On a Python Module for PDE-based Numerical Modelling

Lutz Gross, Elspeth Thorne, Matt Davies, Jon Smillie, and Hans Muhlhaus

Earth Systems Science Computational Centre (ESSCC), The University of Queensland, St Lucia, QLD 4072, Australia `l.gross@uq.edu.au`

Abstract. In this paper we will present the concepts of the *python*–based numerical modeling framework *escript*. Main application areas for *escript* are models based on coupled, non-linear, time-dependent partial differential equations (PDEs). One objective is to provide a easy-to-use programming environment for model developers to work independently form PDE solver libraries. The basic idea is to design an abstraction layer from particular numerical methods, their data structures and possibly platform dependent implementations. The second objective is to provide an environment for the model implementation which allows coupling models in a very easy way and building user interfaces (including GUIs and web services) automatically. This is achieved through implementing models as *python* objects where model parameters are represented as a special type of object attributes. These attributes can be serialized in XML format and can be linked to attributes of other model objects.

1 Spatial Functions

The *escript* [2] package is an extension of *python* [3]. All computational intensive tasks such as solving linear partial differential equations (PDEs), visualizations and data manipulations are implemented in C or C++. Except for data manipulations the *escript* package does not include implementation for these tasks, and relies on the program codes optimized for the particular compute architecture being used.

In *escript*, functions of spatial coordinates are hold in `Data` class objects. A `Data` class objects has a `FunctionSpace` object assigned to it, which defines the `Domain` of the function and the type of function (for instance its smoothness) represented by the object. A `Domain` defines not only the geometry of a domain but also the discretization method to be used. The `FunctionSpace` defines how the function is represented. For examples, in the case of finite elements (FEM) the `Domain` would hold references to the tables of node coordinates and elements. These data are not managed by *escript* but the PDE solver library. Typically, in the FEM context, a temperature distribution is given through its values at nodes and a stress tensor at quadrature points. The coresponding `Data` class objects are defined on the same `Domain` but within different `FunctionSpace`s, namely `ContinuousFunction` and `Function`, respectively. The `Data` class objects are managed by *escript* while PDE solver libraries can read from and write to `Data` class objects under the assumption that the access does not require data conversion or communication. Suitable functions for interpolation and data redistribution which are called by *escript* to change the `FunctionSpace` has to be provided by the solver library.

For each individual data point *escript* support scalar, vector and tensorial quantities up to order 4. From *python* `Data` objects can be manipulated by applying unitary operations (for instance cosine ,sine, logarithm) and be combined by applying binary operations (for instance $+$, $-$,$*$, $/$). If needed *escript* invokes interpolation to match the `FunctionSpace`. Operations are implemented in C/C++ and parallelized using OpenMP (MPI is under construction).

2 Partial Differential Equations

The second key component in *escript* is the `linearPDE` class used to define a general linear, steady, second order PDE for an unknown function u on the PDE domain. 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 defined by `Data` objects. When dealing with non-linear and time-dependent problems, suitable high-level schemes are used to reduce the problem to linear PDEs that are solved in each iteration step. The coefficients are updated trough *escript*. When a solution of the PDE is requested, *escript* passes the PDE coefficient 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.

3 Example

We present a simple example that illustrates how to use *escript* to solve the Stokes equation using the iterative penalty method. In each iteration step the linear PDE

$$-(\eta(v_{i,j} + v_{j,i}))_{,j} - Pe\, v_{k,ki} = F_i - p_j, \tag{2}$$

has to be solved to get new velovity v from the current pressure approximation p. Then the pressure is updated by

$$p = p - Pe\, v_{k,k} \tag{3}$$

The iteration is terminated when $v_{k,k}$ is small. In PDE (2) η is the viscosity, F represents external forces and Pe denotes the penalty factor.

The following *python* function `incompressibleFluid` implements the iteration scheme for given `Domain` object `domain`, viscosity `eta` and internal force `F`. The function returns velocity `v` and pressure `p`:

```
def incompressibleFluid(domain,eta,F):
    E=Tensor4(0,Function(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
    pde1=LinearPDE(domain)
    pde1.setValue(A=E,Y=F)
    pde2=LinearPDE(domain)
    pde2.setValue(D=1.)
    pde1.setReductionOn()
    p=Scalar(0,ReducedSolution(domain))
    while Lsup(vkk)>tol:
        mypde1.setValue(X=kronecker(domain)*p)
        v=pde1.getSolution()
        pde2.setValue(Y=div(v))
        vkk=pde2.getSolution()
        p-=Pe*vkk
    return v,p
```

As required by the Ladyzhenskaya-Babuska-Brezzi condition the pressure is defined in the `FunctionSpace` `ReducedSolution`. The viscosity (and similar the external force) may be a float object or, for instance if viscosity is depending on temperature, a scalar `Data` object. If required, interpolation of the coefficients is performed to match the `FunctionSpace` required by the PDE solver library.

4 Managing Models

Models in *escript* can be implemented as subclasses of the *Model* class. A particular model implements a set of methods, for instance the execution of a time step and calculation of a safe time step size. The function `IncompressibleFlow` defined in the previous section would be implemented in three method, namely initialization phase, update phase and check of the stopping criterion (due to the limitation of space we cannot present more details.). The model parameters viscosity, external force, velocity and pressure are defined as attributes of the class defining the the model.

If the class `IncompressibleFlow` implements a model of an incompressible fluid, `Temperature` implements a model for temperature advection-diffusion, and `MaterialTable` is a `Model` class for a material table providing values for a temperature-dependent viscosity, a coupling of the temperature and fluid flow model can be implemented in the following python script:

```
flow=IncompressibleFlow()
temp=Temperature()
mat=MaterialTable()
flow.eta=Link(mat,"viscosity")
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. When `IncompressibleFlow` references its attribute `eta`, it will access, via the `Link` object, the viscosity 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 work.

As the order in which the models perform their time steps is critical, the model execution is handled by an instance of the `Simulation` class. In the example, this will take the form

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

which will make sure that incompressible flow model updates its velocity before the temperature model performs the next time step. The viscosity is calculated from the temperature of the previous time step. Moreover, *escript* provides a mechanism to build instances of the `Simulation` class from files in the XML dialect ESysXML. Files can be generated through serialization of `Simulation` class objects or from a (graphical) user interface or problem solving environment. A user interface based on GridSphere [4] that allows manipulating and running ESysXML files via a web and grid service is currently under construction.

Acknowledgments

This project 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 [February 2006].
4. http://www.gridsphere.org [February 2006].