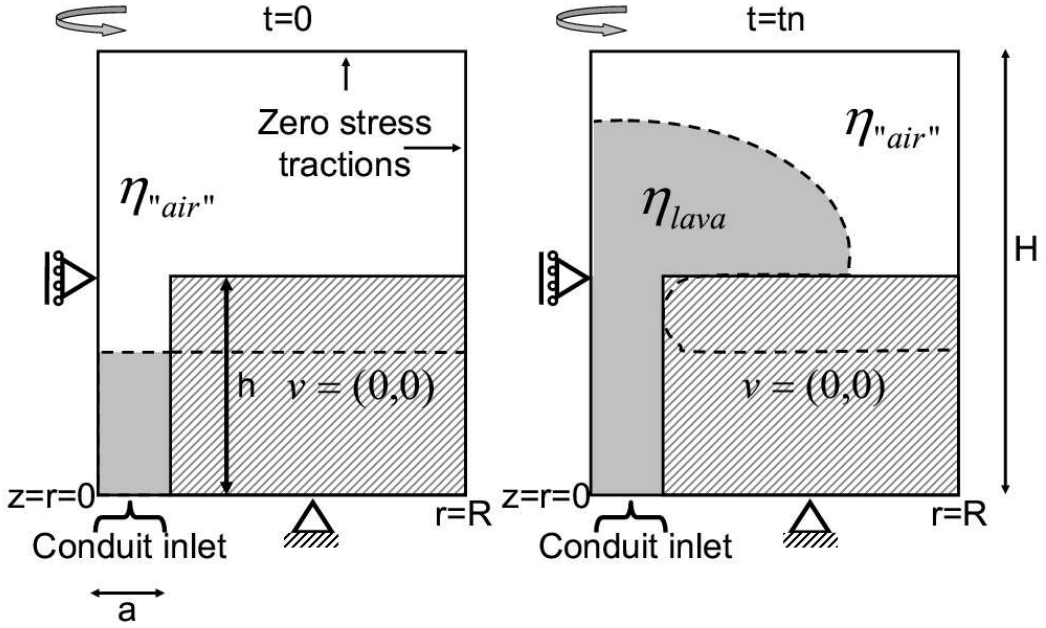

Fig. 2. Geometrical configuration for an axisymmetric volcano with the free-surface for the initial time $t =$ and a later time $t = tn$ shown as a dashed line. The shaded region at the bottom-right of the domain corresponds to the surface of the volcano and has the boundary condition of zero velocity and a height of h. The radius of the conduit is a.

Fig. 3. Free-surface of a lava dome and upper conduit. The conduit radius is 15 m. Above the conduit the lava is free to flow on a flat horizontal plane. The cut-away region also shows the interior temperature of the lava dome as given by the scale.