On Software Infrastructure for Computational Earth Sciences

L. Gross, J. Smillie, E. Thorne

Earth Systems Science Computational Centre (ESSCC), The University of Queensland, Brisbane, Australia

Abstract

Simulation software are build with three layers: the user interface, the mathematical models and the numerical methods. The mathematical layer provides abstraction from the numerical techniques and their implementation, the user interface provides abstraction from mathematical models. Each of the layer has a particular terminology being used, requires special skills fro the user to work within this layer and consequently needs particular computational tools. Appropriate tools for implementing numerical techniques is C/C++, for programming mathematical models is a scripting language such as python and for user interfaces are input files and graphical user interfaces. In this talk we will present the concepts of the mathematical modelling language *escript* and will show how *escript* is linked downwards into numerical numerical and upwards into user interfaces.

Keywords: Software Infrastructure; Mathematical Modelling, Finite Element Method;

Introduction

Conceptional the three layers ”user interface”, ”mathematical models” and ”numerical algorithms” found in numerical simulation codes are independent. For instance information needed to describe a salinity scenario in a specific region is independent from the mathematical calculus used to model salinity. The model on the other hand should be general enough to treat the relevant cases and is independent from finite elements (FEM), finite differences or finite volumes used to discretise the relevant partial differential equations. Moreover, the model is independent from the actual compute platform and the actual implementation of the discretization method which may be even platform dependent. Numerical algorithms are computational intensive and, to achieve sufficient code efficiency and scalability, have to be implemented in C/C++ to work closely to the underlying hardware. However, for the modeller working within the mathematical layer handling model complexity (coupling, time-dependency, non-linearity) and the flexibility to easily modify and test models rather than the code efficiency is of major concern. An object oriented scripting language, such as *python* (3), is appropriate for this layer. The end user of a verified model does not want to see the mathematical formulation involved in the model but wants to apply the model in his/her particular context in order to analyse and predict the behaviour of his/her system. A file, typically in XML format, is the appropriate way for providing a description of the problem. The file may be created through a graphical user interface or a web service.
It is pointed out that each of the layer has its individual terminology: The numerics layer uses terms like floating point numbers and data structures. The mathematical layer talks about functions and partial differential equations. The user interface uses terms like stress, temperature and viscosity. Moving from one layer to the other requires a translation: data like viscosity and temperature become coefficients in partial differential equations and coefficients become arrays of floating point numbers distributed across the processors in parallel machine.
Various tools have been developed to create graphical user interface and web services, some of them using graphical user interfaces themselves. Also a lot of work has been done on tool supporting the efficient and portable implementation of numerical methods. However, only very little work has been spent in the area of tools for the implementation of mathematical models. In this paper we will present the basic ideas of the modeling environment *escript* (2) and how *escript* is linked with numerical as well as the user interface layer. We will focus on the context of partial differential equations (PDEs).

Modelling Environment

In order to make use of existing technologies *escript* is an extension of the interactive scripting environment *python*. It introduces two new classes, namely the `Data` class and the `linearPDE` class.
Objects of the `Data` class define quantities with spatial distribution which are represented through their values on sample points. Examples are a temperature distribution given through its values at nodes and a stress tensor at quadrature points in the elements of a finite element mesh. In *escript* scalar, vector and tensorial quantities up to order 4 are supported. Objects can be manipulated by applying unitary operations (for instance $\cos$,$\sin$, $\log$) and be combined by applying binary operations (for instance $+$, $-$,$*$, $/$). A `Data` object is linked with a certain interpretation provided

by the numerical library in which context the object is used. If needed *escript* invokes interpolation during data manipulation. Typically, this occurs in binary operations when the arguments defined in a different context or when data are passed to a numerical library which requires data to be represented in a particular way, such as a FEM solver that requires the PDE coefficients on quadrature nodes.
A `linearPDE` object is used to define a general linear, steady, second order PDE for an unknown function u on the domain Ω. In tensor notation, the PDE has the form

$$-(A_{ijkl}u_{k,l} + B_{ijk}u_k)_{,j} + C_{ikl}u_{k,l} + D_{ik}u_k = -X_{ij,j} + Y_i \ , \tag{1}$$

where u_k denotes the components of the function u and $u_{,j}$ denotes the derivative of u with respect to the j-th spatial direction. A general form of natural boundary conditions and constraints can be considered. The functions A, B, C, D, X and Y are the coefficients of the PDE and are typically defined by `Data` objects. When a solution of the PDE is requested, *escript* passes the PDE to the solver library which returns a `Data` object representing the solution by its values, for instance, at the nodes of a FEM mesh. Currently *escript* is linked with the FEM solver library *finley* (1) but other libraries and even other discretization approaches can be included.
The following *python* function `incompressibleFluid` implements a simplified form of the penalty iteration scheme for a viscous, incompressible fluid. It takes the PDE domain `dom`, the viscosity `eta` and the internal force `F` as arguments:

```
def incompressibleFluid(dom,eta,F):
    E=Tensor4(0,ContinuousFunction(dom))
    for i in range(dom.getDim()):
      for j in range(dom.getDim()):
        E[i,i,j,j]+=Pe
        E[i,j,i,j]+=eta
        E[i,j,j,i]+=eta
    mypde=LinearPDE(dom)
    mypde.setValue(A=E,Y=F)
    p=Scalar(0,Function(dom))
    while Lsup(vkk)>tol:
        mypde.setValue(X=kronecker(dom)*p)
        v=mypde.getSolution()
        vkk=div(v)
        p-=Pe*vkk
    return v,p
```

The statement `div(v)` returns the divergence $v_{k,k}$ of `v`. The function returns velocity `v` and pressure `p`. The tensor `E` and the the pressure `p` are introduced with different attributes `ContinuousFunction()` and `Continuous()` defining a different degree of "smoothness". This mathematical concept of smoothness is implemented through different representations of values. In case of FEM, the tensor `E` would typically be hold at the nodes of the FEM mesh while the pressure is stored on the quadrature points. The solver library and the discretization method to be used to solve the PDE is defined by the domain `dom`.

Model Interfaces

The `LinearPDE` class provides the interface from *escript* downwards into the numerical algorithm layer. To build user interfaces models are wrapped by *python* classes which are subclasses of the *escript* `Model` class. The main feature of a `Model` class object is the ability to execute a time step for a given suitable step size which is chosen as the minimum step size over all models involved in the simulation. Moreover, model parameter such as viscosity `eta` and external force `F` in the example of the incompressible fluid are "highlighted". They can be linked with parameters of other models and can be exposed in an XML input file to assign values to them for instance through a graphical user interface.
If the class `IncompressibleFlow` implements a model of an incompressible fluid and `MaterialTable` is a `Model` class for a simple material table providing values for a temperature-dependent viscosity one uses

```
flow=IncompressibleFlow()
mat=MaterialTable()
mat.temperature=1000
flow.eta=Link(mat,"viscosity")
```

to link instances of the two classes. At any time of the simulation `IncompressibleFlow` will use the value provided by the `MaterialTable` object at that moment. The capability of *escript* to know about the context of data and to invoke data conversion when required is vital to make this very simple form using models actually working. This script can be represented as an XML file which can be edited, for instance to change the value for the temperature, and then be used to recreate the script for the new configuration.
In case of a Mantel convection simulation we would like to introduce a temperature dependent viscosity. If the `Temperature` class provides an implementation for temperature advection-diffusion model the following statements link this model with the incompressible flow model

```
temp=Temperature()
temp.velocity=Link(flow,"v")
mat.temperature=Link(temp,"T")
```

We assume here that `v` is the velocity provided by the flow model and `T` is the temperature of the temperature model. Instead of a *python* the link between the models can be established through an XML description.
The order in which the models perform there times steps is critical. The `Simulation` class which in this example is used in the form

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

will make sure that incompressible flow is updating its velocity before the temperature model is performing the next time step. The viscosity is calculated from the temperature of the previous time step.
The `Simulation` can be serialized into an XML file. The simulation can be started directly from the file. This opens the door of turning models into services in a grid environment. In the presented modelling environment appropriate interfaces can be built automatically. Suitable tools for building graphical user interface and web services automatically from the XML simualtion file are currently under construction.

Acknowledgements

This work is supported by the 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] 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).
[2] 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).
[3] http://www.python.org [October 2005].