Abstract

\texttt{esys.escript} is a python-based environment for implementing mathematical models, in particular those based on coupled, non-linear, time-dependent partial differential equations.

It consists of four major components

- \texttt{esys.escript} core library
- finite element solver \texttt{esys.finley} (which uses fast vendor-supplied solvers or our paso linear solver library)
- the meshing interface \texttt{esys.pycad}
- VTK visualization interface \texttt{esys.pyvisi}
- model library \texttt{modellib}

The current version supports parallelization through both MPI for distributed memory and OpenMP for distributed shared memory.
1 Tutorial: Solving PDEs  
  1.1 Installation ......................................................... 1  
  1.2 The First Steps .................................................... 1  
  1.3 The Diffusion Problem ........................................... 5  
  1.4 3-D Wave Propagation .......................................... 11  
  1.5 Elastic Deformation ............................................. 14  
  1.6 Stokes Flow ....................................................... 17  

2 Execution of an escript Script  
  2.1 Overview ........................................................... 23  
  2.2 Options ............................................................... 24  
  2.3 Input and Output ................................................ 25  
  2.4 Hints for MPI Programming .................................... 25  

3 The Module esys.escript  
  3.1 esys.escript Classes ............................................. 31  
  3.2 Utilities ............................................................... 41  

4 The Module esys.escript.linearPDEs  
  4.1 Linear Partial Differential Equations ....................... 43  

5 The Module esys.pycad  
  5.1 Introduction ....................................................... 51  
  5.2 esys.pycad Classes ............................................. 52  
  5.3 Interface to the mesh generation software .................. 54  

6 The Module esys.pyvisi  
  6.1 Introduction ....................................................... 57  
  6.2 esys.pyvisi Classes .......................................... 57  
  6.3 More Examples .................................................... 75  
  6.4 Useful Keys ......................................................... 79  
  6.5 Sample Output .................................................... 80  

7 Models  
  7.1 Stokes Problem .................................................... 83  
  7.2 Darcy Flux .......................................................... 85  
  7.3 Level Set Method ................................................. 88  
  7.4 Isotropic Kelvin Material ..................................... 89  

8 Appendix  
  8.1 Einstein Notation ................................................ 93  

Index ................................................................. 95
CHAPTER
ONE

Tutorial: Solving PDEs

1.1 Installation

To download escript and friends, please visit https://launchpad.net/escript-finley. The web site will offer binary distributions for some platforms and provide information about the installation process.

Please direct any questions you might have to mailto:esys@esscc.uq.edu.au.

1.2 The First Steps

In this chapter we give an introduction how to use esys.escript to solve a partial differential equation (PDE). We assume you are at least a little familiar with Python. The knowledge presented at the Python tutorial at http://docs.python.org/tut/tut.html is more than sufficient.

The PDE we wish to solve is the Poisson equation

\[-\Delta u = f\]  \hspace{1cm} (1.1)

for the solution \(u\). The function \(f\) is the given right hand side. The domain of interest, denoted by \(\Omega\), is the unit square

\[\Omega = [0, 1]^2 = \{(x_0; x_1) | 0 \leq x_0 \leq 1 \text{ and } 0 \leq x_1 \leq 1\}\]  \hspace{1cm} (1.2)

The domain is shown in Figure 1.1.

\[\Delta\] denotes the Laplace operator, which is defined by

\[\Delta u = (u_0)_0 + (u_1)_1\]  \hspace{1cm} (1.3)
where, for any function \( u \) and any direction \( i \), \( u_{,i} \) denotes the partial derivative of \( u \) with respect to \( i \). Basically, in the subindex of a function, any index to the left of the comma denotes a spatial derivative with respect to the index. To get a more compact form we will write \( u_{,ij} = (u_{,i})_{,j} \) which leads to

\[
\Delta u = u_{,00} + u_{,11} = \sum_{i=0}^{2} u_{,ii}
\]  

(1.4)

We often find that use of nested \( \sum \) symbols makes formulas cumbersome, and we use the more convenient Einstein summation convention. This drops the \( \sum \) sign and assumes that a summation is performed over any repeated index. For instance we write

\[
x_{,ij}y_{,i} = \sum_{i=0}^{2} x_{,i}y_{,i}
\]  

(1.5)

\[
x_{,i}u_{,i} = \sum_{i=0}^{2} x_{,i}u_{,i}
\]  

(1.6)

\[
u_{,ii} = \sum_{i=0}^{2} u_{,ii}
\]  

(1.7)

\[
x_{ij}u_{,ij} = \sum_{j=0}^{2} \sum_{i=0}^{2} x_{ij}u_{,ij}
\]  

(1.8)

With the summation convention we can write the Poisson equation as

\[-u_{,ii} = 1\]  

(1.10)

where \( f = 1 \) in this example.

On the boundary of the domain \( \Omega \) the normal derivative \( n_{,i}u_{,i} \) of the solution \( u \) shall be zero, ie. \( u \) shall fulfill the homogeneous Neumann boundary condition

\[n_{,i}u_{,i} = 0 .\]  

(1.11)

\( n = (n_{,i}) \) denotes the outer normal field of the domain, see Figure 1.1. Remember that we are applying the Einstein summation convention, i.e \( n_{,i}u_{,i} = n_{,0}u_{,0} + n_{,1}u_{,1} \). The Neumann boundary condition of Equation (1.11) should be fulfilled on the set \( \Gamma^N \) which is the top and right edge of the domain:

\[\Gamma^N = \{ (x_0; x_1) \in \Omega | x_0 = 1 \text{ or } x_1 = 1 \}\]  

(1.12)

On the bottom and the left edge of the domain which is defined as

\[\Gamma^D = \{ (x_0; x_1) \in \Omega | x_0 = 0 \text{ or } x_1 = 0 \}\]  

(1.13)

the solution shall be identically zero:

\[u = 0 .\]  

(1.14)

This kind of boundary condition is called a homogeneous Dirichlet boundary condition. The partial differential equation in Equation (1.10) together with the Neumann boundary condition Equation (1.11) and Dirichlet boundary condition in Equation (1.14) form a so called boundary value problem (BVP) for the unknown function \( u \).

In general the BVP cannot be solved analytically and numerical methods have to be used construct an approximation of the solution \( u \). Here we will use the finite element method (FEM). The basic idea is to fill the domain with a set of points called nodes. The solution is approximated by its values on the nodes. Moreover, the domain

\[\n = \{ n_{,i} \} \]  

denotes the outer normal field of the domain, see Figure 1.1. Remember that we are applying the Einstein summation convention, i.e \( n_{,i}u_{,i} = n_{,0}u_{,0} + n_{,1}u_{,1} \). The Neumann boundary condition of Equation (1.11) should be fulfilled on the set \( \Gamma^N \) which is the top and right edge of the domain:

\[\Gamma^N = \{ (x_0; x_1) \in \Omega | x_0 = 1 \text{ or } x_1 = 1 \}\]  

(1.12)

On the bottom and the left edge of the domain which is defined as

\[\Gamma^D = \{ (x_0; x_1) \in \Omega | x_0 = 0 \text{ or } x_1 = 0 \}\]  

(1.13)

the solution shall be identically zero:

\[u = 0 .\]  

(1.14)

This kind of boundary condition is called a homogeneous Dirichlet boundary condition. The partial differential equation in Equation (1.10) together with the Neumann boundary condition Equation (1.11) and Dirichlet boundary condition in Equation (1.14) form a so called boundary value problem (BVP) for the unknown function \( u \).

In general the BVP cannot be solved analytically and numerical methods have to be used construct an approximation of the solution \( u \). Here we will use the finite element method (FEM). The basic idea is to fill the domain with a set of points called nodes. The solution is approximated by its values on the nodes. Moreover, the domain

\[\n = \{ n_{,i} \} \]  

some readers may familiar with the notation \( \frac{\partial u}{\partial n} = n_{,i}u_{,i} \) for the normal derivative.
is subdivided into smaller sub-domains called elements. On each element the solution is represented by a polynomial of a certain degree through its values at the nodes located in the element. The nodes and its connection through elements is called a mesh. Figure 1.2 shows an example of a FEM mesh with four elements in the $x_0$ and four elements in the $x_1$ direction over the unit square. For more details we refer the reader to the literature, for instance Reference [25, 6].

The esys.escript solver we want to use to solve this problem is embedded into the python interpreter language. So you can solve the problem interactively but you will learn quickly that is more efficient to use scripts which you can edit with your favorite editor. To enter the escript environment you use escript:

```
escript
```

which will pass you on to the python prompt

```
Python 2.5.2 (r252:60911, Oct 5 2008, 19:29:17)
[GCC 4.3.2] on linux2
Type "help", "copyright", "credits" or "license" for more information.
```  

Here you can use all available python commands and language features, for instance

```python
>>> x=2+3
>>> print "2+3=",x
2+3= 5
```

We refer to the python users guide if you not familiar with python.

esys.escript provides the class Poisson to define a Poisson equation. (We will discuss a more general form of a PDE that can be defined through the LinearPDE class later). The instantiation of a Poisson class object requires the specification of the domain $\Omega$. In esys.escript the Domain class objects are used to describe the geometry of a domain but it also contains information about the discretization methods and the actual solver which is used to solve the PDE. Here we are using the FEM library esys.finley. The following statements create the Domain object mydomain from the esys.finley method Rectangle

```python
from esys.finley import Rectangle
mydomain = Rectangle(l0=1.,l1=1.,n0=40, n1=20)
```

In this case the domain is a rectangle with the lower, left corner at point (0,0) and the right, upper corner at

---

3escript is not available under Windows yet. If you run under windows you can just use the python command and the OMP_NUM_THREADS environment variable to control the number of threads.
\((l_0, l_1) = (1, 1)\). The arguments \(n_0\) and \(n_1\) define the number of elements in \(x_0\) and \(x_1\)-direction respectively. For more details on Rectangle and other Domain generators within the esys.finley module, see Chapter ??.

The following statements define the Poisson class object \(\text{mypde}\) with domain \(\text{mydomain}\) and the right hand side \(f\) of the PDE to constant 1:

```python
from esys.escript.linearPDEs import Poisson
mypde = Poisson(mydomain)
mypde.setValue(f=1)
```

We have not specified any boundary condition but the Poisson class implicitly assumes homogeneous Neuman boundary conditions defined by Equation (1.11). With this boundary condition the BVP we have defined has no unique solution. In fact, with any solution \(u\) and any constant \(C\) the function \(u + C\) becomes a solution as well. We have to add a Dirichlet boundary condition. This is done by defining a characteristic function which has positive values at locations \(x = (x_0, x_1)\) where Dirichlet boundary condition is set and 0 elsewhere. In our case of \(\Gamma^D\) defined by Equation (1.13), we need to construct a function \(\gamma_D\) which is positive for the cases \(x_0 = 0\) or \(x_1 = 0\). To get an object \(x\) which contains the coordinates of the nodes in the domain use

\[
x = \text{mydomain.getX()}
\]

The method \(\text{getX}\) of the Domain \(\text{mydomain}\) gives access to locations in the domain defined by \(\text{mydomain}\). The object \(x\) is actually a Data object which will be discussed in Chapter 3 in more detail. What we need to know here is that \(x\) has rank (number of dimensions) and a shape (list of dimensions) which can be viewed by calling the getRank and getShape methods:

```python
print "rank ", x.getRank(), ", shape ", x.getShape()
```

This will print something like

```
rank 1, shape (2,)
```

The Data object also maintains type information which is represented by the FunctionSpace of the object. For instance

```python
print x.getFunctionSpace()
```

will print

```
Function space type: Finley_Nodes on FinleyMesh
```

which tells us that the coordinates are stored on the nodes of (rather than on points in the interior of) a esys.finley mesh. To get the \(x_0\) coordinates of the locations we use the statement

\[
x_0 = x[0]
\]

Object \(x_0\) is again a Data object now with rank 0 and shape (). It inherits the FunctionSpace from \(x\):

```python
print x0.getRank(),x0.getShape(),x0.getFunctionSpace()
```

will print

```
0 () Function space type: Finley_Nodes on FinleyMesh
```

We can now construct a function \(\gamma_D\) which is only non-zero on the bottom and left edges of the domain with

```python
from esys.escript import whereZero
gammaD=whereZero(x[0])+whereZero(x[1])
```

\(\text{whereZero}(x[0])\) creates function which equals 1 where \(x[0]\) is (almost) equal to zero and 0 elsewhere. Similarly, \(\text{whereZero}(x[1])\) creates function which equals 1 where \(x[1]\) is equal to zero and 0 elsewhere. The sum of the results of \(\text{whereZero}(x[0])\) and \(\text{whereZero}(x[1])\) gives a function on the domain \(\text{mydomain}\) which is strictly positive where \(x_0\) or \(x_1\) is equal to zero. Note that \(\gamma_D\) has the same rank , shape and FunctionSpace like \(x\) used to define it. So from

```python
print gammaD.getRank(),gammaD.getShape(),gammaD.getFunctionSpace()
```
one gets

0 () Function space type: Finley_Nodes on FinleyMesh

An additional parameter $q$ of the \texttt{setValue} method of the \texttt{Poisson} class defines the characteristic function of the locations of the domain where homogeneous Dirichlet boundary condition are set. The complete definition of our example is now:

```python
from esys.linearPDEs import Poisson
x = mydomain.getX()
gammaD = whereZero(x[0])+whereZero(x[1])
mypde = Poisson(domain=mydomain)
mypde.setValue(f=1,q=gammaD)
```

The first statement imports the \texttt{Poisson} class definition from the \texttt{esys.escript.linearPDEs} module \texttt{esys.escript} package. To get the solution of the Poisson equation defined by \texttt{mypde} we just have to call its \texttt{getSolution}.

Now we can write the script to solve our Poisson problem

```python
from esys.escript import *
from esys.escript.linearPDEs import Poisson
from esys.finley import Rectangle
# generate domain:
mydomain = Rectangle(l0=1.,l1=1.,n0=40, n1=20)
# define characteristic function of Gamma^D
x = mydomain.getX()
gammaD = whereZero(x[0])+whereZero(x[1])
# define PDE and get its solution u
mypde = Poisson(domain=mydomain)
mypde.setValue(f=1,q=gammaD)
u = mypde.getSolution()
# write u to an external file
saveVTK("u.xml",sol=u)
```

The entire code is available as `poisson.py` in the example directory.

The last statement writes the solution (tagged with the name ”sol”) to a file named `u.xml` in \texttt{VTK} file format. Now you may run the script using the \texttt{esys.escript} environment and visualize the solution using \texttt{mayavi}:

```
escript poisson.py
mayavi -d u.xml -m SurfaceMap
```

See Figure 1.3.

### 1.3 The Diffusion Problem

#### 1.3.1 Outline

In this chapter we will discuss how to solve a time-dependent temperature diffusion PDE for a given block of material. Within the block there is a heat source which drives the temperature diffusion. On the surface, energy can radiate into the surrounding environment. Figure 1.4 shows the configuration.

In the next Section 1.3.2 we will present the relevant model. A time integration scheme is introduced to calculate the temperature at given time nodes $t^{(n)}$. We will see that at each time step a Helmholtz equation must be solved. The implementation of a Helmholtz equation solver will be discussed in Section 1.3.3. In Section 1.3.4 the solver of the Helmholtz equation is used to build a solver for the temperature diffusion problem.
1.3.2 Temperature Diffusion

The unknown temperature \( T \) is a function of its location in the domain and time \( t > 0 \). The governing equation in the interior of the domain is given by

\[
\rho c_p T_t - (\kappa T)_x = q_H \tag{1.15}
\]

where \( \rho c_p \) and \( \kappa \) are given material constants. In case of a composite material the parameters depend on their location in the domain. \( q_H \) is a heat source (or sink) within the domain. We are using the Einstein summation convention as introduced in Chapter 1.2. In our case we assume \( q_H \) to be equal to a constant heat production rate \( q^c \) on a circle or sphere with center \( x^c \) and radius \( r \) and 0 elsewhere:

\[
q_H(x, t) = \begin{cases} 
q^c & \|x - x^c\| \leq r \\
0 & \text{else}
\end{cases} \tag{1.16}
\]

for all \( x \) in the domain and all time \( t > 0 \).

On the surface of the domain we are specifying a radiation condition which prescribes the normal component of the flux \( \kappa T \) to be proportional to the difference of the current temperature to the surrounding temperature \( T_{ref} \):

\[
\kappa T_n = \eta (T_{ref} - T) \tag{1.17}
\]
\( \eta \) is a given material coefficient depending on the material of the block and the surrounding medium. \( n_i \) is the \( i \)-th component of the outer normal field at the surface of the domain.

To solve the time-dependent Equation (1.15) the initial temperature at time \( t = 0 \) has to be given. Here we assume that the initial temperature is the surrounding temperature:

\[
T(x, 0) = T_{\text{ref}}
\]

for all \( x \) in the domain. It is pointed out that the initial conditions satisfy the boundary condition defined by Equation (1.17).

The temperature is calculated at discrete time nodes \( t^{(n)} \) where \( t^{(0)} = 0 \) and \( t^{(n)} = t^{(n-1)} + h \) where \( h > 0 \) is the step size which is assumed to be constant. In the following the upper index \( (n) \) refers to a value at time \( t^{(n)} \). The simplest and most robust scheme to approximate the time derivative of the temperature is the backward Euler scheme. The backward Euler scheme is based on the Taylor expansion of \( T \) at time \( t^{(n)} \):

\[
T^{(n)} - T^{(n-1)} = h \cdot T_t^{(n)}
\]

This is inserted into Equation (1.15). By separating the terms at \( t^{(n)} \) and \( t^{(n-1)} \) one gets for \( n = 1, 2, 3 \ldots \)

\[
\frac{\rho C_p}{h} T^{(n)} - \left( \kappa T_{,i}^{(n)} \right)_{,i} = q_H + \frac{\rho C_p}{h} T^{(n-1)}
\]

where \( T^{(0)} = T_{\text{ref}} \) is taken form the initial condition given by Equation (1.18). Together with the natural boundary condition

\[
\kappa T_{,i}^{(n)} n_i = \eta (T_{\text{ref}} - T^{(n)})
\]

taken from Equation (1.17) this forms a boundary value problem that has to be solved for each time step. As a first step to implement a solver for the temperature diffusion problem we will first implement a solver for the boundary value problem that has to be solved at each time step.

1.3.3 Helmholtz Problem

The partial differential equation to be solved for \( T^{(n)} \) has the form

\[
\omega T^{(n)} - \left( \kappa T_{,i}^{(n)} \right)_{,i} = f
\]

and we set

\[
\omega = \frac{\rho C_p}{h} \quad \text{and} \quad f = q_H + \frac{\rho C_p}{h} T^{(n-1)}
\]

With \( g = \eta T_{\text{ref}} \) the radiation condition defined by Equation (1.21) takes the form

\[
\kappa T_{,i}^{(n)} n_i = g - \eta T^{(n)} \quad \text{on} \quad \Gamma
\]

The partial differential Equation (1.22) together with boundary conditions of Equation (1.24) is called the Helmholtz equation.

We want to use the LinearPDE class provided by esys.escript to define and solve a general linear, steady, second order PDE such as the Helmholtz equation. For a single PDE the LinearPDE class supports the following form:

\[
-(A_{jl} u_{,l})_{,j} + D u = Y
\]

where we show only the coefficients relevant for the problem discussed here. For the general form of single PDE see Equation (4.1). The coefficients \( A \) and \( Y \) have to be specified through Data objects in the general FunctionSpace on the PDE or objects that can be converted into such Data objects. \( A \) is a rank-2 Data object and \( D \) and \( Y \) are scalar. The following natural boundary conditions are considered on \( \Gamma \):

\[
n_j A_{jl} u_{,l} + d u = y
\]

Notice that the coefficient \( A \) is the same like in the PDE Equation (1.25). The coefficients \( d \) and \( y \) are each a scalar Data object in the boundary FunctionSpace. Constraints for the solution prescribing the value of the solution at certain locations in the domain. They have the form

\[
u = r \quad \text{where} \quad q > 0
\]

1.3. The Diffusion Problem
and $q$ are each scalar Data object where $q$ is the characteristic function defining where the constraint is applied. The constraints defined by Equation (1.27) override any other condition set by Equation (1.25) or Equation (1.26).

The Poisson class of the esys.escript.linearPDEs module, which we have already used in Chapter 1.2, is in fact a subclass of the more general LinearPDE class. The esys.escript.linearPDEs module provides a Helmholtz class but we will make direct use of the general LinearPDE class.

By inspecting the Helmholtz equation (1.22) and boundary condition (1.24) and substituting $u$ for $T^{(n)}$ we can easily assign values to the coefficients in the general PDE of the LinearPDE class:

$$A_{ij} = \kappa \delta_{ij} \quad D = \omega \quad Y = f \quad d = \eta \quad y = g$$

(1.28)

$\delta_{ij}$ is the Kronecker symbol defined by $\delta_{ij} = 1$ for $i = j$ and 0 otherwise. Undefined coefficients are assumed to be not present. In this diffusion example we do not need to define a characteristic function $q$ because the boundary conditions we consider in Equation (1.24) are just the natural boundary conditions which are already defined in the LinearPDE class (shown in Equation (1.26)).

The Helmholtz equation can be set up by following way:

```python
mypde = LinearPDE(mydomain)
mypde.setValue(A=kappa*kronecker(mydomain), D=omega, Y=f, d=eta, y=g)
u = mypde.getSolution()
```

where we assume that mydomain is a Domain object and kappa omega eta and $g$ are given scalar values typically float or Data objects. The setValue method assigns values to the coefficients of the general PDE. The getSolution method solves the PDE and returns the solution $u$ of the PDE. kronecker is esys.escript function returning the Kronecker symbol.

The coefficients can set by several calls of setValue where the order can be chosen arbitrarily. If a value is assigned to a coefficient several times, the last assigned value is used when the solution is calculated:

```python
mypde = LinearPDE(mydomain)
mypde.setValue(A=kappa*kronecker(mydomain), d=eta)
mypde.setValue(D=omega, Y=f, d=eta, y=g)
mypde.setValue(d=2*eta) # overwrites d=eta
u = mypde.getSolution()
```

In some cases the solver of the PDE can make use of the fact that the PDE is symmetric where the PDE is called symmetric if

$$A_{ji} = A_{ij}.$$  

(1.29)

Note that $D$ and $d$ may have any value and the right hand sides $Y$, $y$ as well as the constraints are not relevant. The Helmholtz problem is symmetric. The LinearPDE class provides the method checkSymmetry method to check if the given PDE is symmetric.

```python
mypde = LinearPDE(mydomain)
mypde.setValue(A=kappa*kronecker(mydomain), d=eta)
print mypde.checkSymmetry() # returns True
mypde.setValue(B=kronecker(mydomain)[0])
print mypde.checkSymmetry() # returns False
mypde.setValue(C=kronecker(mydomain)[0])
print mypde.checkSymmetry() # returns True
```

Unfortunately, a checkSymmetry is very expensive and is recommendable to use for testing and debugging purposes only. The setSymmetryOn method is used to declare a PDE symmetric:

```python
mypde = LinearPDE(mydomain)
mypde.setValue(A=kappa*kronecker(mydomain))
mypde.setSymmetryOn()
```

Now we want to see how we actually solve the Helmholtz equation on a rectangular domain of length $l_0 = 5$ and height $l_1 = 1$. We choose a simple test solution such that we can verify the returned solution against the exact

4There is a difference in esys.escript of being not present and set to zero. As not present coefficients are not processed, it is more efficient to leave a coefficient undefined instead of assigning zero to it.

5Please, note that this is not a complete code. The complete code can be found in “helmholtz.py”.
answer. Actually, we take \( T = x_0 \) (here \( q_H = 0 \)) and then calculate the right hand side terms \( f \) and \( g \) such that the test solution becomes the solution of the problem. If we assume \( \kappa \) as being constant, an easy calculation shows that we have to choose \( f = \omega \cdot x_0 \). On the boundary we get \( \kappa n u_i = \kappa n_0 \). So we have to set \( g = \kappa n_0 + \eta x_0 \). The following script ‘helmholtz.py’ which is available in the example directory implements this test problem using the esys.finley PDE solver:

```python
from esys.escript import *
from esys.escript.linearPDEs import LinearPDE
from esys.finley import Rectangle
#... set some parameters ...
kappa=1.
omega=0.1
eta=10.
#... generate domain ...
mydomain = Rectangle(l0=5.,l1=1.,n0=50, n1=10)
#... open PDE and set coefficients ...
mypde=LinearPDE(mydomain)
mypde.setSymmetryOn()
n=mydomain.getNormal()
x=mydomain.getX()
mypde.setValue(A=kappa*kronecker(mydomain),D=omega,Y=omega*x[0],
d=eta,y=kappa*n[0]+eta*x[0])
#... calculate error of the PDE solution ...
u=mypde.getSolution()
print "error is ",Lsup(u-x[0])
saveVTK("x0.xml",sol=u)
```

To visualize the solution ‘x0. xml’ just use the command

```
mayavi -d u.xml -m SurfaceMap &
```

and it is easy to see that the solution \( T = x_0 \) is calculated.

The script is similar to the script ‘poisson.py’ discussed in Chapter 1.2. mydomain.getNormal() returns the outer normal field on the surface of the domain. The function \( Lsup \) imported by the from esys.escript import * statement and returns the maximum absolute value of its argument. The error shown by the print statement should be in the order of \( 10^{-7} \). As piecewise bi-linear interpolation is used by esys.finley approximate the solution and our solution is a linear function of the spatial coordinates one might expect that the error would be zero or in the order of machine precision (typically \( \approx 10^{-15} \)). However most PDE packages use an iterative solver which is terminated when a given tolerance has been reached. The default tolerance is \( 10^{-8} \). This value can be altered by using the setTolerance of the LinearPDE class.

### 1.3.4 The Transition Problem

Now we are ready to solve the original time-dependent problem. The main part of the script is the loop over time \( t \) which takes the following form:

```python
t=0
T=Tref
mypde=LinearPDE(mydomain)
mypde.setValue(A=kappa*kronecker(mydomain),D=rhocp/h,d=eta,y=eta*Tref)
while t<t_end:
    t=t+h
    mypde.setValue(Y=q+rhocp/h*T)
    T=mypde.getSolution()
```

\( \kappa, \text{rhocp, } \eta \text{ and } T_{ref} \) are input parameters of the model. \( q \) is the heat source in the domain and \( h \) is the time step size. The variable \( T \) holds the current temperature. It is used to calculate the right hand side coefficient \( f \) in the Helmholtz equation in Equation (1.22). The statement \( T=mypde.getSolution() \) overwrites \( T \) with the temperature of the new time step \( t + h \). To get this iterative process going we need to specify the initial temperature distribution, which equal to \( T_{ref} \). The LinearPDE class object mypde and coefficients not changing over time are set up before the loop over time is entered. In each time step only the coefficient \( Y \) is reset as it depends on the
temperature of the previous time step. This allows the PDE solver to reuse information from previous time steps as much as possible.

The heat source $q_H$ which is defined in Equation (1.16) is $qc$ in an area defined as a circle of radius $r$ and center $xc$ and zero outside this circle. $q0$ is a fixed constant. The following script defines $q_H$ as desired:

```python
from esys.escript import length, whereNegative
xc=[0.02,0.002]
r=0.001
x=mydomain.getX()
qH=q0*whereNegative(length(x-xc)-r)
```

$x$ is a Data class object of the esys.escript module defining locations in the Domain mydomain. The length() imported from the esys.escript module returns the Euclidean norm:

$$\|y\| = \sqrt{y_i y_i} = \text{esys.escript.length}(y)$$

So length$(x-xc)$ calculates the distances of the location $x$ to the center of the circle $xc$ where the heat source is acting. Note that the coordinates of $xc$ are defined as a list of floating point numbers. It is automatically converted into a Data class object before being subtracted from $x$. The function whereNegative applied to length$(x-xc)-r$, returns a Data object which is equal to one where the object is negative (inside the circle) and zero elsewhere. After multiplication with $qc$ we get a function with the desired property of having value $qc$ inside the circle and zero elsewhere.

Now we can put the components together to create the script ‘diffusion.py’ which is available in the example directory:

```python
from esys.escript import *
from esys.escript.linearPDEs import LinearPDE
from esys.finley import Rectangle
# ... set some parameters ...
xc=[0.02,0.002]
r=0.001
qc=50.e6
Tref=0.
rhocp=2.6e6
et=75.
kappa=240.
tend=5.
# ... time, time step size and counter ...
t=0
h=0.1
i=0
# ... generate domain ...
mydomain = Rectangle(l0=0.05,l1=0.01,n0=250, n1=50)
# ... open PDE ...
mypde=LinearPDE(mydomain)
mypde.setSymmetryOn()
mypde.setValue(A=kappa*kronecker(mydomain),D=rhocp/h,d=eta,y=eta*Tref)
# ... set heat source: ...
x=mydomain.getX()
qH=qc*whereNegative(length(x-xc)-r)
# ... set initial temperature ...
T=Tref
# ... start iteration:
while t<tend:
i+=1
  t+=h
  print "time step ", t
  mypde.setValue(Y=qH+rhocp/h*T)
  T=mypde.getSolution()
  saveVTK("T.%d.xml"%i, temp=T)
```

The script will create the files ‘T.1.xml’, ‘T.2.xml’, ..., ‘T.50.xml’ in the directory where the script has been started. The files give the temperature distributions at time steps 1, 2, ..., 50 in the VTK file format.
Figure 1.5 shows the result for some selected time steps. An easy way to visualize the results is the command

```bash
mayavi -d T.1.xml -m SurfaceMap &
```

Use the Configure Data window in mayavi to move forward and backwards in time.

1.4 3-D Wave Propagation

In this next example we want to calculate the displacement field \( u_i \) for any time \( t > 0 \) by solving the wave equation:

\[
\rho u_{i,t,t} - \sigma_{ij,j} = 0
\]  

(1.31)

in a three dimensional block of length \( L \) in \( x_0 \) and \( x_1 \) direction and height \( H \) in \( x_2 \) direction. \( \rho \) is the known density which may be a function of its location. \( \sigma_{ij} \) is the stress field which in case of an isotropic, linear elastic material is given by

\[
\sigma_{ij} = \lambda u_{k,k} \delta_{ij} + \mu (u_{i,j} + u_{j,i})
\]

(1.32)

where \( \lambda \) and \( \mu \) are the Lame coefficients and \( \delta_{ij} \) denotes the Kronecker symbol. On the boundary the normal stress is given by

\[
\sigma_{ij} n_j = 0
\]

(1.33)

for all time \( t > 0 \).

At initial time \( t = 0 \) the displacement \( u_i \) and the velocity \( u_{i,t} \) are given:

\[
u_i(0,x) = \begin{cases} U_0 & \text{for } x \text{ at point charge, } x = x_C \\ 0 & \text{elsewhere} \end{cases} \quad \text{and} \quad u_{i,t}(0,x) = 0
\]

(1.34)

for all \( x \) in the domain.

Here we are modelling a point source at the point \( x_C \), in the numerical solution we set the initial displacement to be \( U_0 \) in a sphere of small radius around the point \( x_C \).

We use an explicit time integration scheme to calculate the displacement field \( u \) at certain time marks \( t^{(n)} \) where

\[
t^{(n)} = t^{(n-1)} + h
\]

with time step size \( h > 0 \). In the following the upper index \( (n) \) refers to values at time \( t^{(n)} \). We use the Verlet scheme with constant time step size \( h \) which is defined by

\[
u^{(n)} = 2u^{(n-1)} - u^{(n-2)} + h^2 a^{(n)}
\]

(1.35)

(1.36)
for all \( n = 2, 3, \ldots \). It is designed to solve a system of equations of the form

\[
u_{tt} = G(u) \quad (1.37)\]

where one sets \( a^{(n)} = G(u^{(n-1)}) \).

In our case \( a^{(n)} \) is given by

\[
\rho a_{i}^{(n)} = \sigma_{ij}^{(n-1)} \quad (1.38)
\]

and boundary conditions

\[
\sigma_{ij}^{(n-1)} n_j = 0 \quad (1.39)
\]

derived from Equation (1.33) where

\[
\sigma_{ij}^{(n-1)} = \lambda u_{k,k}^{(n-1)} \delta_{ij} + \mu (u_{ij}^{(n-1)} + u_{ji}^{(n-1)}). \quad (1.40)
\]

Now we have converted our problem for displacement, \( u^{(n)} \), into a problem for acceleration, \( a^{(n)} \), which now depends on the solution at the previous two time steps, \( u^{(n-1)} \) and \( u^{(n-2)} \).

In each time step we have to solve this problem to get the acceleration \( a^{(n)} \), and we will use the \texttt{LinearPDE} class of the \texttt{esys.escript.linearPDEs} to do so. The general form of the PDE defined through the \texttt{LinearPDE} class is discussed in Section 4.1. The form which is relevant here is

\[
D_{ij} a_{j}^{(n)} = -X_{ij,j} \quad (1.41)
\]

The natural boundary condition

\[
n_{j} X_{ij} = 0 \quad (1.42)
\]

is used.

With \( u = a^{(n)} \) we can identify the values to be assigned to \( D \) and \( X \):

\[
D_{ij} = \rho \delta_{ij}, \quad X_{ij} = -\sigma_{ij}^{(n-1)} \quad (1.43)
\]

The following script defines a the function \texttt{wavePropagation} which implements the Verlet scheme to solve our wave propagation problem. The argument \texttt{domain} which is a \texttt{Domain} class object defines the domain of the problem. \( h \) and \( tend \) are the time step size and the end time of the simulation. \( lam, mu \) and \( rho \) are material properties.

```python
from esys.linearPDEs import LinearPDE
from numarray import identity, zeros, ones
from esys.escript import *
from esys.escript.pdetools import Locator

def wavePropagation(domain,h,tend,lam,mu,rho,U0):
    x=domain.getX()
    # ... open new PDE ...
    mypde=LinearPDE(domain)
    mypde.setSolverMethod(LinearPDE.LUMPING)
    kronecker=identity(mypde.getDim())
    # spherical source at middle of bottom face
    xc=[width/2.,width/2.,0.]
    # define small radius around point xc
    # Lsup(x) returns the maximum value of the argument x
    src_radius = 0.1*Lsup(domain.getSize())
    print "src_radius = ",src_radius
    dunit=numarray.array([1.,0.,0.]) # defines direction of point source
    mypde.setValue(D=kronecker*rho)
    # ... set initial values ....
    n=0
    # initial value of displacement at point source is constant (U0=0.01)
    # for first two time steps
    u=U0*whereNegative(length(x-xc)-src_radius)*dunit
```

Chapter 1. Tutorial: Solving PDEs
u_last=U0*whereNegative(length(x-xc)-src_radius)*dunit

t=0

# define the location of the point source
L=Locator(domain,xc)

# find potential at point source
u_pc=L.getValue(u)

print "u at point charge=",u_pc
u_pc_x = u_pc[0]
u_pc_y = u_pc[1]
u_pc_z = u_pc[2]

# open file to save displacement at point source
u_pc_data=open('./data/U_pc.out','w')

u_pc_data.write("%f %f %f %f
"%(t,u_pc_x,u_pc_y,u_pc_z))

while t<tend:
    # ... get current stress ....
    g=grad(u)
    stress=lam*trace(g)*kronecker+mu*(g+transpose(g))
    # ... get new acceleration ....
    mypde.setValue(X=-stress)
    a=mypde.getSolution()
    # ... get new displacement ...
    u_new=2*u-u_last+h**2*a
    # ... shift displacements ....
    u_last=u
    u=u_new
    t+=h
    n+=1

    print n,"-th time step t ",t
    L=Locator(domain,xc)
    u_pc=L.getValue(u)

    print "u at point charge=",u_pc
    u_pc_x = u_pc[0]
    u_pc_y = u_pc[1]
    u_pc_z = u_pc[2]

    # save displacements at point source to file for t > 0
    u_pc_data.write("%f %f %f %f
"%(t,u_pc_x,u_pc_y,u_pc_z))

    # save current acceleration in units of gravity and displacements
    if n==1 or n%10==0: saveVTK("./data/usoln.%i.vtu"%(n/10),acceleration=length(a)/9.81,
                               displacement = length(u), Ux = u[0] )

u_pc_data.close()
One of the big advantages of the Verlet scheme is the fact that the problem to be solved in each time step is very simple and does not involve any spatial derivatives (which is what allows us to use lumping in this simulation). The problem becomes so simple because we use the stress from the last time step rather then the stress which is actually present at the current time step. Schemes using this approach are called an explicit time integration schemes. The backward Euler scheme we have used in Chapter 1.3 is an example of an implicit scheme. In this case one uses the actual status of each variable at a particular time rather then values from previous time steps. This will lead to a problem which is more expensive to solve, in particular for non-linear problems. Although explicit time integration schemes are cheap to finalize a single time step, they need significantly smaller time steps then implicit schemes and can suffer from stability problems. Therefore they need a very careful selection of the time step size \( h \).

An easy, heuristic way of choosing an appropriate time step size is the Courant condition which says that within a time step a information should not travel further than a cell used in the discretization scheme. In the case of the wave equation the velocity of a (p-) wave is given as \( \sqrt{\frac{\lambda+2\mu}{\rho}} \) so one should choose \( h \) from

\[
    h = \frac{1}{5} \sqrt{\frac{\rho}{\lambda + 2\mu}} \Delta x
\]

where \( \Delta x \) is the cell diameter. The factor \( \frac{1}{5} \) is a safety factor considering the heuristics of the formula.

The following script uses the \texttt{wavePropagation} function to solve the wave equation for a point source located at the bottom face of a block. The width of the block in each direction on the bottom face is 10km (\( x_0 \) and \( x_1 \) directions ie. 10 and 11). The \( ne \) gives the number of elements in \( x_0 \) and \( x_1 \) directions. The depth of the block is aligned with the \( x_2 \)-direction. The depth (12) of the block in the \( x_2 \)-direction is chosen so that there are 10 elements and the magnitude of the of the depth is chosen such that the elements become cubic. We chose 10 for the number of elements in \( x_2 \)-direction so that the computation would be faster. \texttt{Brick}(\( n_0, n_1, n_2, l_0, l_1, l_2 \)) is an \texttt{esys.finley} function which creates a rectangular mesh with \( n_0 \times n_1 \times n_2 \) elements over the brick \([0, l_0] \times [0, l_1] \times [0, l_2] \).

```python
from esys.finley import Brick
ne=32  # number of cells in x_0 and x_1 directions
width=10000. # length in x_0 and x_1 directions
lam=3.462e9
mu=3.462e9
rho=1154.
tend=60
h=(1./5.)*sqrt(rho/(lam+2*mu))*(width/ne)
print "time step size = ",h
U0=0.01 # amplitude of point source
mydomain=Brick(ne,ne,10,l0=width,l1=width,l2=10.*width/32.)
wavePropagation(mydomain,h,tend,lam,mu,rho,U0)
```

The script is available as ‘\texttt{wave.py}’ in the example directory. Before running the code make sure you have created a directory called \texttt{data} in the current working directory. To visualize the results from the data directory:

```
mayavi -d usoln.1.vtu -m SurfaceMap &
```

You can rotate this figure by clicking on it with the mouse and moving it around. Again use \texttt{Configure Data} to move backwards and forward in time, and also to choose the results (acceleration, displacement or \( u_x \)) by using \texttt{Select Scalar}. Figure 1.7 shows the results for the displacement at various time steps.

### 1.5 Elastic Deformation

In this section we want to examine the deformation of a linear elastic body caused by expansion through a heat distribution. We want a displacement field \( u_i \) which solves the momentum equation:

\[
    - \sigma_{ij,j} = 0
\]
where the stress $\sigma$ is given by

$$
\sigma_{ij} = \lambda u_{k,k} \delta_{ij} + \mu (u_{i,j} + u_{j,i}) - (\lambda + \frac{2\mu}{3}) \alpha (T - T_{ref}) \delta_{ij}.
$$

(1.46)

In this formula $\lambda$ and $\mu$ are the Lame coefficients, $\alpha$ is the temperature expansion coefficient, $T$ is the temperature distribution and $T_{ref}$ a reference temperature. Note that Equation (1.45) is similar to eqnWAVE general problem introduced in section Section 1.4 but the inertia term $\rho u_{i,tt}$ has been dropped as we assume a static scenario here. Moreover, in comparison to the Equation (1.32) definition of stress $\sigma$ in Equation (1.46) an extra term is introduced to bring in stress due to volume changes through temperature dependent expansion.

Our domain is the unit cube

$$
\Omega = \{ (x_i | 0 \leq x_i \leq 1) \}
$$

(1.47)

On the boundary the normal stress component is set to zero

$$
\sigma_{ij} n_j = 0
$$

(1.48)

and on the face with $x_i = 0$ we set the $i$-th component of the displacement to 0

$$
u_i(x) = 0 \text{ where } x_i = 0
$$

(1.49)

For the temperature distribution we use

$$
T(x) = T_0 e^{-\beta ||x - x^c||},
$$

(1.50)

with a given positive constant $\beta$ and location $x^c$ in the domain.

When we insert Equation (1.46) we get a second order system of linear PDEs for the displacements $u$ which is called the Lame equation. We want to solve this using the LinearPDE class to this. For a system of PDEs and a solution with several components the LinearPDE class takes PDEs of the form

$$
-A_{ijkl} u_{k,l} = -X_{ij,j}.
$$

(1.51)

$A$ is of rank-$4$ Data object and $X$ is of rank-$2$ Data object. We show here the coefficients relevant for the we trying to solve. The full form is given in Equation (4.4). The natural boundary conditions take the form

$$
n_j A_{ijkl} u_{k,l} = n_j X_{ij}.
$$

(1.52)

Constraints take the form

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

(1.53)
Figure 1.7: Selected time steps (n = 0, 1, 30, 100, 300 and 2880) of a wave propagation over a 10km × 10km × 3.125km block from a point source initially at (5km, 5km, 0) with time step size h = 0.02083. Color represents the displacement. Here the view is oriented onto the bottom face.

r and q are each rank-1 Data object. We can easily identify the coefficients in Equation (1.51):

\[ A_{ijkl} = \lambda \delta_{ij} \delta_{kl} + \mu (\delta_{ik} \delta_{jl} + \delta_{il} \delta_{jk}) \]  

(1.54)

\[ X_{ij} = (\lambda + \frac{2}{3} \mu) \alpha (T - T_{ref}) \delta_{ij} \]  

(1.55)

The characteristic function \( q \) defining the locations and components where constraints are set is given by:

\[ q_{i}(x) = \begin{cases} 
1 & x_{i} = 0 \\
0 & \text{otherwise} \end{cases} \]  

(1.57)

Under the assumption that \( \lambda, \mu, \beta \) and \( T_{ref} \) are constant we may use \( Y_{i} = \lambda + \frac{2}{3} \mu \alpha T_{i} \). However, this choice would lead to a different natural boundary condition which does not set the normal stress component as defined in Equation (1.46) to zero.

Analogously to concept of symmetry for a single PDE, we call the PDE defined by Equation (1.51) symmetric if

\[ A_{ijkl} = A_{klij} \]  

(1.58)

This Lame equation is in fact symmetric, given the difference in \( D \) and \( d \) as compared to the scalar case. The LinearPDE class is notified of this fact by calling its setSymmetryOn method.

After we have solved the Lame equation we want to analyse the actual stress distribution. Typically the von–Mises stress defined by

\[ \sigma_{\text{mises}} = \sqrt{\frac{1}{6}((\sigma_{00} - \sigma_{11})^2 + (\sigma_{11} - \sigma_{22})^2 + (\sigma_{22} - \sigma_{00})^2) + \sigma_{01}^2 + \sigma_{12}^2 + \sigma_{20}^2} \]  

(1.60)

is used to detect material damage. Here we want to calculate the von–Mises and write the stress to a file for visualization.

The following script, which is available in ‘heatedbox.py’ in the example directory, solves the Lame equation and writes the displacements and the von–Mises stress into a file ‘deform.xml’ in the VTK file format:
from esys.escript import *
from esys.escript.linearPDEs import LinearPDE
from esys.finley import Brick

#... set some parameters ...
lam=1.
mu=0.1
alpha=1.e-6
xc=[0.3,0.3,1.]
beta=8.
T_ref=0.
T_0=1.

#... generate domain ...
mydomain = Brick(l0=1.,l1=1., l2=1.,n0=10, n1=10, n2=10)
x=mydomain.getX()

#... set temperature ...
T=T_0*exp(-beta*length(x-xc))

#... open symmetric PDE ...
mypde=LinearPDE(mydomain)
mypde.setSymmetryOn()

#... set coefficients ...
C=Tensor4(0.,Function(mydomain))
for i in range(mydomain.getDim()):
    for j in range(mydomain.getDim()):
        C[i,i,j,j]+=lam
        C[j,i,j,i]+=mu
        C[j,i,i,j]+=mu
msk=whereZero(x[0])*[1.,0.,0.]
    +whereZero(x[1])*[0.,1.,0.]
    +whereZero(x[2])*[0.,0.,1.]

sigma0=(lam+2./3.*mu)*alpha*(T-T_ref)*kronecker(mydomain)
mypde.setValue(A=C,X=sigma0,q=msk)

#... solve pde ...
u=mypde.getSolution()

#... calculate von-Misses stress
q=grad(u)
sigma=mu*(g+transpose(g))+lam*trace(g)*kronecker(mydomain)-sigma0
sigma_mises=sqrt(((sigma[0,0]-sigma[1,1])**2+(sigma[1,1]-sigma[2,2])**2+ 
    (sigma[2,2]-sigma[0,0])**2)/6. 
    +sigma[0,1]**2 + sigma[1,2]**2 + sigma[2,0]**2)

#... output ...
saveVTK("deform.xml",disp=u,stress=sigma_mises)

Finally the the results can be visualize by calling
mayavi -d deform.xml -f CellToPointData -m VelocityVector -m SurfaceMap &

Note that the filter CellToPointData is applied to create smooth representation of the von–Mises stress. Figure 1.8 shows the results where the vertical planes showing the von–Mises stress and the horizontal plane shows the vector representing displacements.

1.6 Stokes Flow

In this section we will look at Computational Fluid Dynamics (CFD) to simulate the flow of fluid under the influence of gravity. The StokesProblemCartesian class will be used to calculate the velocity and pressure of the fluid. The fluid dynamics is governed by the Stokes equation. In geophysical problems the velocity of fluids are low; that is, the inertial forces are small compared with the viscous forces, therefore the inertial terms in the Navier-Stokes equations can be ignored. For a body force, $f$, the governing equations are given by:

$$ \nabla \cdot (\eta (\nabla \vec{v} + \nabla^T \vec{v})) - \nabla p = -f, \quad (1.61) $$

with the incompressibility condition

$$ \nabla \cdot \vec{v} = 0. \quad (1.62) $$
where \( p, \eta \) and \( f \) are the pressure, viscosity and body forces, respectively. Alternatively, the Stokes equations can be represented in Einstein summation tensor notation (compact notation):

\[
-(\eta(v_{i,j} + v_{j,i}))_{ij} - p_i = f_i, \tag{1.63}
\]

with the incompressibility condition

\[
-v_{i,i} = 0. \tag{1.64}
\]

The subscript comma \( i \) denotes the derivative of the function with respect to \( x_i \). The body force \( f \) in Equation (1.63) is the gravity acting in the \( x_3 \) direction and is given as \( f = -g\rho_0\delta_3 \). The Stokes equations is a saddle point problem, and can be solved using a Uzawa scheme. A class called StokesProblemCartesian in Escript can be used to solve for velocity and pressure; more detail on the class can be view in Chapter 7. In order to keep numerical stability, the time-step size needs to be kept below a certain value, to satisfy the Courant condition. The Courant number is defined as:

\[
C = \frac{\nu \delta t}{h}. \tag{1.65}
\]

where \( \delta t, \nu, \) and \( h \) are the time-step, velocity, and the width of an element in the mesh, respectively. The velocity \( v \) may be chosen as the maximum velocity in the domain. In this problem the time-step size was calculated for a Courant number of 0.4.

The following PYTHON script is the setup for the Stokes flow simulation, and is available in the example directory as 'fluid.py'. It starts off by importing the classes, such as the StokesProblemCartesian class, for solving the Stokes equation and the incompressibility condition for velocity and pressure. Physical constants are defined for the viscosity and density of the fluid, along with the acceleration due to gravity. Solver settings are set for the maximum iterations and tolerance; the default solver used is PCG. The mesh is defined as a rectangle, to represent the body of fluid. The gravitational force is calculated base on the fluid density and the acceleration due to gravity. The boundary conditions are set for a slip condition at the base of the mesh; fluid movement in the \( x \)-direction is free, but fixed in the \( y \)-direction. An instance of the StokesProblemCartesian is defined for the given computational mesh, and the solver tolerance set. Inside the while loop, the boundary conditions, viscosity and body force are initialized. The Stokes equation is then solved for velocity and pressure. The time-step size is calculated base on the Courant condition, to ensure stable solutions. The nodes in the mesh are then displaced based on the current velocity and time-step size, to move the body of fluid. The output for the simulation of velocity and pressure is then save to file for visualization.

```python
from esys.escript import *
import esys.finley
from esys.escript.linearPDEs import LinearPDE
from esys.escript.models import StokesProblemCartesian
```
#physical constants
et=1.0
e=100.0
g=10.0

#solver settings
tolerance=1.0e-4
max_iter=200
t_end=50
t=0.0
time=0
verbose='TRUE'
useUzawa='TRUE'

#define mesh
H=2.0
L=1.0
W=1.0
mesh = esys.finley.Rectangle(l0=L, l1=H, order=2, n0=20, n1=20)
coordinates = mesh.getX()

#gravitational force
Y=Vector(0.0, Function(mesh))
Y[1]=e*rho*g

#element spacing
h=Lsup(mesh.getSize())

#boundary conditions for slip at base
boundary_cond=whereZero(coordinates[1])*[0.0,1.0]

#velocity and pressure vectors
velocity=Vector(0.0, ContinuousFunction(mesh))
pressure=Scalar(0.0, ContinuousFunction(mesh))

#Stokes Cartesian
solution=StokesProblemCartesian(mesh)
solution.setTolerance(tolerance)

while t <= t_end:
    print " ----- Time step = %s -----
    print "Time = %s seconds"%( (t )
    print "Time = %s seconds"%( t )

    solution.initialize(fixed_u_mask=boundary_cond,eta=eta,f=Y)
    velocity,pressure=solution.solve(velocity,pressure,max_iter=max_iter, verbose=verbose,useUzawa=useUzawa)

    print "Max velocity =", Lsup(velocity), "m/s"

    #Courant condition
dt=0.4*h/(Lsup(velocity))
    print "dt", dt

    #displace the mesh
    displacement = velocity * dt
    coordinates = mesh.getX()
    mesh.setX(coordinates + displacement)

    time += dt

    vel_mag = length(velocity)
#save velocity and pressure output
saveVTK("vel.%2.2i.vtu"%(t),vel=vel_mag,vec=velocity,pressure=pressure)
t = t+1.0

The results from the simulation can be viewed with mayavi, by executing the following command:

mayavi -d vel.00.vtu -m SurfaceMap

Colour coded scalar maps and velocity flow fields can be viewed by selecting them in the menu. The time-steps can be swept through to view a movie of the simulation. Figures 1.9 and 1.10 shows the simulation output. Velocity vectors and a colour map for pressure are shown. As the time progresses the body of fluid falls under the influence of gravity. The view used here to track the fluid is the Lagrangian view, since the mesh moves with the fluid. One of the disadvantages of using the Lagrangian view is that the elements in the mesh become severely distorted after a period of time and introduce solver errors. To get around this limitation the Level Set Method can be used, with the Eulerian point of view for a fixed mesh.
Figure 1.9: Simulation output for Stokes flow. Fluid body starts off as a rectangular shape, then progresses downwards under the influence of gravity. Color coded distribution represents the scalar values for pressure. Velocity vectors are displayed at each node in the mesh to show the flow field. Computational mesh used was $20 \times 20$ elements.
FIGURE 1.10: Simulation output for Stokes flow.
Execution of an *escript* Script

2.1 Overview

A typical way of starting your *escript* script ‘*myscript.py*’ is with the *escript* command:\(^1\):

```
escript myscript.py
```

as already shown in section 1.2\(^2\). In some cases it can be useful to work interactively e.g. when debugging a script, with the command

```
escript -i myscript.py
```

This will execute *myscript.py* and when it completes (or an error occurs), a *python* prompt will be provided. To leave the prompt press `Control-d`.

To start *escript* using four threads (e.g. if you use a multi-core processor) you can use

```
escript -t 4 myscript.py
```

This will require *escript* to be compiled for *OpenMP* \(^16\).

To start *escript* using *MPI* \(^12\) with 8 processes you use

```
escript -p 8 myscript.py
```

If the processors which are used are multi-core processors or multi-processor shared memory architectures you can use threading in addition to *MPI*. For instance to run 8 *MPI* processes with using 4 threads each, you use the command

```
escript -p 8 -t 4 myscript.py
```

In the case of a super computer or a cluster, you may wish to distribute the workload over a number of nodes\(^3\). For example, to use 8 nodes, with 4 *MPI* processes per node, write

```
escript -n 8 -p 4 myscript.py
```

Since threading has some performance advantages over processes, you may specify a number of threads as well.

```
escript -n 8 -p 4 -t 2 myscript.py
```

This runs the script on 8 nodes, with 4 processes per node and 2 threads per process.

---

\(^1\)The *escript* launcher is not supported under *MS Windows* yet.

\(^2\)For this discussion, it is assumed that *escript* is included in your *PATH* environment. See installation guide for details.

\(^3\)For simplicity, we will use the term node to refer to either a node in a super computer or an individual machine in a cluster.
2.2 Options

The general form of the escript launcher is as follows:

```
```

where file is the name of a script, ARGS are arguments for the script. The escript program will import your current environment variables. If no file is given, then you will be given a python prompt (see -i for restrictions).

The options are used as follows:

- **-n nn** the number of compute nodes nn to be used. The total number of process being used is nn · ns. This option overwrites the value of the ESCRIPT_NUM_NODES environment variable. If a hostfile is given, the number of nodes needs to match the number hosts given in the host file. If nn > 1 but escript is not compiled for MPI a warning is printed but execution is continued with nn = 1. If -n is not set the number of hosts in the host file is used. The default value is 1.

- **-p np** the number of MPI processes per node. The total number of processes to be used is nn · np. This option overwrites the value of the ESCRIPT_NUM_PROCS environment variable. If np > 1 but escript is not compiled for MPI a warning is printed but execution is continued with np = 1. The default value is 1.

- **-t nt** the number of threads used per processes. The option overwrites the value of the ESCRIPT_NUM_THREADS environment variable. If nt > 1 but escript is not compiled for OpenMP a warning is printed but execution is continued with nt = 1. The default value is 1.

- **-f hostfile** the name of a file with a list of host names. Some systems require to specify the addresses or names of the compute nodes where MPI process should be spawned. The list of addresses or names of the compute nodes is listed in the file with the name hostfile. If -n is set the the number of different hosts defined in hostfile must be equal to the number of requested compute nodes nn. The option overwrites the value of the ESCRIPT_HOSTFILE environment variable. By default value no host file is used.

- **-c** prints the information about the settings used to compile escript and stops execution.

- **-V** prints the version of escript and stops execution.

- **-h** prints a help message and stops execution.

- **-i** executes the script file and switches to interactive mode after the execution is finished or an exception has occurred. This option is useful for debugging a script. The option cannot be used if more then one process (nn · np > 1) is used.

- **-b** do not invoke python. This is used to run non-python programs.

- **-e** shows additional environment variables and commands used during escript execution. This option is useful if users wish to execute scripts without using the escript command.

- **-o** switches on the redirection of output of processors with MPI rank greater than zero to the files 'stdout_r.out' and 'stderr_r.out' where r is the rank of the processor. The option overwrites the value of the ESCRIPT_STDFILES environment variable

- **-v** prints some diagnostic information.

### 2.2.1 Notes

- Make sure that mpiexec is in your PATH.

- For MPICH and INTELMPI and for the case a hostfile is present escript will start the mpd demon before execution.
2.3 Input and Output

When MPI is used on more than one process \((nn \cdot np > 1)\) no input from the standard input is accepted. Standard output on any process other the the master process \((rank=0)\) will not be available. Error output from any processor will be redirected to the node where escript has been invoked. If the `-o` or `ESCRIPT_STDFILES` is set\(^4\), then the standard and error output from any process other than the master process will be written to files of the names `stdout_r.out` and `stderr_r.out` (where \(r\) is the rank of the process).

If files are created or read by individual MPI processes with information local to the process (e.g. in the `dump` function) and more than one process is used \((nn \cdot np > 1)\), the MPI process rank is appended to the file names. This will avoid problems if processes are using a shared file system. Files which collect data which are global for all MPI processors will created by the process with MPI `rank` 0 only. Users should keep in mind that if the file system is not shared, then a file containing global information which is read by all processors needs to be copied to the local file system before `escript` is invoked.

2.4 Hints for MPI Programming

In general a script based on the `esys.escript` module does not require modifications when running under MPI. However, one needs to be careful if other modules are used.

When MPI is used on more than one process \((nn \cdot np > 1)\) the user needs to keep in mind that several copies of his script are executed at the same time\(^5\) while data exchange is performed through the `esys.escript` module. At any time, `esys.escript` assumes that an argument of the type `int`, `float`, `str` and `numarray` has an identical value across all processors. All values of these types returned by `esys.escript` have the same value on all processors. If values produced by other modules are used as argument the user has to make sure that the value is identical on all processors. For instance, the usage of a random number generator to create a value for argument bears the risk that the value may depend on the processor.

Another case which needs special attention is the usage of files. When reading data from a file it advisable to use the `r` for readable when opened. Keep in mind that several scripts will simultaneously access the file. If data are written to a file only one processor must open the file for writing. The function `getMPIRankWorld` which returns the processor id between 0 and the number of processors helps to achieve this. The following script writes to the file `test.txt` on the processor with id 0 only:

```python
from esys.escript import *
if getMPIRankWorld() == 0 :
    f = open('test.txt', 'w')
    f.write('test message')
    f.close()
```

Another technique is to extend the file name by the processor id to avoid conflicts while writing into a shared file system:

```python
from esys.escript import *
f = open('test.txt.%s'%getMPIRankWorld(), 'w')
f.write('test message')
f.close()
```

creates files with names `test.txt.0, test.txt.1, test.txt.2, . . .` .

If there is the situation that if execution on one of the processors is throwing an exception, for instance as opening a file for writing fails, are not made aware of this as MPI is not handling exceptions. However, MPI will terminate the other processes but may not inform the user of the reason in the obvious way. The user needs to inspect the error output files to identify the exception.

\(^4\)That is, it has a non-empty value.
\(^5\)In case of OpenMP only one copy is running but `esys.escript` temporarily spawns threads.
esys.escript is a Python module that allows you to represent the values of a function at points in a Domain in such a way that the function will be useful for the Finite Element Method (FEM) simulation. It also provides what we call a function space that describes how the data is used in the simulation. Stored along with the data is information about the elements and nodes which will be used by esys.finley.

In order to understand what we mean by the term ’function space’, consider that the solution of a partial differential equation (PDE) is a function on a domain $\Omega$. When solving a PDE using FEM, the solution is piecewise-differentiable but, in general, its gradient is discontinuous. To reflect these different degrees of smoothness, different function spaces are used. For instance, in FEM, the displacement field is represented by its values at the nodes of the mesh, and so is continuous. The strain, which is the symmetric part of the gradient of the displacement field, is stored on the element centers, and so is considered to be discontinuous.

A function space is described by a FunctionSpace object. 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:

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

The following generators for function spaces on a Domain mydomain are available:

- `Solution(mydomain)`: solutions of a PDE.
- `ReducedSolution(mydomain)`: solutions of a PDE with a reduced smoothness requirement.
- `ContinuousFunction(mydomain)`: continuous functions, eg. a temperature distribution.
- `Function(mydomain)`: general functions which are not necessarily continuous, eg. a stress field.
- `FunctionOnBoundary(mydomain)`: functions on the boundary of the domain, eg. a surface pressure.
- `FunctionOnContact0(mydomain)`: functions on side 0 of the discontinuity.
- `FunctionOnContact1(mydomain)`: functions on side 1 of the discontinuity.

The reduced smoothness for PDE solution is often used to fulfill the Ladyzhenskaya-Babuska-Brezzi condition [10] when solving saddle point problems, eg. the Stokes equation. A discontinuity is a region within the domain across which functions may be discontinuous. The location of discontinuity is defined in the Domain object. Figure 3.1 shows the dependency between the types of function spaces in Finley (other libraries may have different relationships).

The 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 one side of discontinuity (the result will be different depending on which side is chosen). Functions on any side of the discontinuity can be seen as a function on the corresponding other 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 is identical, however in some cases, eg. when periodic boundary conditions are used in esys.finley, a solution fulfills periodic boundary conditions while a continuous function does not have to be periodic.
The concept of function spaces describes the properties of functions and allows abstraction from the actual representation of the function in the context of a particular application. For instance, in the FEM context a function of the general FunctionSpace type (written as Function() in Figure 3.1) is usually represented by its values at the element center, but in a finite difference scheme the edge midpoint of cells is preferred. By changing its function space you can use the same function in a Finite Difference scheme instead of Finite Element scheme. 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 must be interpolated to the element centers or the cell edges, respectively. Interpolation happens automatically in esys.escript whenever it is required.

In esys.escript the class that stores these functions is called Data. The function is represented through its values on data sample points where the data sample points are chosen according to the function space of the function. Data class objects are used to define the coefficients of the PDEs to be solved by a PDE solver library and also to store the solutions of the PDE.

The values of the function have a rank which gives the number of indices, and a shape defining the range of each index. The rank in esys.escript is limited to the range 0 through 4 and it is assumed that the rank and shape is the same for all data sample points. The shape of a Data object is a tuple (list) of integers. The length of s is the rank of the Data object and the i-th index ranges between 0 and s[i] − 1. 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 with values of shape (2, 3) and rank 2:

```
mydat=Data(value=1,what=ContinuousFunction(myDomain),shape=(2,3))
```

The initial value is the constant 1 for all data sample points and all components. Data objects can also be created from any numarray array or any object, such as a list of floating point numbers, that can be converted into a numarray.NumArray [17]. The following two statements create objects which are equivalent to mydat:

```
mydat1=Data(value=numarray.ones((2,3)),what=ContinuousFunction(myDomain))
mydat2=Data(value=[[1,1],[1,1],[1,1]],what=ContinuousFunction(myDomain))
```
In the first case the initial value is `numarray.ones((2,3))` which generates a $2 \times 3$ matrix as a `numarray.NumArray` filled with ones. The shape of the created `Data` object is taken from the shape of the array. In the second case, the creator converts the initial value, which is a list of lists, and converts it into a `numarray.NumArray` before creating the actual `Data` object.

For convenience `esys.escript` provides creators for the most common types of `Data` objects in the following forms ($d$ defines the spatial dimension):

- `Scalar(0,Function(mydomain))` is the same as `Data(0,Function(myDomain),(,))` (each value is a scalar), e.g. a temperature field.
- `Vector(0,Function(mydomain))` is the same as `Data(0,Function(myDomain),(d))` (each value is a vector), e.g. a velocity field.
- `Tensor(0,Function(mydomain))` is the same as `Data(0,Function(myDomain),(d,d))`, e.g. a stress field.
- `Tensor4(0,Function(mydomain))` is the same as `Data(0,Function(myDomain),(d,d,d,d))` e.g. a Hook tensor field.

Here the initial value is 0 but any object that can be converted into a `numarray.NumArray` and whose shape is consistent with the shape of the `Data` object to be created can be used as the initial value.

`Data` objects can be manipulated by applying unary operations (e.g. cos, sin, log) point and can be combined point-wise by applying arithmetic operations (e.g. +, -, *, /). It is to be emphasized that `esys.escript` itself does not handle any spatial dependencies as it does not know how values are interpreted by the processing PDE solver library. However `esys.escript` invokes interpolation if this is needed during data manipulations. Typically, this occurs in binary operation when both arguments belong to different function spaces or when data are handed over to a PDE solver library which requires functions to be represented in a particular way.

The following example shows the usage of `Data` objects: Assume we have a displacement field $u$ and we want to calculate the corresponding stress field $\sigma$ using the linear–elastic isotropic material model

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

where $\delta_{ij}$ is the Kronecker symbol and $\lambda$ and $\mu$ are the Lame coefficients. The following function takes the displacement $u$ and the Lame coefficients $\text{lam}$ and $\text{mu}$ as arguments and returns the corresponding stress:

```python
from esys.escript import *

def getStress(u,lam,mu):
    d=u.getDomain().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 $d-1$. The call `grad(u)` requires the displacement field $u$ to be in the `Solution` or continuous `FunctionSpace` function space. The result $g$ as well as the returned stress will be in the general `FunctionSpace` function space. If, for example, $u$ is the solution of a PDE then `getStress` might be called in the following way:

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

However `getStress` can also be called with `Data` objects as values for `lam` and `mu` which, for instance in the case of a temperature dependency, are calculated by an expression. The following call is equivalent to the previous example:

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

The function `lam` belongs to the continuous `FunctionSpace` function space but with $g$ the function `trace(g)` is in the general `FunctionSpace` function space. In the evaluation of the product `lam*trace(g)` we have different function spaces (on the nodes versus in the centers) and at first glance we have incompatible data.
esys.escript converts the arguments in an appropriate function space according to Table 3.1. In this example that means esys.escript sees \( \text{lam} \) as a function of the general FunctionSpace function space. In the context of FEM this means the nodal values of \( \text{lam} \) are interpolated to the element centers. The interpolation is automatic and requires no special handling.

Material parameters such as the Lame coefficients are typically dependent on rock types present in the area of interest. A common technique to handle these kinds of material parameters is "tagging", which uses storage efficiently. Figure 3.2 shows an example. In this case two rock types \textit{white} and \textit{gray} can be found in the domain. The domain is subdivided into triangular shaped cells. Each cell has a tag indicating the rock type predominately found in this cell. Here \( 1 \) is used to indicate rock type \textit{white} and \( 2 \) for rock type \textit{gray}. The tags are assigned at the time when the cells are generated and stored in the Domain class object. To allow easier usage of tags, names can be used instead of numbers. These names are typically defined at the time when the geometry is generated.

The following statements show how, for the example of Figure 3.2, the stress calculation discussed above and tagged values are used for \textit{lam}:

\begin{verbatim}
lam=Scalar(value=2.,what=Function(mydomain))
insertTaggedValue(lam,white=30.,gray=5000.)
s=getStress(u,lam,2.)
\end{verbatim}

In this example \textit{lam} is set to 30 for those cells with tag \textit{white} (=1) and to 5000. for those cells with tag \textit{gray} (=2). The initial value 2 of \textit{lam} is used as a default value for the case when a tag is encountered which has not been linked with a value. The \textit{getStress} method does not need to be changed now that we are using tags. esys.escript resolves the tags when \( \text{lam} \cdot \text{trace}(g) \) is calculated.

This brings us to a very important point about esys.escript. You can develop a simulation with constant Lame coefficients, and then later switch to tagged Lame coefficients without otherwise changing your python script. In short, you can use the same script to model with different domains and different types of input data.

There are three main ways in which Data objects are represented internally: constant, tagged, and expanded. In
the constant case, the same value is used at each sample point and only a single value is stored to save memory. In the expanded case, each sample point has an individual value (such as for the solution of a PDE). This is where your largest data sets will be created because the values are stored as a complete array. The tagged case has already been discussed above.

Expanded data is created when you create a Data object with expanded=True. Tagged data sets are created when you use the insertTaggedValue() method as shown above.

Values are accessed through a sample reference number. Operations on expanded Data objects have to be performed for each sample point individually. When tagged values are used, the values are held in a dictionary. Operations on tagged data require processing the set of tagged values only, rather than processing the value for each individual sample point. esys.e script allows any mixture of constant, tagged and expanded data in a single expression.

Data objects can be written to disk files and read with dump and load, both of which use netCDF [14]. Use these to save data for visualization, checkpoint/restart or simply to save and reuse data that was expensive to compute.

For instance to save the coordinates of the data points of the continuous FunctionSpace to the file x.nc use

```python
x=ContinuousFunction(mydomain).getX()
x.dump("x.nc")
```

To recover the object x use

```python
x=load("x.nc", mydomain)
```

The dump file x.nc does not contain a representation of the Domain, even though it is required to recreate x. It is common to simply recreate the Domain before reading a Data, or you may read and write your Domain in a separate file with domain=ReadMesh(fileName) and domain.write(fileName).

The function space of the Data is stored in x.nc, though. If the Data object is expanded, the number of data points in the file and of the Domain for the particular FunctionSpace must match. Moreover, the ordering of the values is checked using the reference identifiers provided by FunctionSpace on the Domain. In some cases, data points will be re-ordered. Take care to be sure you get what you want!

### 3.1 esys.e script Classes

#### 3.1.1 Domain class

A Domain object is used to describe a geometric region together with a way of representing functions over this region. The Domain class provides an abstract interface to the domain of FunctionSpace and Data objects. Domain needs to be subclassed in order to provide a complete implementation.

The following methods are available:

- `getDim()`
  - Returns the spatial dimension of the Domain.

- `getX()`
  - Returns the locations in the Domain. The FunctionSpace of the returned Data object is chosen by the Domain implementation. Typically it will be in the general FunctionSpace.

- `setX(newX)`
  - Assigns a new location to the Domain. newX has to have shape \(d\) where \(d\) is the spatial dimension of the domain. Typically newX must be in the continuous FunctionSpace but the space actually to be used depends on the Domain implementation.

- `getNormal()`
  - Returns the surface normals on the boundary of the Domain as Data object.

- `getSize()`
  - Returns the local sample size, e.g. the element diameter, as Data object.
setTagMap (tag_name, tag)
  defines a mapping of the tag name tag_name to the tag.

getag (tag_name)
  returns the tag associated with the tag name tag_name.

isValidTagName (tag_name)
  return True if tag_name is a valid tag name.

__eq__ (arg)
  (python == operator) returns True if the Domain arg describes the same domain. Otherwise False is returned.

__ne__ (arg)
  (python != operator) returns True if the Domain arg does not describe the same domain. Otherwise False is returned.

__str__ (arg)
  (python str() function) returns string representation of the Domain.

onMasterProcessor ()
  returns True if the processor is the master processor within the MPI processor group used by the Domain. This is the processor with rank 0. If MPI support is not enabled the return value is always True.

getMPISize ()
  returns the number of MPI processors used for this Domain. If MPI support is not enabled 1 is returned.

getMPIRank ()
  returns the rank of the processor executing the statement within the MPI processor group used by the Domain. If MPI support is not enabled 0 is returned.

MPIBarrier ()
  executes barrier synchronization within the MPI processor group used by the Domain. If MPI support is not enabled, this command does nothing.

3.1.2 FunctionSpace class

class FunctionSpace ()
  FunctionSpace objects are used to define properties of Data objects, such as continuity. FunctionSpace objects are instantiated by generator functions. A Data object in a particular FunctionSpace is represented by its values at data sample points which are defined by the type and the Domain of the FunctionSpace.

The following methods are available:

getDim ()
  returns the spatial dimension of the Domain of the FunctionSpace.

getX ()
  returns the location of the data sample points.

getNormal ()
  If the domain of functions in the FunctionSpace is a hypermanifold (e.g. the boundary of a domain) the method returns the outer normal at each of the data sample points. Otherwise an exception is raised.

getSize ()
  returns a Data objects measuring the spacing of the data sample points. The size may be zero.

getDomain ()
  returns the Domain of the FunctionSpace.

setTags (new_tag, mask)
  assigns a new tag new_tag to all data sample where mask is positive for a least one data point. mask must be defined on the this FunctionSpace. Use the setTagMap to assign a tag name to new_tag.

__eq__ (arg)
  (python == operator) returns True if the Domain arg describes the same domain. Otherwise False is
returned.

```
__ne__(arg)
```
(python != operator) returns True if the Domain arg do not describe the same domain. Otherwise False is returned.

```
__str__(g)
```
(python str() function) returns string representation of the Domain.

The following function provide generators for FunctionSpace objects:
```
Function(domain)
```
returns the general FunctionSpace on the Domain domain. Data objects in this type of general FunctionSpace are defined over the whole geometric region defined by domain.

```
ContinuousFunction(domain)
```
returns the continuous FunctionSpace on the Domain domain. Data objects in this type of general FunctionSpace are defined over the whole geometric region defined by domain and assumed to represent a continuous function.

```
FunctionOnBoundary(domain)
```
returns the continuous FunctionSpace on the Domain domain. Data objects in this type of general FunctionSpace are defined on the boundary of the geometric region defined by domain.

```
FunctionOnContactZero(domain)
```
returns the contact FunctionSpace on side 0 the Domain domain. Data objects in this type of general FunctionSpace are defined on side 0 of a discontinuity within the geometric region defined by domain. The discontinuity is defined when domain is instantiated.

```
FunctionOnContactOne(domain)
```
returns the contact FunctionSpace on side 1 on the Domain domain. Data objects in this type of general FunctionSpace are defined on side 1 of a discontinuity within the geometric region defined by domain. The discontinuity is defined when domain is instantiated.

```
Solution(domain)
```
returns the solution FunctionSpace on the Domain domain. Data objects in this type of general FunctionSpace are defined on geometric region defined by domain and are solutions of partial differential equations.

```
ReducedSolution(domain)
```
returns the reduced solution FunctionSpace on the Domain domain. Data objects in this type of general FunctionSpace are defined on geometric region defined by domain and are solutions of partial differential equations with a reduced smoothness for the solution approximation.

### 3.1.3 Data Class

The following table shows arithmetic operations that can be performed point-wise on Data objects.

<table>
<thead>
<tr>
<th>expression</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>+arg0</td>
<td>identical to arg</td>
</tr>
<tr>
<td>-arg0</td>
<td>negation</td>
</tr>
<tr>
<td>arg0+arg1</td>
<td>adds arg0 and arg1</td>
</tr>
<tr>
<td>arg0*arg1</td>
<td>multiplies arg0 and arg1</td>
</tr>
<tr>
<td>arg0-arg1</td>
<td>difference arg1 from arg1</td>
</tr>
<tr>
<td>arg0/arg1</td>
<td>divide arg0 by arg1</td>
</tr>
<tr>
<td>arg0**arg1</td>
<td>raises arg0 to the power of arg1</td>
</tr>
</tbody>
</table>

At least one of the arguments arg0 or arg1 must be a Data object. Either of the arguments may be a Data object, a python number or a numarray object.

If arg0 or arg1 are not defined on the same FunctionSpace, then an attempt is made to convert arg0 to the FunctionSpace of arg1 or to convert arg1 to the FunctionSpace of arg0. Both arguments must have the same shape or one of the arguments may be of rank 0 (a constant).
The returned `Data` object has the same shape and is defined on the data sample points as `arg0` or `arg1`.

The following table shows the update operations that can be applied to `Data` objects:

<table>
<thead>
<tr>
<th>expression</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>arg0+=arg2</code></td>
<td>adds <code>arg0</code> to <code>arg2</code></td>
</tr>
<tr>
<td><code>arg0*=arg2</code></td>
<td>multiplies <code>arg0</code> with <code>arg2</code></td>
</tr>
<tr>
<td><code>arg0-=arg2</code></td>
<td>subtracts <code>arg2</code> from <code>arg0</code></td>
</tr>
<tr>
<td><code>arg0/=arg2</code></td>
<td>divides <code>arg0</code> by <code>arg2</code></td>
</tr>
<tr>
<td><code>arg0**=arg2</code></td>
<td>raises <code>arg0</code> by <code>arg2</code></td>
</tr>
</tbody>
</table>

`arg0` must be a `Data` object. `arg1` must be a `Data` object or an object that can be converted into a `Data` object. `arg1` must have the same shape as `arg0` or have rank 0. In the latter case it is assumed that the values of `arg1` are constant for all components. `arg1` must be defined in the same `FunctionSpace` as `arg0` or it must be possible to interpolate `arg1` onto the `FunctionSpace` of `arg0`.

The `Data` class supports taking slices from a `Data` object as well as assigning new values to a slice of an existing `Data` object. The following expressions for taking and setting slices are valid:

<table>
<thead>
<tr>
<th>rank of <code>arg</code></th>
<th>slicing expression</th>
<th>shape of returned and assigned object</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>no slicing</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td><code>arg[l0:u0]</code></td>
<td><code>(u0-l0,)</code></td>
</tr>
<tr>
<td>2</td>
<td><code>arg[l0:u0,l1:u1]</code></td>
<td><code>(u0-l0,u1-l1)</code></td>
</tr>
<tr>
<td>3</td>
<td><code>arg[l0:u0,l1:u1,l2:u2]</code></td>
<td><code>(u0-l0,u1-l1,u2-l2)</code></td>
</tr>
<tr>
<td>4</td>
<td><code>arg[l0:u0,l1:u1,l2:u2,l3:u3]</code></td>
<td><code>(u0-l0,u1-l1,u2-l2,u3-l3)</code></td>
</tr>
</tbody>
</table>

where `s` is the shape of `arg` and

\[
0 \leq l0 \leq u0 \leq s[0], \\
0 \leq l1 \leq u1 \leq s[1], \\
0 \leq l2 \leq u2 \leq s[2], \\
0 \leq l3 \leq u3 \leq s[3].
\]

Any of the lower indexes `l0`, `l1`, `l2` and `l3` may not be present in which case 0 is assumed. Any of the upper indexes `u0`, `u1`, `u2` and `u3` may be omitted, in which case, the upper limit for that dimension is assumed. The lower and upper index may be identical, in which case the column and the lower or upper index may be dropped. In the returned or in the object assigned to a slice, the corresponding component is dropped, i.e. the rank is reduced by one in comparison to `arg`.

The following examples show slicing in action:

```python
t=Data(1.,(4,4,6,6),Function(mydomain))
t[1,1,0]=9. 
s=t[:2,:2:6,5] # s has rank 3 
s[:,:,1]=1. 
t[:2,:2,5,5]=s[2:4,1,:2]
```

### 3.1.4 Generation of `Data` objects

**class `Data`(value=0,shape=(),what=FunctionSpace(),expand=False)**

creates a `Data` object with shape `shape` in the `FunctionSpace` `what`. The values at all data sample points are set to the double value `value`. If `expanded` is True the `Data` object is represented in expanded form.

**class `Data`(value,what=FunctionSpace(),expand=False)**

creates a `Data` object in the `FunctionSpace` `what`. The value for each data sample points is set to `value`, which could be a `numarray`, `Data` object `value` or a dictionary of `numarray` or floating point numbers. In the latter case the keys must be integers and are used as tags. The shape of the returned object is equal to the shape of `value`. If `expanded` is True the `Data` object is represented in expanded form.

**class `Data()`**

creates an empty `Data` object. The empty `Data` object is used to indicate that an argument is not present where a `Data` object is required.
Scalar \text{(value=0.,what=FunctionSpace(),expand=False)}
returns a Data object of rank 0 (a constant) in the FunctionSpace what. Values are initialised with value, a double precision quantity. If expanded is True the Data object is represented in expanded from.

Vector \text{(value=0.,what=FunctionSpace(),expand=False)}
returns a Data object of shape (d,) in the FunctionSpace what, where d is the spatial dimension of the Domain of what. Values are initiated with value, a double precision quantity. If expanded is True the Data object is represented in expanded from.

Tensor \text{(value=0.,what=FunctionSpace(),expand=False)}
returns a Data object of shape (d,d) in the FunctionSpace what, where d is the spatial dimension of the Domain of what. Values are initiated with value, a double precision quantity. If expanded is True the Data object is represented in expanded from.

Tensor3 \text{(value=0.,what=FunctionSpace(),expand=False)}
returns a Data object of shape (d,d,d) in the FunctionSpace what, where d is the spatial dimension of the Domain of what. Values are initiated with value, a double precision quantity. If expanded is True the Data object is represented in expanded from.

Tensor4 \text{(value=0.,what=FunctionSpace(),expand=False)}
returns a Data object of shape (d,d,d,d) in the FunctionSpace what, where d is the spatial dimension of the Domain of what. Values are initiated with value, a double precision quantity. If expanded is True the Data object is represented in expanded from.

load \text{(filename,domain)}
recovers a Data object on Domain domain from the file filename, which was created by dump.

3.1.5 Data methods

These are the most frequently-used methods of the Data class. A complete list of methods can be found on http://shake200.esscc.uq.edu.au/esys/docs.html.

getFunctionSpace()
returns the FunctionSpace of the object.

defDomain()
returns the Domain of the object.

getShape()
returns the shape of the object as a tuple of integers.

getRank()
returns the rank of the data on each data point.

isEmpty()
returns True if the Data object is the empty Data object. Otherwise False is returned. Note that this is not the same as asking if the object contains no data sample points.

setTaggedValue(tag name,value)
assigns the value to all data sample points which have the tag assigned to tag name. value must be an object of class numarray.NumArray or must be convertible into a numarray.NumArray object. value (or the corresponding numarray.NumArray object) must be of rank 0 or must have the same rank like the object. If a value has already be defined for tag tag name within the object it is overwritten by the new value. If the object is expanded, the value assigned to data sample points with tag tag name is replaced by value. If no tag is assigned tag name tag name, no value is set.

dump \text{(filename)}
dumps the Data object to the file filename. The file stores the function space but not the Domain. It is in the responsibility of the user to save the Domain.

__str__()
returns a string representation of the object.

3.1. esys.escript Classes 35
3.1.6 Functions of Data objects

This section lists the most important functions for Data class objects. A complete list and a more detailed description of the functionality can be found on [http://shake200.esscc.uq.edu.au/esys/docs.html](http://shake200.esscc.uq.edu.au/esys/docs.html).

**saveVTK**(filename,**kwdata)

writes Data defined by keywords in the file with filename using the vtk file format VTK file format. The key word is used as an identifier. The statement

```
saveVTK("out.xml",temperature=T,velocity=v)
```

will write the scalar \( T \) as temperature and the vector \( v \) as velocity into the file ‘out.xml’. Restrictions on the allowed combinations of FunctionSpace apply.

**saveDX**(filename,**kwdata)

writes Data defined by keywords in the file with filename using the vtk file format OpenDX [15] file format. The key word is used as an identifier. The statement

```
saveDX("out.dx",temperature=T,velocity=v)
```

will write the scalar \( T \) as temperature and the vector \( v \) as velocity into the file ‘out.dx’. Restrictions on the allowed combinations of FunctionSpace apply.

**kronecker** \((d)\)

returns a rank-2 Data object Data object in FunctionSpace \( d \) such that

\[
\text{kronecker}(d) \left[ i,j \right] = \begin{cases} 
1 & \text{if } i = j \\
0 & \text{otherwise}
\end{cases} \quad (3.2)
\]

If \( d \) is an integer a \((d,d)\) numarray array is returned.

**identityTensor** \((d)\)

is a synonym for kronecker (see above).

**identityTensor4** \((d)\)

returns a rank-4 Data object Data object in FunctionSpace \( d \) such that

\[
\text{identityTensor}(d) \left[ i,j,k,l \right] = \begin{cases} 
1 & \text{if } i = k \text{ and } j = l \\
0 & \text{otherwise}
\end{cases} \quad (3.3)
\]

If \( d \) is an integer a \((d,d,d,d)\) numarray array is returned.

**unitVector** \((i,d)\)

returns a rank-1 Data object Data object in FunctionSpace \( d \) such that

\[
\text{identityTensor}(d) \left[ j \right] = \begin{cases} 
1 & \text{if } j = i \\
0 & \text{otherwise}
\end{cases} \quad (3.4)
\]

If \( d \) is an integer a \((d,\)\) numarray array is returned.

**Lsup** \((a)\)

returns the \( L^{sup} \) norm of arg. This is the maximum of the absolute values over all components and all data sample points of \( a \).

**sup** \((a)\)

returns the maximum value over all components and all data sample points of \( a \).

**inf** \((a)\)

returns the minimum value over all components and all data sample points of \( a \).

**minval** \((a)\)

returns at each data sample points the minimum value over all components.

**maxval** \((a)\)

returns at each data sample points the maximum value over all components.
length(a)
returns at Euclidean norm at each data sample points. For a rank-4 Data object a this is

\[ \text{length}(a) = \sqrt{\sum_{ijkl} a[i,j,k,l]^2} \]  
\[ (3.5) \]

trace(a[\text{axis} \_\text{offset}=0])
returns the trace of a. This is the sum over components \text{axis} \_\text{offset} and \text{axis} \_\text{offset}+1 with the same index. For instance in the case of a rank-2 Data object function and this is

\[ \text{trace}(a) = \sum_i a[i,i] \]  
\[ (3.6) \]

and for a rank-4 Data object function and \text{axis} \_\text{offset}=1 this is

\[ \text{trace}(a,1) [i,j] = \sum_k a[i,k,k,j] \]  
\[ (3.7) \]

transpose(a[\text{axis} \_\text{offset}=\text{None}])
returns the transpose of a. This swaps the first \text{axis} \_\text{offset} components of a with the rest. If \text{axis} \_\text{offset} is not present \text{int} (r/2) is used where r is the rank of a the sum over components \text{axis} \_\text{offset} and \text{axis} \_\text{offset}+1 with the same index. For instance in the case of a rank-2 Data object function and this is

\[ \text{transpose}(a) [i,j] = a[j,i] \]  
\[ (3.8) \]

and for a rank-4 Data object function and \text{axis} \_\text{offset}=1 this is

\[ \text{transpose}(a,1) [i,j,k,l] = a[j,k,l,i] \]  
\[ (3.9) \]

swap_axes(a[\text{axis}0=0, \text{axis}1=1])
returns a but with swapped components \text{axis}0 and \text{axis}1. The argument a must be at least of rank-2 Data object. For instance in the for a rank-4 Data object argument, \text{axis}0=1 and \text{axis}1=2 this is

\[ \text{swap} \_\text{axes}(a,1,2) [i,j,k,l] = a[i,k,j,l] \]  
\[ (3.10) \]

symmetric(a)
returns the symmetric part of a. This is \( (a+\text{transpose}(a)) / 2 \).

nonsymmetric(a)
returns the non–symmetric part of a. This is \( (a-\text{transpose}(a)) / 2 \).

inverse(a)
return the inverse of a. This is

\[ \text{matrix} \_\text{mult}(\text{inverse}(a),a) = \text{kronecker}(d) \]  
\[ (3.11) \]

if a has shape \( (d,d) \). The current implementation is restricted to arguments of shape \( (2,2) \) and \( (3,3) \).

eigenvalues(a)
return the eigenvalues of a. This is

\[ \text{matrix} \_\text{mult}(a,V) = e[i] \cdot V \]  
\[ (3.12) \]

where \( e = \text{eigenvalues}(a) \) and V is suitable non–zero vector V. The eigenvalues are ordered in increasing size. The argument a has to be the symmetric, ie. \( a = \text{symmetric}(a) \). The current implementation is restricted to arguments of shape \( (2,2) \) and \( (3,3) \).

eigenvalues_and_eigenvectors(a)
return the eigenvalues and eigenvectors of a. This is

\[ \text{matrix} \_\text{mult}(a,V[:,i]) = e[i] \cdot V[:,i] \]  
\[ (3.13) \]

where \( e, V = \text{eigenvalues} \_\text{and} \_\text{eigenvectors}(a) \). The eigenvectors V are orthogonal and normalized, ie.

\[ \text{matrix} \_\text{mult}(\text{transpose}(V),V) = \text{kronecker}(d) \]  
\[ (3.14) \]

if a has shape \( (d,d) \). The eigenvalues are ordered in increasing size. The argument a has to be the symmetric, ie. \( a = \text{symmetric}(a) \). The current implementation is restricted to arguments of shape \( (2,2) \) and \( (3,3) \).
\section*{The Module esys.escript}

\begin{verbatim}
maximum(*a)
returns the maximum value over all arguments at all data sample points and for each component. For instance
\begin{equation}
\text{maximum}(a0, a1) [i, j] = \max(a0[i, j], a1[i, j])
\end{equation}
at all data sample points.

minimum(*a)
returns the minimum value over all arguments at all data sample points and for each component. For instance
\begin{equation}
\text{minimum}(a0, a1) [i, j] = \min(a0[i, j], a1[i, j])
\end{equation}
at all data sample points.

clip(a[, minval=0.], maxval=1.)
cuts back a into the range between minval and maxval. A value in the returned object equals minval if the corresponding value of a is less than minval, equals maxval if the corresponding value of a is greater than maxval or corresponding value of a otherwise.

inner(a0, a1)
returns the inner product of a0 and a1. For instance in the case of rank-2 Data object arguments and this is
\begin{equation}
\text{inner}(a) = \sum_{ij} a0[j, i] \cdot a1[j, i]
\end{equation}
and for a rank-4 Data object arguments this is
\begin{equation}
\text{inner}(a) = \sum_{ijkl} a0[i, j, k, l] \cdot a1[i, j, k, l]
\end{equation}

matrix_mult(a0, a1)
returns the matrix product of a0 and a1. If a1 is rank-1 Data object this is
\begin{equation}
\text{matrix_mult}(a) [i] = \sum_k a0 \cdot [i, k] \cdot a1 [k]
\end{equation}
and if a1 is rank-2 Data object this is
\begin{equation}
\text{matrix_mult}(a) [i, j] = \sum_k a0 \cdot [i, k] \cdot a1 [k, j]
\end{equation}

transposed_matrix_mult(a0, a1)
returns the matrix product of the transposed of a0 and a1. The function is equivalent to \text{matrix_mult}(\text{transpose}(a0), a1). If a1 is rank-1 Data object this is
\begin{equation}
\text{transposed_matrix_mult}(a) [i] = \sum_k a0 \cdot [k, i] \cdot a1 [k]
\end{equation}
and if a1 is rank-2 Data object this is
\begin{equation}
\text{transposed_matrix_mult}(a) [i, j] = \sum_k a0 \cdot [k, i] \cdot a1 [k, j]
\end{equation}

matrix_transposed_mult(a0, a1)
returns the matrix product of a0 and the transposed of a1. The function is equivalent to \text{matrix_mult}(a0, \text{transpose}(a1)). If a1 is rank-2 Data object this is
\begin{equation}
\text{matrix_transposed_mult}(a) [i, j] = \sum_k a0 \cdot [i, k] \cdot a1 [j, k]
\end{equation}

outer(a0, a1)
returns the outer product of a0 and a1. For instance if a0 and a1 both are rank-1 Data object then
\begin{equation}
\text{outer}(a) [i, j] = a0[i] \cdot a1[j]
\end{equation}
and if a0 is rank-1 Data object and a1 is rank-3 Data object
\begin{equation}
\text{outer}(a) [i, j, k] = a0[i] \cdot a1[j, k]
\end{equation}
\end{verbatim}
tensor_mult \((a0, a1)\)

returns the tensor product of \(a0\) and \(a1\). If \(a1\) is rank-2 Data object this is

\[
tensor\_mult(a) [i, j] = \sum_{kl} a0[i, j, k, l] \cdot a1[k, l]
\] (3.26)

and if \(a1\) is rank-4 Data object this is

\[
tensor\_mult(a) [i, j, k, l] = \sum_{mn} a0[i, j, m, n] \cdot a1[m, n, k, l]
\] (3.27)

transposed_tensor_mult \((a0, a1)\)

returns the tensor product of the transposed of \(a0\) and \(a1\). The function is equivalent to \(\text{tensor\_mult(\text{transpose}(a0), a1)}\). If \(a1\) is rank-2 Data object this is

\[
transposed\_tensor\_mult(a) [i, j] = \sum_{kl} a0[k, l, i, j] \cdot a1[k, l]
\] (3.28)

and if \(a1\) is rank-4 Data object this is

\[
transposed\_tensor\_mult(a) [i, j, k, l] = \sum_{mn} a0[m, n, i, j] \cdot a1[m, n, k, l]
\] (3.29)

tensor_transposed_mult \((a0, a1)\)

returns the tensor product of \(a0\) and the transposed of \(a1\). The function is equivalent to \(\text{tensor\_mult(a0, \text{transpose}(a1)})\). If \(a1\) is rank-2 Data object this is

\[
tensor\_transposed\_mult(a) [i, j] = \sum_{kl} a0[i, j, k, l] \cdot a1[l, k]
\] (3.30)

and if \(a1\) is rank-4 Data object this is

\[
tensor\_transposed\_mult(a) [i, j, k, l] = \sum_{mn} a0[i, j, m, n] \cdot a1[k, l, m, n]
\] (3.31)

grad \((a), \text{where=None}\)

returns the gradient of \(a\). If \(\text{where}\) is present the gradient will be calculated in FunctionSpace \(\text{where}\) otherwise a default FunctionSpace is used. In case that \(a\) has rank-2 Data object one has

\[
\text{grad}(a)[i, j, k] = \frac{\partial a[i, j]}{\partial x_k}
\] (3.32)

integrate \((a), \text{where=None}\)

returns the integral of \(a\) where the domain of integration is defined by the FunctionSpace of \(a\). If \(\text{where}\) is present the argument is interpolated into FunctionSpace \(\text{where}\) before integration. For instance in the case of a rank-2 Data object argument in continuous FunctionSpace it is

\[
\text{integrate}(a)[i, j] = \int_{\Omega} a[i, j] \, d\Omega
\] (3.33)

where \(\Omega\) is the spatial domain and \(d\Omega\) volume integration. To integrate over the boundary of the domain one uses

\[
\text{integrate}(a, \text{where=FunctionOnBoundary(a.getDomain)})[i, j] = \int_{\partial\Omega} a[i, j] \, ds
\] (3.34)

where \(\partial\Omega\) is the surface of the spatial domain and \(ds\) area or line integration.

interpolate \((a, \text{where})\)

interpolates argument \(a\) into the FunctionSpace \(\text{where}\).

div \((a), \text{where=None}\)

returns the divergence of \(a\). This

\[
div(a) = \text{trace(\text{grad}(a), \text{where})}
\] (3.35)
The following functions operate “point-wise”. That is, the operation is applied to each component of each point individually.

\[
\begin{align*}
\sin (a) & \quad \text{applies sine function to } a. \\
\cos (a) & \quad \text{applies cosine function to } a. \\
\tan (a) & \quad \text{applies tangent function to } a. \\
\arcsin (a) & \quad \text{applies arc (inverse) sine function to } a. \\
\arccos (a) & \quad \text{applies arc (inverse) cosine function to } a. \\
\arctan (a) & \quad \text{applies arc (inverse) tangent function to } a. \\
\sinh (a) & \quad \text{applies hyperbolic sine function to } a. \\
\cosh (a) & \quad \text{applies hyperbolic cosine function to } a. \\
\tanh (a) & \quad \text{applies hyperbolic tangent function to } a. \\
\text{asinh} (a) & \quad \text{applies arc (inverse) hyperbolic sine function to } a. \\
\text{acosh} (a) & \quad \text{applies arc (inverse) hyperbolic cosine function to } a. \\
\text{atanh} (a) & \quad \text{applies arc (inverse) hyperbolic tangent function to } a. \\
\exp (a) & \quad \text{applies exponential function to } a. \\
\sqrt{a} & \quad \text{applies square root function to } a. \\
\log (a) & \quad \text{applies the natural logarithm to } a. \\
\log_{10} (a) & \quad \text{applies the base-10 logarithm to } a. \\
\text{sign} (a) & \quad \text{applies the sign function to } a, \text{that is } 1 \text{ where } a \text{ is positive, } -1 \text{ where } a \text{ is negative and 0 otherwise.} \\
\text{wherePositive} (a) & \quad \text{returns a function which is } 1 \text{ where } a \text{ is positive and 0 otherwise.} \\
\text{whereNegative} (a) & \quad \text{returns a function which is } 1 \text{ where } a \text{ is negative and 0 otherwise.}
\end{align*}
\]
whereNonNegative \((a)\)
returns a function which is 1 where \(a\) is non-negative and 0 otherwise.

whereNonPositive \((a)\)
returns a function which is 1 where \(a\) is non-positive and 0 otherwise.

whereZero \((a[, \text{tol}=\text{None}, [\text{rtol}=1.e-8]])\)
returns a function which is 1 where \(a\) equals zero with tolerance \(\text{tol}\) and 0 otherwise. If \(\text{tol}\) is not present, the absolute maximum value of \(C_a\) times \(C_{rtol}\) is used.

whereNonZero \((a[, \text{tol}=\text{None}, [\text{rtol}=1.e-8]])\)
returns a function which is 1 where \(a\) different from zero with tolerance \(\text{tol}\) and 0 otherwise. If \(\text{tol}\) is not present, the absolute maximum value of \(C_a\) times \(C_{rtol}\) is used.

### 3.1.7 Operator Class

The Operator class provides an abstract access to operators build within the LinearPDE class. Operator objects are created when a PDE is handed over to a PDE solver library and handled by the LinearPDE object defining the PDE. The user can gain access to the Operator of a LinearPDE object through the getOperator method.

```python
class Operator():
    # creates an empty Operator object.
    def isEmpty(self, fileName):
        # returns True is the object is empty. Otherwise True is returned.
    def setValue(self, value):
        # resets all entries in the object representation to value
    solves(self, rhs):
        # solves the operator equation with right hand side rhs
    of(self, u):
        # applies the operator to the Data object u
    saveMM(self, fileName):
        # saves the object to a matrix market format file of name fileName, see http://maths.nist.gov/MatrixMarket
```

### 3.2 Utilities

**setEscriptParamInt** \((\text{name}, \text{value})\)
assigns the integer value \(\text{value}\) to the parameter \(\text{name}\). If \(\text{name}="\text{TOO_MANY_LINES}"\) conversion of any Data object to a string switches to a condensed format if more than \(\text{value}\) lines would be created.

**getEscriptParamInt** \((\text{name})\)
returns the current value of integer parameter \(\text{name}\).

**listEscriptParams** \((\text{a})\)
returns a list of valid parameters and their description.

**getMPISizeWorld()**
returns the number of MPI processors in use in the MPI_COMM_WORLD processor group. If MPI is not used 1 is returned.

**getMPIRankWorld()**
returns the rank of the process within the MPI_COMM_WORLD processor group. If MPI is not used 0 is returned.

**MPIBarrierWorld()**
performs a barrier synchronization across all processors within MPI_COMM_WORLD processor group.

**getMPIWorldMax** \((\text{a})\)
returns the maximum value of the integer \(\text{a}\) across all processors within MPI_COMM_WORLD.
4.1 Linear Partial Differential Equations

The \texttt{LinearPDE} class is used to define a general linear, steady, second order PDE for an unknown function \( u \) on a given \( \Omega \) defined through a \texttt{Domain} object. In the following \( \Gamma \) denotes the boundary of the domain \( \Omega \). \( n \) denotes the outer normal field on \( \Gamma \).

For a single PDE with a solution with a single component the linear PDE is defined in the following form:

\[
-(A_{jl}u)_j - (B_j u)_j + C_l u + D u = -X_{j,j} + Y .
\]  

(4.1)

\( u, j \) denotes the derivative of \( u \) with respect to the \( j \)-th spatial direction. Einstein’s summation convention, i.e. summation over indexes appearing twice in a term of a sum is performed, is used. The coefficients \( A, B, C, D, X \) and \( Y \) have to be specified through \texttt{Data} objects in the general \texttt{FunctionSpace} on the PDE or objects that can be converted into such \texttt{Data} objects. \( A \) is a rank-2 \texttt{Data} object, \( B, C \) and \( X \) are rank-1 \texttt{Data} object and \( D \) and \( Y \) are scalar. The following natural boundary conditions are considered on \( \Gamma \):

\[
 n_j (A_{jl} u, l + B_j u) + d u = n_j X_{j,j} + y .
\]  

(4.2)

Notice that the coefficients \( A, B \) and \( X \) are defined in the PDE. The coefficients \( d \) and \( y \) are each a scalar \texttt{Data} object in the boundary \texttt{FunctionSpace}. Constraints for the solution prescribing the value of the solution at certain locations in the domain. They have the form

\[
 u = r \text{ where } q > 0
\]  

(4.3)

\( r \) and \( q \) are each scalar \texttt{Data} object where \( q \) is the characteristic function defining where the constraint is applied. The constraints defined by Equation (4.3) override any other condition set by Equation (4.1) or Equation (4.2).

For a system of PDEs and a solution with several components the PDE has the form

\[
-(A_{ijkl}u_{k,l})_j - (B_{ijk} u_k)_j + C_{ikl} u_{k,l} + D_{ik} u_k = -X_{ij,j} + Y_i .
\]  

(4.4)

\( A \) is a rank-4 \texttt{Data} object, \( B \) and \( C \) are each a rank-3 \texttt{Data} object, \( D \) and \( X \) are each a rank-2 \texttt{Data} object and \( Y \) is a rank-1 \texttt{Data} object. The natural boundary conditions take the form:

\[
 n_j (A_{ijkl} u_{k,l} + B_{ijk} u_k) + d_{ik} u_k = n_j X_{ij,j} + y_i .
\]  

(4.5)

The coefficient \( d \) is a rank-2 \texttt{Data} object and \( y \) is a rank-1 \texttt{Data} object both in the boundary \texttt{FunctionSpace}. Constraints take the form

\[
 u_i = r_i \text{ where } q_i > 0
\]  

(4.6)

\( r \) and \( q \) are each rank-1 \texttt{Data} object. Notice that not necessarily all components must have a constraint at all locations.

\texttt{LinearPDE} also supports solution discontinuities over contact region \( \Gamma^{contact} \) in the domain \( \Omega \). To specify the conditions across the discontinuity we are using the generalised flux \( J \) which is in the case of a systems of PDEs and several components of the solution defined as

\[
 J_{ij} = A_{ijkl} u_{k,l} + B_{ijk} u_k - X_{ij}
\]  

(4.7)
For the case of single solution component and single PDE $J$ is defined

$$J_j = A_{jl}u_l + B_j u_k - X_j$$  \hfill (4.8)

In the context of discontinuities $n$ denotes the normal on the discontinuity pointing from side 0 towards side 1. For a system of PDEs the contact condition takes the form

$$n_j J^0_{ij} = n_j J^1_{ij} = y^\text{contact}_i - d^\text{contact}_i [u]$$ \hfill (4.9)

where $J^0$ and $J^1$ are the fluxes on side 0 and side 1 of the discontinuity $\Gamma^\text{contact}$, respectively. $[u]$, which is the difference of the solution at side 1 and at side 0, denotes the jump of $u$ across $\Gamma^\text{contact}$. The coefficient $d^\text{contact}$ is a rank-2 Data object and $y^\text{contact}$ is a rank-1 Data object both in the contact FunctionSpace on side 0 or contact FunctionSpace on side 1. In case of a single PDE and a single component solution the contact condition takes the form

$$n_j J^0_j = n_j J^1_j = y^\text{contact} - d^\text{contact} [u]$$ \hfill (4.10)

In this case the the coefficient $d^\text{contact}$ and $y^\text{contact}$ are each scalar Data object both in the contact FunctionSpace on side 0 or contact FunctionSpace on side 1.

The PDE is symmetrical if

$$A_{jl} = A_{lj} \text{ and } B_j = C_j$$ \hfill (4.11)

The system of PDEs is symmetrical if

$$A_{ijkl} = A_{klij} \text{, } B_{ijk} = C_{kij} \text{, } D_{ik} = D_{ki} \text{, } d_{ik} = d_{ki} \text{, } d^\text{contact}_{ik} = d^\text{contact}_{ki}$$ \hfill (4.12-16)

Note that in contrast with the scalar case Equation (4.11) now the coefficients $D$, $d$ abd $d^\text{contact}$ have to be inspected.

### 4.1.1 Classes

The module `esys.escript.linearPDEs` provides an interface to define and solve linear partial differential equations within `esys.escript`. The module `esys.escript.linearPDEs` does not provide any solver capabilities in itself but hands the PDE over to the PDE solver library defined through the Domain of the PDE. The general interface is provided through the `LinearPDE` class. The `AdvectivePDE` which is derived from the `LinearPDE` class provides an interface to a PDE dominated by its advective terms. The `Poisson` class which is also derived form the `LinearPDE` class should be used to define the Poisson equation.

### 4.1.2 LinearPDE class

This is the general class to define a linear PDE in `esys.escript`. We list a selection of the most important methods of the class. For a complete list, see the reference at `http://shake200.esscc.uq.edu.au/esys/docs.html`.

**class** `LinearPDE` (**domain**,**numEquations**=0,**numSolutions**=0)

opens a linear, steady, second order PDE on the Domain domain. numEquations and numSolutions gives the number of equations and the number of solution components. If numEquations and numSolutions is non-positive, the number of equations and the number solutions, respectively, stay undefined until a coefficient is defined.

**LinearPDE methods**

- **setValue** ([A], [B], [C], [D], [X], [Y], [dContact], [yContact], [q], [r])

  assigns new values to coefficients. By default all values are assumed to be zero. If the new coefficient is not a Data object, it is converted into a Data object in the appropriate FunctionSpace.

\footnote{In fact it is assumed they are not present by assigning the value `escript.Data()`. The can by used by the solver library to reduce computational costs.}
getCoefficient (name)
return the value assigned to coefficient name. If name is not a valid name an exception is raised.

getShapeOfCoefficient (name)
returns the shape of coefficient name even if no value has been assigned to it.

getFunctionSpaceForCoefficient (name)
returns the FunctionSpace of coefficient name even if no value has been assigned to it.

setDebugOn ()
switches on debug mode.

setDebugOff ()
switches off debug mode.

isUsingLumping ()
returns True if LinearPDE.LUMPING is set as the solver for the system of linear equations. Otherwise False is returned.

setSolverMethod ([ solver=LinearPDE.DEFAULT ], preconditioner=LinearPDE.DEFAULT )
sets the solver method and preconditioner to be used. It should be noted that a PDE solver library may not know the specified solver method but may choose a similar method and preconditioner.

getSolverMethodName ()
returns the name of the solver method and preconditioner which is in use.

getSolverMethod ()
returns the solver method and preconditioner which is in use.

setSolverPackage ([ package=LinearPDE.DEFAULT ])
sets the solver package to be used by PDE library to solve the linear systems of equations. The specified package may not be supported by the PDE solver library. In this case, depending on the PDE solver, the default solver is used or an exception is thrown. If package is not specified, the default package of the PDE solver library is used.

getSolverPackage ()
returns the linear solver package currently by the PDE solver library

setTolerance ([ tol=1.e-8 ])
resets the tolerance for solution. The actually meaning of tolerance depends on the underlying PDE library. In most cases, the tolerance will only consider the error from solving the discrete problem but will not consider any discretization error.

setToleranceReductionFactor (TOL)
lowers the tolerance by a factor of TOL.

getTolerance ()
returns the current tolerance of the solution

getDomain ()
returns the Domain of the PDE.

getDim ()
returns the spatial dimension of the PDE.

getNumEquations ()
returns the number of equations.

getNumSolutions ()
returns the number of components of the solution.

checkSymmetry (verbose=False)
returns True if the PDE is symmetric and False otherwise. The method is very computationally expensive and should only be called for testing purposes. The symmetry flag is not altered. If verbose=True information about where symmetry is violated are printed.

getFlux (u)
returns the flux $J_{ij}$ for given solution $u$ defined by Equation (4.7) and Equation (4.8), respectively.

4.1. Linear Partial Differential Equations
isSymmetric() returns True if the PDE has been indicated to be symmetric. Otherwise False is returned.

setSymmetryOn() indicates that the PDE is symmetric.

setSymmetryOff() indicates that the PDE is not symmetric.

setReducedOrderOn() switches on the reduction of polynomial order for the solution and equation evaluation even if a quadratic or higher interpolation order is defined in the Domain. This feature may not be supported by all PDE libraries.

setReducedOrderOff() switches off the reduction of polynomial order for the solution and equation evaluation.

getOperator() returns the Operator of the PDE.

getRightHandSide() returns the right hand side of the PDE as a Data object. If ignoreConstraint=True, then the constraints are not considered when building up the right hand side.

getSystem() returns the Operator and right hand side of the PDE.

getSolution( verbose=False, reordering=LinearPDE.NO_REORDERING, iter_max=1000, drop_tolerance=0.01, drop_storage=1.2, truncation=-1, restart=-1) returns (an approximation of) the solution of the PDE. If verbose=True, then some information is printed during the solution process. reordering selects a reordering methods that is applied before or during the solution process (=LinearPDE.NO_REORDERING, LinearPDE.MINIMUM_FILL_IN, LinearPDE.NESTED_DISSECTION). iter_max specifies the maximum number of iteration steps that are allowed to reach the specified tolerance. drop_tolerance specifies a relative tolerance for small elements to be dropped when building a preconditioner (eg. in LinearPDE.ILUT). drop_storage limits the extra storage allowed when building a preconditioner (eg. in LinearPDE.ILUT). The extra storage is given relative to the size of the stiffness matrix, eg. drop_storage=1.2 will allow the preconditioner to use the 1.2 fold storage space than used for the stiffness matrix. truncation defines the truncation.

LinearPDE symbols/members

DEFAULT default method, preconditioner or package to be used to solve the PDE. An appropriate method should be chosen by the used PDE solver library.

SCSL the SCSL library by SGI, Reference [2]

MKL the MKL library by Intel, Reference [11].

UMFPACK the UMFPACK, Reference [22]. Remark: UMFPACK is not parallelized.

PASO the solver library of esys.finley, see Section ??.

ITERATIVE the default iterative method and preconditioner. The actually used method depends on the PDE solver library and the solver package been chosen. Typically, LinearPDE.PCG is used for symmetric PDEs and LinearPDE.BICGSTAB otherwise, both with LinearPDE.JACOBI preconditioner.

DIRECT the default direct linear solver.

2The SCSL library will only be available on SGI systems
3The MKL library will only be available when the Intel compilation environment is used.
CHOLEVSKY

direct solver based on Cholevsky factorization (or similar), see Reference [18]. The solver will require a symmetric PDE.

PCG

preconditioned conjugate gradient method, see Reference [23]. The solver will require a symmetric PDE.

TFQMR

transpose-free quasi-minimal residual method, see Reference [23].

GMRES

the GMRES method, see Reference [23]. Truncation and restart are controlled by the parameters truncation and restart of getSolution.

MINRES

minimal residual method method,

LUMPING

uses lumping to solve the system of linear equations. This solver technique condenses the stiffness matrix to a diagonal matrix so the solution of the linear systems becomes very cheap. It can be used when only $D$ is present but in any case has to applied with care. The difference in the solutions with and without lumping can be significant but is expected to converge to zero when the mesh gets finer. Lumping does not use the linear system solver library.

PRES20

the GMRES method with truncation after five residuals and restart after 20 steps, see Reference [23].

CGS

conjugate gradient squared method, see Reference [23].

BICGSTAB

stabilized bi-conjugate gradients methods, see Reference [23].

SSOR

symmetric successive over-relaxation method, see Reference [23]. Typically used as preconditioner but some linear solver libraries support this as a solver.

ILU0

the incomplete LU factorization preconditioner with no fill-in, see Reference [18].

ILUT

the incomplete LU factorization preconditioner with fill-in, see Reference [18]. During the LU-factorization element with relative size less then drop_tolerance are dropped. Moreover, the size of the LU-factorization is restricted to the drop_storage-fold of the stiffness matrix. drop_tolerance and drop_storage are both set in the getSolution call.

JACOBI

the Jacobi preconditioner, see Reference [18].

AMG

the algebraic–multi grid method, see Reference [19]. This method can be used as linear solver method but is more robust when used in a preconditioner.

GS

the symmetric Gauss-Seidel preconditioner, see Reference [18].

RILU

recursive incomplete LU factorization preconditioner, see Reference [24]. This method is similar to LinearPDE.ILUT but uses smoothing between levels. During the LU-factorization element with relative size less then drop_tolerance are dropped. Moreover, the size of the LU-factorization is restricted to the drop_storage-fold of the stiffness matrix. drop_tolerance and drop_storage are both set in the getSolution call.

NO_REORDERING

no ordering is used during factorization.

MINIMUM_FILL_IN
applies reordering before factorization using a fill-in minimization strategy. You have to check with the particular solver library or linear solver package if this is supported. In any case, it is advisable to apply reordering on the mesh to minimize fill-in.

**NESTED_DISSECTION**

applies reordering before factorization using a nested dissection strategy. You have to check with the particular solver library or linear solver package if this is supported. In any case, it is advisable to apply reordering on the mesh to minimize fill-in.

### 4.1.3 The Poisson Class

The **Poisson** class provides an easy way to define and solve the Poisson equation

\[-u_{,ii} = f. \quad (4.17)\]

with homogeneous boundary conditions

\[n_i u_{,i} = 0 \quad (4.18)\]

and homogeneous constraints

\[u = 0 \text{ where } q > 0 \quad (4.19)\]

\(f\) has to be a scalar Data object in the general FunctionSpace and \(q\) must be a scalar Data object in the solution FunctionSpace.

**class Poisson (domain)**

opens a Poisson equation on the Domain domain. Poisson is derived from LinearPDE.

**setValue** (f=escript.Data(), q=escript.Data())

assigns new values to \(f\) and \(q\).

### 4.1.4 The Helmholtz Class

The **Helmholtz** class defines the Helmholtz problem

\[\omega u - (k u_{,j})_j = f \quad (4.20)\]

with natural boundary conditions

\[k u_{,j} n_j = g - \alpha u \quad (4.21)\]

and constraints:

\[u = r \text{ where } q > 0 \quad (4.22)\]

\(\omega, k, f\) have to be a scalar Data object in the general FunctionSpace, \(g\) and \(\alpha\) must be a scalar Data object in the boundary FunctionSpace, and \(q\) and \(r\) must be a scalar Data object in the solution FunctionSpace or must be mapped or interpolated into the particular FunctionSpace.

**class Helmholtz (domain)**

opens a Helmholtz equation on the Domain domain. Helmholtz is derived from LinearPDE.

**setValue** ([omega] [k] [f] [alpha] [g] [r] [q])

assigns new values to \(omega, k, f, alpha, g, r, q\). By default all values are set to be zero.

### 4.1.5 The Lame Class

The **Lame** class defines a Lame equation problem:

\[-\mu(u_{,i,j} + u_{,j,i}) + \lambda u_{k,k})_j = F_i - \sigma_{ij,j} \quad (4.23)\]

with natural boundary conditions:

\[n_j (\mu (u_{,i,j} + u_{,j,i}) + \lambda * u_{k,k}) = f_i + n_j \sigma_{ij} \quad (4.24)\]
and constraint

\[ u_i = r_i \text{ where } q_i > 0 \] (4.25)

\( \mu, \lambda \) have to be a scalar Data object in the general FunctionSpace, \( F \) has to be a vector Data object in the general FunctionSpace, \( \sigma \) has to be a tensor Data object in the general FunctionSpace, \( f \) must be a vector Data object in the boundary FunctionSpace, and \( q \) and \( r \) must be a vector Data object in the solution FunctionSpace or must be mapped or interpolated into the particular FunctionSpace.

class Lame(domain)

opens a Lame equation on the Domain domain. Lame is derived from LinearPDE.

setValue( [lame_lambda], lame_mu, F, sigma, f, r, q )

assigns new values to \( \text{lame}_\lambda, \text{lame}_\mu, F, \sigma, f, r \) and \( q \). By default all values are set to be zero.
CHAPTER
FIVE

The Module \texttt{esys.pycad}

5.1 Introduction

\texttt{esys.pycad} provides a simple way to build a mesh for your finite element simulation. You begin by building what we call a \textit{Design} using primitive geometric objects, and then to go on to build a mesh from the \textit{Design}. The final step of generating the mesh from a \textit{Design} uses freely available mesh generation software, such as \textit{Gmsh}[9].

A \textit{Design} is built by defining points, which are used to specify the corners of geometric objects and the vertices of curves. Using points you construct more interesting objects such as lines, rectangles, and arcs. By adding many of these objects into what we call a \textit{Design}, you can build meshes for arbitrarily complex 2-D and 3-D structures.

The example included below shows how to use \textit{pycad} to create a 2-D mesh in the shape of a trapezoid with a cutout area.

```python
from esys.pycad import *
from esys.pycad.gmsh import Design
from esys.finley import MakeDomain

# A trapezoid
p0=Point(0.0, 0.0, 0.0)
p1=Point(1.0, 0.0, 0.0)
p2=Point(1.0, 0.5, 0.0)
p3=Point(0.0, 1.0, 0.0)
101=Line(p0, p1)
112=Line(p1, p2)
123=Line(p2, p3)
130=Line(p3, p0)
c=CurveLoop(101, 112, 123, 130)

# A small triangular cutout
x0=Point(0.1, 0.1, 0.0)
x1=Point(0.5, 0.1, 0.0)
x2=Point(0.5, 0.2, 0.0)
x01=Line(x0, x1)
x12=Line(x1, x2)
x20=Line(x2, x0)
cutout=CurveLoop(x01, x12, x20)

# Create the surface with cutout
s=PlaneSurface(c, holes=[cutout])

# Create a Design which can make the mesh
d=Design(dim=2, element_size=0.05)

# Add the trapezoid with cutout
d.addItems(s)

# Create the geometry, mesh and Escript domain
d.setScriptFileName("trapezoid.geo")
```
d.setMeshFileName("trapezoid.msh")
domain=MakeDomain(d, integrationOrder=-1, reducedIntegrationOrder=-1, optimizeLabeling=True)

# Create a file that can be read back into python with mesh=ReadMesh(fileName)
domain.write("trapezoid.fly")

This example is included with the software in pycad/examples/trapezoid.py. If you have gmsh installed you can run the example and view the geometry and mesh with:

```
python trapezoid.py
gmsh trapezoid.geo
gmsh trapezoid.msh
```

A CurveLoop is used to connect several lines into a single curve. It is used in the example above to create the trapezoidal outline for the grid and also for the triangular cutout area. You can use any number of lines when creating a CurveLoop, but the end of one line must be identical to the start of the next.

Sometimes you might see us write -c where c is a CurveLoop. This is the reverse curve of the curve c. It is identical to the original except that its points are traversed in the opposite order. This may make it easier to connect two curves in a CurveLoop.

The example python script above calls both d.setScriptFileName() and d.setMeshFileName(). You need only call these if you wish to save the gmsh geometry and mesh files.

Note that the underlying mesh generation software will not accept all the geometries you can create with pycad. For example, pycad will happily allow you to create a 2-D Design that is a closed loop with some additional points or lines lying outside of the enclosed area, but gmsh will fail to create a mesh for it.

5.2 esys.pycad Classes

5.2.1 Primitives

Some of the most commonly-used objects in pycad are listed here. For a more complete list see the full API documentation.

**class Point (x1, x2, x3)**  
Create a point with from coordinates.

**class Line (point1, point2)**  
Create a line with between starting and ending points.

**class Curve (point1, point2, ...)**  
Create a Curve, which is simply a list of points.

**class Spline (curve)**  
Interpret a Curve using a spline.

**class BSpline (curve)**  
Interpret a Curve using a b-spline.

**class BezierCurve (curve)**  
Interpret a Curve using a Bezier curve.

**class CurveLoop (list)**  
Create a closed Curve connecting the lines and/or points given in the list.

**class Arc (center_point, start_point, end_point)**  
Create an arc by specifying a center for a circle and start and end points. An arc may subtend an angle of at most \( \pi \) radians.

**class PlaneSurface (loop, [holes=list])**  
Create a surface for a 2-D mesh, which may have one or more holes.

**class RuledSurface (list)**  
Create a surface that can be interpolated using transfinite interpolation.
class SurfaceLoop(list)
Create a loop of 2D primitives, which defines the shell of a volume.

class Volume(loop, [holes=list])
Create a volume for a 3-D mesh given a SurfaceLoop, which may have one or more holes.

class PropertySet(list)
Create a PropertySet given a list of 1-D, 2-D or 3-D items. See the section on Properties below for more information.

5.2.2 Transformations

Sometimes it’s convenient to create an object and then make copies at different orientations and in different sizes. Transformations are used to move geometrical objects in the 3-dimensional space and to resize them.

class Translation([b=[0,0,0]])
defines a translation $x \rightarrow x + b$. $b$ can be any object that can be converted into a numarray object of shape (3,).

class Rotation([axis=[1,1,1], point=[0,0,0], angle=0*RAD])
defines a rotation by angle around axis through point point and direction axis. axis and point can be any object that can be converted into a numarray object of shape (3,). axis does not have to be normalized but must have positive length. The right hand rule [1] applies.

class Dilation([factor=1., center=[0,0,0]])
defines a dilation by the expansion/contraction factor with center as the dilation center. center can be any object that can be converted into a numarray object of shape (3,).

class Reflection([normal=[1,1,1], offset=0])
defines a reflection on a plane defined in normal form $n^T x = d$ where $n$ is the surface normal normal and $d$ is the plane offset. normal can be any object that can be converted into a numarray object of shape (3,). normal does not have to be normalized but must have positive length.

DEG
A constant to convert from degrees to an internal angle representation in radians. For instance use 90*DEG for 90 degrees.

5.2.3 Properties

If you are building a larger geometry you may find it convenient to create it in smaller pieces and then assemble them into the whole. Property sets make this easy, and they allow you to name the smaller pieces for convenience. Property sets are used to bundle a set of geometrical objects in a group. The group is identified by a name. Typically a property set is used to mark subregions with share the same material properties or to mark portions of the boundary. For efficiency, the Design class object assigns a integer to each of its property sets, a so-called tag. The appropriate tag is attached to the elements at generation time.

See the file pycad/examples/quad.py for an example using a PropertySet.

class PropertySet(name, *items)
defines a group geometrical objects which can be accessed through a name The objects in the tuple items must all be Manifold1D, Manifold2D or Manifold3D objects.

getManifoldClass()returns the manifold class Manifold1D ,Manifold2D or Manifold3D expected from the items in the property set.

getDim()returns the spatial dimension of the items in the property set.

getName()returns the name of the set

setName(name)
sets the name. This name should be unique within a Design.

addItem(*items)
adds a tuple of items. They need to be objects of class Manifold1D, Manifold2D or Manifold3D.

getItems()
returns the list of items

clearItems()
clears the list of items

getTag()
returns the tag used for this property set

5.3 Interface to the mesh generation software

The class and methods described here provide an interface to the mesh generation software, which is currently gmsh. This interface could be adopted to triangle or another mesh generation package if this is deemed to be desirable in the future.

class Design(
    [dim=3,
     [element size=1.,
      [order=1, [keep_files=False]]]])

The Design describes the geometry defined by primitives to be meshed. The dim specifies the spatial dimension. The argument element size defines the global element size which is multiplied by the local scale to set the element size at each Point. The argument order defines the element order to be used. If keep_files is set to True temporary files are kept otherwise they are removed when the instance of the class is deleted.

setDim([dim=3])
sets the spatial dimension which needs to be 1, 2 or 3.

getDim()
returns the spatial dimension.

setElementOrder([order=1])
sets the element order which needs to be 1 or 2.

getElementOrder()
returns the element order.

setElementSize([element size=1])
sets the global element size. The local element size at a point is defined as the global element size multiplied by the local scale. The element size must be positive.

getElementSize()
returns the global element size.

DELAUNAY
the gmsh Delaunay triangulator.

TETGEN

NETGEN
the NETGEN [8] triangulator.

setKeepFilesOn()
work files are kept at the end of the generation.

setKeepFilesOff()
work files are deleted at the end of the generation.

keepFiles()
returns True if work files are kept. Otherwise False is returned.

setScriptFileName([name=None])
set the filename for the gmsh input script. If no name is given a name with extension “geo” is generated.
getScriptFileName()  
returns the name of the file for the gmsh script.

setMeshFileName([name=None])  
sets the name for the gmsh mesh file. If no name is given a name with extension "msh" is generated.

getMeshFileName()  
returns the name of the file for the gmsh msh

addItems(*items)  
adds the tuple of varitems. An item can be any primitive or a PropertySet. **Warning:** If a PropertySet is added as an item added object that are not part of a PropertySet are not considered in the meshing.

getItems()  
returns a list of the items

clearItems()  
resets the items in design

getMeshHandler()  
returns a handle to the mesh. The call of this method generates the mesh from the geometry and returns a mechanism to access the mesh data. In the current implementation this method returns a file name for a gmsh file containing the mesh data.

getScriptString()  
returns the gmsh script to generate the mesh as a string.

getCommandString()  
returns the gmsh command used to generate the mesh as string.

setOptions([algorithm=None, [optimize_quality=True, [smoothing=1]]]])  
sets options for the mesh generator. algorithm sets the algorithm to be used. The algorithm needs to be Design.DELAUNAY Design.TETGEN or Design.NETGEN. By default Design.DELAUNAY is used. optimize_quality=True invokes an optimization of the mesh quality. smoothing sets the number of smoothing steps to be applied to the mesh.

getTagMap()  
returns a TagMap to map the name PropertySet in the class to tag numbers generated by gmsh.
The Module \texttt{esys.pyvisi}

Warning: The Module \texttt{esys.pyvisi} is not supported under MPI.

6.1 Introduction

\texttt{esys.pyvisi} is a Python module that is used to generate 2D and 3D visualizations for \texttt{escript} and its PDE solver \texttt{finley}. The module provides an easy to use interface to the \textsc{VTK} library (http://www.vtk.org/) to render (generate) surface maps and contours for scalar fields, arrows and streamlines for vector fields, and ellipsoids for tensor fields. There are three approaches for rendering an object. (1) Online - object is rendered on-screen with interaction capability (i.e. zoom and rotate), (2) Offline - object is rendered off-screen (no pop-up window) and (3) Display - object is rendered on-screen but with no interaction capability (on-the-fly animation). All three approaches have the option to save the rendered object as an image (e.g. jpeg) and subsequently converting a series of images into a movie (mpeg).

The following outlines the general steps to use Pyvisi:

1. Create a \texttt{Scene} instance - a window in which objects will be rendered on.
2. Create a data input instance (i.e. \texttt{DataCollector} or \texttt{ImageReader}) - reads the source data for visualization.
3. Create a data visualization object (i.e. \texttt{Map}, \texttt{Velocity}, \texttt{Ellipsoid}, \texttt{Contour}, \texttt{Carpet}, \texttt{StreamLine}, etc.) - creates a visual representation of the source data.
4. Create a \texttt{Camera} or \texttt{Light} instance - controls the viewing angle and lighting effects.
5. Render the object - using either the Online, Offline or Display approach.
6. Generate movie - converts a series of images into a movie. (optional)

\hspace{1cm} scene $\rightarrow$ data input $\rightarrow$ data visualization $\rightarrow$ camera/light $\rightarrow$ render $\rightarrow$ movie

6.2 \texttt{esys.pyvisi} Classes

The following subsections give a brief overview of the important classes and some of their corresponding methods. Please refer to http://shake200.esscc.uq.edu.au/esys/docs.html for full details.

6.2.1 Scene Classes

This section details the instances used to setup the viewing environment.
Scene class

class Scene (renderer = Renderer.ONLINE, num_viewport = 1, x_size = 1152, y_size = 864)
A scene is a window in which objects are to be rendered on. Only one scene needs to be created. However, a scene may be divided into four smaller windows called viewports (if needed). Each viewport in turn can render a different object.

The following are some of the methods available:
setBackgroundColor (color)
Set the background color of the scene.

class Scene (image_name = None)
Render the object using either the Online, Offline or Display mode.

Camera class

class Camera (scene, viewport = Viewport.SOUTH_WEST)
A camera controls the display angle of the rendered object and one is usually created for a Scene. However, if a Scene has four viewports, then a separate camera may be created for each viewport.

The following are some of the methods available:
setFocalPoint (position)
Set the focal point of the camera.
 setPosition (position)
Set the position of the camera.
 azimuth (angle)
Rotate the camera to the left and right. The angle parameter is in degrees.
 elevation (angle)
Rotate the camera up and down (angle must be between -90 and 90).
 backView ()
Rotate the camera to view the back of the rendered object.
 topView ()
Rotate the camera to view the top of the rendered object.
 bottomView ()
Rotate the camera to view the bottom of the rendered object.
 leftView ()
Rotate the camera to view the left side of the rendered object.
 rightView ()
Rotate the camera to view the right side of the rendered object.
 isometricView ()
Rotate the camera to view an isometric projection of the rendered object.
 dolly (distance)
Move the camera towards (greater than 1) the rendered object. However, it is not possible to move the camera away from the rendered object with this method.

Light class

class Light (scene, viewport = Viewport.SOUTH_WEST)
A light controls the lighting effect for the rendered object and is set up in a similar way to Camera.

The following are some of the methods available:
setColor (color)
Set the light color.
setFocalPoint (position)
    Set the focal point of the light.
setPosition (position)
    Set the position of the light.
setAngle (elevation = 0, azimuth = 0)
    An alternative to set the position and focal point of the light by using elevation and azimuth.

6.2.2  Input Classes

This subsection details the instances used to read and load the source data for visualization.

DataCollector class

class DataCollector (source = Source.XML)
    A data collector is used to read data either from an XML file (using setFileName()) or from an escript object directly (using setData()). Writing XML files is expensive but has the advantage that the results can be analyzed easily after the simulation has completed.

The following are some of the methods available:
setFileName (file_name)
    Set the XML file name to read.
setData (**args)
    Create data using the <name>=<data> pairing. The method assumes that the data is given in the appropriate format.
setActiveScalar (scalar)
    Specify the scalar field to load.
setActiveVector (vector)
    Specify the vector field to load.
setActiveTensor (tensor)
    Specify the tensor field to load.

ImageReader class

class ImageReader (format)
    An image reader is used to read data from an image in a variety of formats.

The following is one of the methods available:
setImageName (image_name)
    Set the filename of the image to be loaded.

Text2D class

class Text2D (scene, text, viewport = Viewport.SOUTH_WEST)
    This class is used to insert two-dimensional text for annotations (e.g. titles, authors and labels).

The following are some of the methods available:
setFontSize (size)
    Set the 2D text size.
boldOn ()
    Use bold font style for the text.
setColor (color)
    Set the color of the 2D text.

Including methods from Actor2D.
6.2.3 Data Visualization Classes

This subsection details the instances used to process and manipulate the source data. The typical usage of some of the classes is also shown. See Section 6.5 for sample images generated with these classes.

One point to note is that the source can either be point or cell data. If the source is cell data, a conversion to point data may or may not be required, in order for the object to be rendered correctly. If a conversion is needed, the 'cell_to_point' flag (see below) must be set to 'True', otherwise to 'False' (which is the default). On occasions, an inaccurate object may be rendered from cell data even after conversion.

Map class

```python
from esys.pyvisi import Scene, DataCollector, Map, Camera
import os

PYVISI_EXAMPLE_MESHES_PATH = "data_meshes"
PYVISI_EXAMPLE_IMAGES_PATH = "data_sample_images"
X_SIZE = 800
Y_SIZE = 800

SCALAR_FIELD_POINT_DATA = "temperature"
SCALAR_FIELD_CELL_DATA = "temperature_cell"
FILE_3D = "interior_3D.xml"
IMAGE_NAME = "map.jpg"
JPG_RENDERER = Renderer.ONLINE_JPG

# Create a Scene with four viewports.
s = Scene(renderer = JPG_RENDERER, num_viewport = 4, x_size = X_SIZE,
y_size = Y_SIZE)

# Create a DataCollector reading from a XML file.
dcl = DataCollector(source = Source.XML)
dcl.setFileName(file_name = os.path.join(PYVISI_EXAMPLE_MESHES_PATH, FILE_3D))

dcl.setActiveScalar(scalar = SCALAR_FIELD_POINT_DATA)

# Create a Map for the first viewport.
m1 = Map(scene = s, data_collector = dcl, viewport = Viewport.SOUTH_WEST,
lut = Lut.COLOR, cell_to_point = False, outline = True)
m1.setRepresentationToWireframe()

# Create a Camera for the first viewport
cl1 = Camera(scene = s, viewport = Viewport.SOUTH_WEST)
c1.isometricView()
```

```python
# Create a second DataCollector reading from the same XML file but specifying
# a different scalar field.
dc2 = DataCollector(source = Source.XML)
dc2.setFileName(file_name = os.path.join(PYVISI_EXAMPLE_MESHES_PATH, FILE_3D))
```
dc2.setActiveScalar(scalar = SCALAR_FIELD_CELL_DATA)

# Create a Map for the third viewport.
m2 = Map(scene = s, data_collector = dc2, viewport = Viewport.NORTH_EAST, 
        lut = Lut.COLOR, cell_to_point = True, outline = True)

# Create a Camera for the third viewport

c2 = Camera(scene = s, viewport = Viewport.NORTH_EAST)

# Render the object.
s.render(image_name = os.path.join(PYVISI_EXAMPLE_IMAGES_PATH, IMAGE_NAME))

MapOnPlaneCut class

class MapOnPlaneCut (scene, data_collector, viewport = Viewport.SOUTH_WEST, lut = Lut.COLOR, 
        cell_to_point = False, outline = True)

This class works in a similar way to Map, except that the result is a slice of the scalar field produced by 
cutting the map with a plane. The plane can be translated and rotated to its desired position.

The following are some of the methods available:
Methods from Actor3D, Transform and DataSetMapper.

MapOnPlaneClip class

class MapOnPlaneClip (scene, data_collector, viewport = Viewport.SOUTH_WEST, lut = Lut.COLOR, 
        cell_to_point = False, outline = True)

This class works in a similar way to MapOnPlaneCut, except that the defined plane is used to clip the 
scalar field.

The following are some of the methods available:
Methods from Actor3D, Transform, Clipper and DataSetMapper.

MapOnScalarClip class

class MapOnScalarClip (scene, data_collector, viewport = Viewport.SOUTH_WEST, lut = Lut.COLOR, 
        cell_to_point = False, outline = True)

This class works in a similar way to Map, except that it only shows parts of the scalar field matching a scalar 
value.

The following are some of the methods available:
Methods from Actor3D, Clipper and DataSetMapper.

MapOnScalarClipWithRotation class

class MapOnScalarClipWithRotation (scene, data_collector, viewport = Viewport.SOUTH_WEST, lut = 
        Lut.COLOR, cell_to_point = False)

This class works in a similar way to Map except that it shows a 2D scalar field clipped using a scalar value 
and subsequently rotated around the z-axis to create a 3D looking effect. This class should only be used 
with 2D data sets and NOT 3D.

The following are some of the methods available:
Methods from Actor3D, Clipper, Rotation and DataSetMapper.

Velocity class

class Velocity (scene, data_collector, arrow = Arrow.TWO_D, color_mode = ColorMode.VECTOR, viewport 
        = Viewport.SOUTH_WEST, lut = Lut.COLOR, cell_to_point = False, outline = True)

This class is used to display a vector field using arrows. The arrows can either be color or gray-scale, 
depending on the lookup table used. If the arrows are colored, there are two possible coloring modes, either
using vector data or scalar data. Similarly, there are two possible types of arrows, either two-dimensional or three-dimensional.

The following are some of the methods available:
Methods from Actor3D, Glyph3D, MaskPoints and DataSetMapper.

VelocityOnPlaneCut class

class VelocityOnPlaneCut (scene, data_collection, arrow = Arrow.TWO_D, color_mode = ColorMode.VECTOR, viewport = Viewport.SOUTH_WEST, lut = Lut.COLOR, cell_to_point = False, outline = True)

This class works in a similar way to MapOnPlaneCut, except that it shows a vector field using arrows cut using a plane.

The following are some of the methods available:
Methods from Actor3D, Glyph3D, Transform, MaskPoints and DataSetMapper.

A typical usage of VelocityOnPlaneCut is shown below.

```python
# Import the necessary modules
from esys.pyvisi import Scene, DataCollector, VelocityOnPlaneCut, Camera
from esys.pyvisi.constant import *
import os

PYVISI_EXAMPLE_MESHES_PATH = "data_meshes"
PYVISI_EXAMPLE_IMAGES_PATH = "data_sample_images"
X_SIZE = 400
Y_SIZE = 400

VECTOR_FIELD_CELL_DATA = "velocity"
FILE_3D = "interior_3D.xml"
IMAGE_NAME = "velocity.jpg"
JPG_RENDERER = Renderer.ONLINE_JPG

# Create a Scene.
s = Scene(renderer = JPG_RENDERER, num_viewport = 1, x_size = X_SIZE, y_size = Y_SIZE)

# Create a DataCollector reading from a XML file.
dc1 = DataCollector(source = Source.XML)
dc1.setFileName(file_name = os.path.join(PYVISI_EXAMPLE_MESHES_PATH, FILE_3D))
dc1.setActiveVector(vector = VECTOR_FIELD_CELL_DATA)

# Create VelocityOnPlaneCut.
vopc1 = VelocityOnPlaneCut(scene = s, data_collector = dc1, viewport = Viewport.SOUTH_WEST, color_mode = ColorMode.VECTOR, arrow = Arrow.THREE_D, lut = Lut.COLOR, cell_to_point = False, outline = True)
vopc1.setScaleFactor(scale_factor = 0.5)
vopc1.setPlaneToXY(offset = 0.5)
vopc1.setRatio(2)
vopc1.randomOn()

# Create a Camera.
c1 = Camera(scene = s, viewport = Viewport.SOUTH_WEST)
c1.isometricView()
c1.elevation(angle = -20)

# Render the object.
s.render(image_name = os.path.join(PYVISI_EXAMPLE_IMAGES_PATH, IMAGE_NAME))
```
VelocityOnPlaneClip class

```python
class VelocityOnPlaneClip(scene, data_collector, arrow = Arrow.TWO_D, color_mode = ColorMode.VECTOR, viewport = Viewport.SOUTH_WEST, lut = Lat.COLOR, cell_to_point = False, online = True)
```

This class works in a similar way to MapOnPlaneClip, except that it shows a vector field using arrows clipped using a plane.

The following are some of the methods available:
Methods from Actor3D, Glyph3D, Transform, Clipper, MaskPoints and DataSetMapper.

Ellipsoid class

```python
class Ellipsoid(scene, data_collector, viewport = Viewport.SOUTH_WEST, lut = Lat.COLOR, cell_to_point = False, outline = True)
```

Class that shows a tensor field using ellipsoids. The ellipsoids can either be color or gray-scale, depending on the lookup table used.

The following are some of the methods available:
Methods from Actor3D, Sphere, TensorGlyph, MaskPoints and DataSetMapper.

EllipsoidOnPlaneCut class

```python
class EllipsoidOnPlaneCut(scene, data_collector, viewport = Viewport.SOUTH_WEST, lut = Lat.COLOR, cell_to_point = False, outline = True)
```

This class works in a similar way to MapOnPlaneCut, except that it shows a tensor field using ellipsoids cut using a plane.

The following are some of the methods available:
Methods from Actor3D, Sphere, TensorGlyph, Transform, Clipper, MaskPoints and DataSetMapper.

EllipsoidOnPlaneClip class

```python
class EllipsoidOnPlaneClip(scene, data_collector, viewport = Viewport.SOUTH_WEST, lut = Lat.COLOR, cell_to_point = False, outline = True)
```

This class works in a similar way to MapOnPlaneClip, except that it shows a tensor field using ellipsoids clipped using a plane.

The following are some of the methods available:
Methods from Actor3D, Sphere, TensorGlyph, Transform, Clipper, MaskPoints and DataSetMapper.

A typical usage of EllipsoidOnPlaneClip is shown below.

```python
Author: John Ngui, john.ngui@uq.edu.au
```

```python
# Import the necessary modules
from esys.pyvisi import Scene, DataCollector, EllipsoidOnPlaneClip, Camera
from esys.pyvisi.constant import *
import os

PYVISI_EXAMPLE_MESHES_PATH = "data_meshes"
PYVISI_EXAMPLE_IMAGES_PATH = "data_sample_images"
X_SIZE = 400
Y_SIZE = 400

TENSOR_FIELD_CELL_DATA = "stress_cell"
FILE_3D = "interior_3D.xml"
IMAGE_NAME = "ellipsoid.jpg"
JPG_RENDERER = Renderer.ONLINE_JPG
```
# Create a Scene.
s = Scene(renderer = JPG_RENDERER, num_viewport = 1, x_size = X_SIZE, y_size = Y_SIZE)

# Create a DataCollector reading from a XML file.
dc1 = DataCollector(source = Source.XML)
dc1.setFileName(file_name = os.path.join(PYVISI_EXAMPLE_MESHES_PATH, FILE_3D))
dc1 setActiveTensor(tensor = TENSOR_FIELD_CELL_DATA)

# Create an EllipsoidOnPlaneClip.
eopc1 = EllipsoidOnPlaneClip(scene = s, data_collector = dc1, viewport = Viewport.SOUTH_WEST, lut = Lut.COLOR, cell_to_point = True, outline = True)
eopc1.setPlaneToXY()
eopc1.setScaleFactor(scale_factor = 0.2)
eopc1.rotateX(angle = 10)

# Create a Camera.
c1 = Camera(scene = s, viewport = Viewport.SOUTH_WEST)
c1.bottomView()
c1.azimuth(angle = -90)
c1.elevation(angle = 10)

# Render the object.
s.render(image_name = os.path.join(PYVISI_EXAMPLE_IMAGES_PATH, IMAGE_NAME))

```python
class Contour (scene, data_collector, viewport = Viewport.SOUTH_WEST, lut = Lut.COLOR, cell_to_point = False, outline = True)
    Class that shows a scalar field using contour surfaces. The contour surfaces can either be color or gray-scale, depending on the lookup table used. This class can also be used to generate isosurfaces.

The following are some of the methods available:
Methods from Actor3D, ContourModule and DataSetMapper.

A typical usage of Contour is shown below.

```
```
dcl = DataCollector(source = Source.XML)
dcl.setFileName(file_name = os.path.join(PYVISI_EXAMPLE_MESHES_PATH, FILE_3D))
dcl.setActiveScalar(scalar = SCALAR_FIELD_POINT_DATA)

# Create three contours.
ctr1 = Contour(scene = s, data_collector = dcl, viewport = Viewport.SOUTH_WEST,
               lut = Lut.COLOR, cell_to_point = False, outline = True)
ctr1.generateContours(contours = 3)

# Create a Camera.
cam1 = Camera(scene = s, viewport = Viewport.SOUTH_WEST)
cam1.elevation(angle = -40)

# Render the object.
s.render(image_name = os.path.join(PYVISI_EXAMPLE_IMAGES_PATH, IMAGE_NAME))
```

ContourOnPlaneCut class

```python
class ContourOnPlaneCut (scene, data_collector, viewport = Viewport.SOUTH_WEST, lut = Lut.COLOR,
                       cell_to_point = False, outline = True)
    This class works in a similar way to MapOnPlaneCut, except that it shows a scalar field using contour
    surfaces cut using a plane.

    The following are some of the methods available:
    Methods from Actor3D, ContourModule, Transform and DataSetMapper.
```

ContourOnPlaneClip class

```python
class ContourOnPlaneClip (scene, data_collector, viewport = Viewport.SOUTH_WEST, lut = Lut.COLOR,
                         cell_to_point = False, outline = True)
    This class works in a similar way to MapOnPlaneClip, except that it shows a scalar field using contour
    surfaces clipped using a plane.

    The following are some of the methods available:
    Methods from Actor3D, ContourModule, Transform, Clipper and DataSetMapper.
```

StreamLine class

```python
class StreamLine (scene, data_collector, viewport = Viewport.SOUTH_WEST, color_mode = ColorMode.VECTOR, lut = Lut.COLOR,
                  cell_to_point = False, outline = True)
    Class that shows the direction of particles of a vector field using streamlines. The streamlines can either
    be color or gray-scale, depending on the lookup table used. If the streamlines are colored, there are two
    possible coloring modes, either using vector data or scalar data.

    The following are some of the methods available:
    Methods from Actor3D, PointSource, StreamLineModule, Tube and DataSetMapper.

    A typical usage of StreamLine is shown below.

    """
    Author: John Ngui, john.ngui@uq.edu.au
    """
```

# Import the necessary modules.
from esys.pyvisi import Scene, DataCollector, StreamLine, Camera
from esys.pyvisi.constant import *
import os

PYVISI_EXAMPLE_MESHES_PATH = "data_meshes"
PYVISI_EXAMPLE_IMAGES_PATH = "data_test_images"
X_SIZE = 400
```
Y_SIZE = 400
VECTOR_FIELD_CELL_DATA = "temperature"
FILE_3D = "interior_3D.xml"
IMAGE_NAME = "streamline.jpg"
JPG_RENDERER = Renderer.ONLINE_JPG

# Create a Scene.
s = Scene(renderer = JPG_RENDERER, num_viewport = 1, x_size = X_SIZE,
y_size = Y_SIZE)

# Create a DataCollector reading from a XML file.
dc1 = DataCollector(source = Source.XML)
dc1.setFileName(file_name = os.path.join(PYVISI_EXAMPLE_MESHES_PATH, FILE_3D))

# Create streamlines.
sl1 = StreamLine(scene = s, data_collector = dc1,
                 viewport = Viewport.SOUTH_WEST, color_mode = ColorMode.SCALAR,
                 lut = Lut.COLOR, cell_to_point = False, outline = True)
sl1.setTubeRadius(radius = 0.02)
sl1.setTubeNumberOfSides(3)
sl1.setTubeRadiusToVaryByVector()
sl1.setPointSourceRadius(0.9)

# Create a Camera.
c1 = Camera(scene = s, viewport = Viewport.SOUTH_WEST)
c1.isometricView()

# Render the object.
s.render(image_name = os.path.join(PYVISI_EXAMPLE_IMAGES_PATH, IMAGE_NAME))

Carpet class

class Carpet (scene, data_collector, viewport = Viewport.SOUTH_WEST, warp_mode = WarpMode.SCALAR, lut = Lut.COLOR, cell_to_point = False, outline = True)
    This class works in a similar way to MapOnPlaneCut, except that it shows a scalar field cut on a plane and deformed (warped) along the normal. The plane can either be color or gray-scale, depending on the lookup table used. Similarly, the plane can be deformed either using scalar data or vector data.

The following are some of the methods available:
Methods from Actor3D, Warp, Transform and DataSetMapper.

A typical usage of Carpet is shown below.

Author: John Ngui, john.ngui@uq.edu.au

# Import the necessary modules.
from esys.pyvisi import Scene, DataCollector, Carpet, Camera
from esys.pyvisi.constant import *
import os

PYVISI_EXAMPLE_MESHES_PATH = "data_meshes"
PYVISI_EXAMPLE_IMAGES_PATH = "data_sample_images"
X_SIZE = 400
Y_SIZE = 400

SCALAR_FIELD_CELL_DATA = "temperature_cell"
FILE_3D = "interior_3D.xml"
IMAGE_NAME = "carpet.jpg"
JPG_RENDERER = Renderer.ONLINE_JPG
# Create a Scene.
s = Scene(renderer = JPG_RENDERER, num_viewport = 1, x_size = X_SIZE,
y_size = Y_SIZE)

# Create a DataCollector reading from a XML file.
dc1 = DataCollector(source = Source.XML)
dc1.setFileName(file_name = os.path.join(PYVISI_EXAMPLE_MESHES_PATH, FILE_3D))
dc1.setActiveScalar(scalar = SCALAR_FIELD_CELL_DATA)

# Create a Carpet.
cpt1 = Carpet(scene = s, data_collector = dc1, viewport = Viewport.SOUTH_WEST,
warp_mode = WarpMode.SCALAR, lut = Lut.COLOR, cell_to_point = True,
outline = True)
cpt1.setPlaneToXY(0.2)
cpt1.setScaleFactor(1.9)

# Create a Camera.
c1 = Camera(scene = s, viewport = Viewport.SOUTH_WEST)
c1.isometricView()

# Render the object.
s.render(image_name = os.path.join(PYVISI_EXAMPLE_IMAGES_PATH, IMAGE_NAME))

Legend class

class Legend (scene, data_collector, viewport = Viewport.SOUTH_WEST, lut = Lut.COLOR, legend = LegendType.SCALAR)
    Class that shows a scalar field on a domain surface. The domain surface can either be color or gray-scale,
depending on the lookup table used

The following are some of the methods available:
Methods from Actor3D, ScalarBar and DataSetMapper.

Rectangle class

class Rectangle (scene, viewport = Viewport.SOUTH_WEST)
    Class that generates a rectangle box.

The following are some of the methods available:
Methods from Actor3D, CubeSource and DataSetMapper.

Image class

class Image (scene, image_reader, viewport = Viewport.SOUTH_WEST)
    Class that displays an image which can be scaled (upwards and downwards) and has interaction capability.
The image can also be translated and rotated along the X, Y and Z axes. One of the most common use of
this feature is pasting an image on a surface map.

The following are some of the methods available:
Methods from Actor3D, PlaneSource and Transform.

A typical usage of Image is shown below.

Author: John Ngui, john.ngui@uq.edu.au

# Import the necessary modules.
from esys.pyvisi import Scene, DataCollector, Map, ImageReader, Image, Camera
from esys.pyvisi import GlobalPosition
from esys.pyvisi.constant import *

6.2. esys.pyvisi Classes 67
import os

PYVISI_EXAMPLE_MESHES_PATH = "data_meshes"
PYVISI_EXAMPLE_IMAGES_PATH = "data_sample_images"
X_SIZE = 400
Y_SIZE = 400

SCALAR_FIELD_POINT_DATA = "temperature"
FILE_3D = "interior_3D.xml"
LOAD_IMAGE_NAME = "flinders.jpg"
SAVE_IMAGE_NAME = "image.jpg"
JPG_RENDERER = Renderer.ONLINE_JPG

# Create a Scene.
s = Scene(renderer = JPG_RENDERER, num_viewport = 1, x_size = X_SIZE,
y_size = Y_SIZE)

# Create a DataCollector reading from a XML file.
dc1 = DataCollector(source = Source.XML)
dc1.setFileName(file_name = os.path.join(PYVISI_EXAMPLE_MESHES_PATH, FILE_3D))

# Create a Map.
m1 = Map(scene = s, data_collector = dc1, viewport = Viewport.SOUTH_WEST,
    lut = Lut.COLOR, cell_to_point = False, outline = True)
m1.setOpacity(0.3)

# Create an ImageReader (in place of DataCollector).
ir = ImageReader(ImageFormat.JPG)
ir.setImageName(image_name = os.path.join(PYVISI_EXAMPLE_MESHES_PATH, 
    LOAD_IMAGE_NAME))

# Create an Image.
i = Image(scene = s, image_reader = ir, viewport = Viewport.SOUTH_WEST)
i.setOpacity(opacity = 0.9)
i.translate(0,0,-1)
i.setPoint1(GlobalPosition(2,0,0))
i.setPoint2(GlobalPosition(0,2,0))

# Create a Camera.
c1 = Camera(scene = s, viewport = Viewport.SOUTH_WEST)

# Render the image.
s.render(image_name = os.path.join(PYVISI_EXAMPLE_IMAGES_PATH, SAVE_IMAGE_NAME))

Logo class

class Logo (scene, image_reader, viewport = Viewport.SOUTH_WEST):
    Class that displays a static image, in particular a logo (e.g. company symbol) and has NO interaction capability. The position and size of the logo can be specified.

    The following are some of the methods available:
    Methods from ImageReslice and Actor2D.

Movie class

class Movie (parameter_file = "make_movie")
    This class is used to create movies out of a series of images. The parameter specifies the name of a file that will contain the required information for the 'ppmtojpeg' command which is used to generate the movie.

    The following are some of the methods available:
imageRange (input_directory, first_image, last_image)
Use this method to specify that the movie is to be generated from image files with filenames in a certain
range (e.g. 'image000.jpg' to 'image050.jpg').

imageList (input_directory, image_list)
Use this method to specify a list of arbitrary image filenames from which the movie is to be generated.

makeMovie (movie)
Generate the movie with the specified filename.

A typical usage of Movie is shown below.

```python
# Import the necessary modules.
from esys.pyvisi import Scene, DataCollector, Map, Camera, Velocity, Legend
from esys.pyvisi import Movie, LocalPosition
from esys.pyvisi.constant import *
import os

PYVISI_EXAMPLE_MESHES_PATH = "data_meshes"
PYVISI_EXAMPLE_IMAGES_PATH = "data_sample_images"
X_SIZE = 800
Y_SIZE = 800

SCALAR_FIELD_POINT_DATA = "temp"
FILE_2D = "tempvel-"
IMAGE_NAME = "movie"
JPG_RENDERER = Renderer.OFFLINE_JPG

# Create a Scene.
s = Scene(renderer = JPG_RENDERER, num_viewport = 1, x_size = X_SIZE,
y_size = Y_SIZE)

# Create a DataCollector reading from a XML file.
dc1 = DataCollector(source = Source.XML)
dc1.setActiveScalar(scalar = SCALAR_FIELD_POINT_DATA)

# Create a Map.
m1 = Map(scene = s, data_collector = dc1,
viewport = Viewport.SOUTH_WEST, lut = Lut.COLOR, cell_to_point = False,
outline = True)

# Create a Camera.
cam1 = Camera(scene = s, viewport = Viewport.SOUTH_WEST)

# Create a movie.
mov = Movie()
lst = []

# Read in one file one after another and render the object.
for i in range(938, 949):
    dc1.setFileName(file_name = os.path.join(PYVISI_EXAMPLE_MESHES_PATH,
FILE_2D + "%06d.vtu" % i))
    s.render(image_name = os.path.join(PYVISI_EXAMPLE_IMAGES_PATH,
IMAGE_NAME + "%06d.jpg" % i))
    lst.append(IMAGE_NAME + "%06d.jpg" % i)

# Images (first and last inclusive) from which the movie is to be generated.
mov.imageRange(input_directory = PYVISI_EXAMPLE_IMAGES_PATH,
first_image = IMAGE_NAME + "000938.jpg",
last_image = IMAGE_NAME + "000938.jpg",
```
6.2.4 Coordinate Classes

This subsection details the instances used to position rendered objects.

LocalPosition class

class LocalPosition (x_coor, y_coor)
Class that defines a position (X and Y) in the local 2D coordinate system.

GlobalPosition class

class GlobalPosition (x_coor, y_coor, z_coor)
Class that defines a position (X, Y and Z) in the global 3D coordinate system.

6.2.5 Supporting Classes

This subsection details the supporting classes and their corresponding methods inherited by the input (see Section 6.2.2) and data visualization classes (see Section 6.2.3).

Actor3D class

Class that defines a 3D actor.

The following are some of the methods available:

setOpacity (opacity)
Set the opacity (transparency) of the 3D actor.

setColor (color)
Set the color of the 3D actor.

setRepresentationToWireframe ()
Set the representation of the 3D actor to wireframe.

Actor2D class

Class that defines a 2D actor.

The following are some of the methods available:

setPosition (position)
Set the position (XY) of the 2D actor. Default position is the lower left hand corner of the window / viewport.

Clipper class

Class that defines a clipper.
The following are some of the methods available:

setInsideOutOn()  
Clips one side of the rendered object.

setInsideOutOff()  
Clips the other side of the rendered object.

setClipValue(value)  
Set the scalar clip value (instead of using a plane) for the clipper.

ContourModule class  
Class that defines the contour module.

The following are some of the methods available:

generateContours(contours = None, lower_range = None, upper_range = None)  
Generate the specified number of contours within the specified range. In order to generate a single isosurface, the 'lower_range' and 'upper_range' must be set to the same value.

Glyph3D class  
Class that defines 3D glyphs.

The following are some of the methods available:

setScaleModeByVector()  
Set the 3D glyph to scale according to the vector data.

setScaleModeByScalar()  
Set the 3D glyph to scale according to the scalar data.

setScaleFactor(scale_factor)  
Set the 3D glyph scale factor.

TensorGlyph class  
Class that defines tensor glyphs.

The following are some of the methods available:

setScaleFactor(scale_factor)  
Set the scale factor for the tensor glyph.

setMaxScaleFactor(max_scale_factor)  
Set the maximum allowable scale factor for the tensor glyph.

PlaneSource class  
Class that defines a plane source. A plane source is defined by an origin and two other points, which form the axes (X and Y).

The following are some of the methods available:

setOrigin(position)  
Set the origin of the plane source.
setPoint1 (position)
    Set the first point from the origin of the plane source.

setPoint2 (position)
    Set the second point from the origin of the plane source.

**PointSource class**

Class that defines the source (location) to generate points. The points are generated within the radius of a sphere.

The following are some of the methods available:

**setPointSourceRadius (radius)**
    Set the radius of the sphere.

**setPointSourceCenter (center)**
    Set the center of the sphere.

**setPointSourceNumberOfPoints (points)**
    Set the number of points to generate within the sphere (the larger the number of points, the more streamlines are generated).

**Sphere class**

Class that defines a sphere.

The following are some of the methods available:

**setThetaResolution (resolution)**
    Set the theta resolution of the sphere.

**setPhiResolution (resolution)**
    Set the phi resolution of the sphere.

**StreamLineModule class**

Class that defines the streamline module.

The following are some of the methods available:

**setMaximumPropagationTime (time)**
    Set the maximum length of the streamline expressed in elapsed time.

**setIntegrationToBothDirections ()**
    Set the integration to occur both sides: forward (where the streamline goes) and backward (where the streamline came from).

**Transform class**

Class that defines the orientation of planes.

The following are some of the methods available:

**translate (x_offset, y_offset, z_offset)**
    Translate the rendered object along the x, y and z-axes.

**rotateX (angle)**
    Rotate the plane around the x-axis.
rotateY (angle)
    Rotate the plane around the y-axis.

rotateZ (angle)
    Rotate the plane around the z-axis.

setPlaneToXY (offset = 0)
    Set the plane orthogonal to the z-axis.

setPlaneToYZ (offset = 0)
    Set the plane orthogonal to the x-axis.

setPlaneToXZ (offset = 0)
    Set the plane orthogonal to the y-axis.

**Tube class**

Class that defines the tube wrapped around the streamlines.

The following are some of the methods available:

`setTubeRadius (radius)`
    Set the radius of the tube.

`setTubeRadiusToVaryByVector ()`
    Set the radius of the tube to vary by vector data.

`setTubeRadiusToVaryByScalar ()`
    Set the radius of the tube to vary by scalar data.

**Warp class**

Class that defines the deformation of a scalar field.

The following are some of the methods available:

`setScaleFactor (scale_factor)`
    Set the displacement scale factor.

**MaskPoints class**

Class that defines masking of points. This is useful to prevent the rendered object from being cluttered with arrows or ellipsoids.

The following are some of the methods available:

`setRatio (ratio)`
    Mask every n’th point.

`randomOn ()`
    Enables randomization of the points selected for masking.

**ScalarBar class**

Class that defines a scalar bar.

The following are some of the methods available:
setTitle \((title)\)
Set the title of the scalar bar.

setPosition \((position)\)
Set the local position of the scalar bar.

setOrientationToHorizontal ()
Set the orientation of the scalar bar to horizontal.

setOrientationToVertical ()
Set the orientation of the scalar bar to vertical.

setHeight \((height)\)
Set the height of the scalar bar.

setWidth \((width)\)
Set the width of the scalar bar.

setLabelColor \((color)\)
Set the color of the scalar bar’s label.

setTitleColor \((color)\)
Set the color of the scalar bar’s title.

ImageReslice class

Class that defines an image reslice which is used to resize static (no interaction capability) images (i.e. logo).

The following are some of the methods available:

setSize \((size)\)
Set the size factor of the image. The value must be between 0 and 2. Size 1 (one) keeps the image in its original size (which is the default).

DataSetMapper class

Class that defines a data set mapper.

The following are some of the methods available:

setScalarRange \((lower\_range, upper\_range)\)
Set the minimum and maximum scalar range for the data set mapper. This method is called when the range has been specified by the user. Therefore, the scalar range read from the source will be ignored.

CubeSource class

Class that defines a cube source. The center of the cube source defines the point from which the cube is to be generated and the X, Y and Z lengths define the length of the cube from the center point. If X length is 3, then the X length to the left and right of the center point is 1.5 respectively.

The following are some of the methods available:

setCenter \((center)\)
Set the cube source center.

setXLength \((length)\)
Set the cube source length along the x-axis.

setYLength \((length)\)
Set the cube source length along the y-axis.
setZLength \( (\text{length}) \)

Set the cube source length along the z-axis.

Rotation class

Class that sweeps 2D data around the z-axis to create a 3D looking effect.

The following are some of the methods available:

setResolution \( (\text{resolution}) \)

Set the resolution of the sweep for the rotation, which controls the number of intermediate points.

setAngle \( (\text{angle}) \)

Set the angle of rotation.

6.3 More Examples

This section provides examples for some common tasks.

6.3.1 Reading a Series of Files

The following script shows how to generate images from a time series using two data sources.

```python
# Import the necessary modules.
from esys.pyvisi import Scene, DataCollector, Contour, Camera
from esys.pyvisi.constant import *
import os

PYVISI_EXAMPLE_MESHES_PATH = "data_meshes"
PYVISI_EXAMPLE_IMAGES_PATH = "data_sample_images"
X_SIZE = 400
Y_SIZE = 300

SCALAR_FIELD_POINT_DATA_1 = "lava"
SCALAR_FIELD_POINT_DATA_2 = "talus"
FILE_2D = "phi_talus_lava."
IMAGE_NAME = "seriesofreads"
JPG_RENDERER = Renderer.ONLINE_JPG

# Create a Scene.
s = Scene(renderer = JPG_RENDERER, num_viewport = 1, x_size = X_SIZE, y_size = Y_SIZE)

# Create a DataCollector reading from an XML file.
dc1 = DataCollector(source = Source.XML)
dc1.setActiveScalar(scalar = SCALAR_FIELD_POINT_DATA_1)

# Create a Contour.
mosc1 = Contour(scene = s, data_collector = dc1, viewpoint = Viewport.SOUTH_WEST, lut = Lut.COLOR, cell_to_point = False, outline = True)
mosc1.generateContours(0)

# Create a second DataCollector reading from the same XML file
```

6.3. More Examples
# but specifying a different scalar field.
dc2 = DataCollector(source = Source.XML)
dc2 setActiveScalar(scalar = SCALAR_FIELD_POINT_DATA_2)

# Create a second Contour.
mosc2 = Contour(scene = s, data_collector = dc2,
                   viewport = Viewport.SOUTH_WEST, lut = Lut.COLOR, cell_to_point = False,
                   outline = True)
mosc2 generateContours(0)

# Create a Camera.
cam1 = Camera(scene = s, viewport = Viewport.SOUTH_WEST)

# Read in one file after another and render the object.
for i in range(99, 104):
            dc1 setFileName(file_name = os.path.join(PYVISI_EXAMPLE_MESHES_PATH, 
                             FILE_2D + "%04d.vtu") % i)
dc2.setFileName(file_name = os.path.join(PYVISI_EXAMPLE_MESHES_PATH, 
                             FILE_2D + "%04d.vtu") % i)
s.render(image_name = os.path.join(PYVISI_EXAMPLE_IMAGES_PATH, 
                             IMAGE_NAME + "%04d.jpg") % i)

6.3.2 Creating Slices of a Data Source

The following script shows how to save a series of images that slice the data at different points by gradually translating the cut plane.

```python
Author: John Ngui, john.ngui@uq.edu.au
```

# Import the necessary modules.
from esys.pyvisi import Scene, DataCollector, MapOnPlaneCut, Camera
from esys.pyvisi.constant import *
import os

PYVISI_EXAMPLE_MESHES_PATH = "data_meshes"
PYVISI_EXAMPLE_IMAGES_PATH = "data_sample_images"
X_SIZE = 400
Y_SIZE = 400

SCALAR_FIELD_POINT_DATA = "temperature"
FILE_3D = "interior_3D.xml"
IMAGE_NAME = "seriesofcuts"
JPG_RENDERER = Renderer.ONLINE_JPG

# Create a Scene.
s = Scene(renderer = JPG_RENDERER, num_viewport = 1, x_size = X_SIZE,
           y_size = Y_SIZE)

# Create a DataCollector reading from an XML file.
dcl = DataCollector(source = Source.XML)
dcl.setFileName(file_name = os.path.join(PYVISI_EXAMPLE_MESHES_PATH, FILE_3D))
dcl.setActiveScalar(scalar = SCALAR_FIELD_POINT_DATA)

# Create a MapOnPlaneCut.
mopc1 = MapOnPlaneCut(scene = s, data_collector = dcl,
                      viewport = Viewport.SOUTH_WEST, lut = Lut.COLOR, cell_to_point = False,
                      outline = True)
mopc1.setPlaneToYZ(offset = 0.1)
# Create a Camera.
c1 = Camera(scene = s, viewport = Viewport.SOUTH_WEST)
c1.isometricView()

# Render the object with multiple cuts using a series of translations.
for i in range(0, 5):
    s.render(image_name = os.path.join(PYVISI_EXAMPLE_IMAGES_PATH, IMAGE_NAME + "%02d.jpg") % i)
mopc1.translate(0.6,0,0)

6.3.3 Reading Data Directly from escript Objects

The following script shows how to combine Pyvisi code with escript code to generate visualizations on the fly.

```python
Author: Lutz Gross, l.gross@uq.edu.au
Author: John Ngui, john.ngui@uq.edu.au

# Import the necessary modules.
from esys.escript import *
from esys.escript.linearPDEs import LinearPDE
from esys.finley import Rectangle
from esys.pyvisi import Scene, DataCollector, Map, Camera
from esys.pyvisi.constant import *
import os
PYVISI_EXAMPLE_IMAGES_PATH = "data_sample_images"
X_SIZE = 400
Y_SIZE = 400
JPG_RENDERER = Renderer.ONLINE_JPG

#... set some parameters ...
xc = [0.02,0.002]
r = 0.001
qc = 50.e6
Tref = 0.
rhocp = 2.6e6
eta = 75.
kappa = 240.
tend = 5.
# initialize time, time step size and counter ...
t=0
h=0.1
i=0

# generate domain ...
mydomain = Rectangle(l0=0.05, l1=0.01, n0=250, n1=50)
# open PDE ...
mypde = LinearPDE(mydomain)
mypde.setSymmetryOn()
mypde.setValue(A=kappa*kronecker(mydomain), D=rhocp/h, d=eta, y=eta*Tref)
# set heat source: ...
x = mydomain.getX()
qH = qc*whereNegative(length(x-xc)-r)
# set initial temperature ....
T=Tref

# Create a Scene.
s = Scene(renderer = JPG_RENDERER, x_size = X_SIZE, y_size = Y_SIZE)
```

6.3. More Examples
# Create a DataCollector reading directly from escript objects.
dc = DataCollector(source = Source.ESCRIP)

# Create a Map.
m = Map(scene = s, data_collector = dc, 
    viewport = Viewport.SOUTH_WEST, lut = Lut.COLOR, 
    cell_to_point = False, outline = True)

# Create a Camera.
c = Camera(scene = s, viewport = Viewport.SOUTH_WEST)

# start iteration
while t < 0.4:
    i += 1
    t += h
    mypde.setValue(Y=qH+rhocp/h*T)
    T = mypde.getSolution()

    dc.setData(temp = T)

    # Render the object.
s.render(image_name = os.path.join(PYVISI_EXAMPLE_IMAGES_PATH, 
        "diffusion%02d.jpg") % i)
6.4 Useful Keys

This section lists keyboard shortcuts available when interacting with rendered objects using the Online approach.

<table>
<thead>
<tr>
<th>Key</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Keypress 'c' / 'a'</td>
<td>Toggle between the camera ('c') and object ('a') mode. In camera mode, mouse events affect the camera position and focal point. In object mode, mouse events affect the rendered object’s element (i.e. cut surface map, clipped velocity field, streamline, etc) that is under the mouse pointer.</td>
</tr>
<tr>
<td>Mouse button 1</td>
<td>Rotate the camera around its focal point (if in camera mode) or rotate the rendered object’s element (if in object mode).</td>
</tr>
<tr>
<td>Mouse button 2</td>
<td>Pan the camera (if in camera mode) or translate the rendered object’s element (if in object mode).</td>
</tr>
<tr>
<td>Mouse button 3</td>
<td>Zoom the camera (if in camera mode) or scale the rendered object’s element (if in object mode).</td>
</tr>
<tr>
<td>Keypress 3</td>
<td>Toggle the render window in and out of stereo mode. By default, red-blue stereo pairs are created.</td>
</tr>
<tr>
<td>Keypress 'e' / 'q'</td>
<td>Exit the application if only one file is to be read, or read and display the next file if multiple files are to be read.</td>
</tr>
<tr>
<td>Keypress 's'</td>
<td>Modify the representation of the rendered object to surfaces.</td>
</tr>
<tr>
<td>Keypress 'w'</td>
<td>Modify the representation of the rendered object to wireframe.</td>
</tr>
<tr>
<td>Keypress 'r'</td>
<td>Reset the position of the rendered object to the center.</td>
</tr>
</tbody>
</table>

Table 6.1: Useful keys in Online render mode
6.5 Sample Output

This section shows sample images produced with the various classes of Pyvisi. The source code to produce these images is included in the Pyvisi distribution.

Map

MapOnPlaneCut

MapOnPlaneClip

MapOnScalarClip

MapOnScalarClipWithRotation

Streamline

Velocity

VelocityOnPlaneCut

VelocityOnPlaneClip

Ellipsoid

EllipsoidOnPlaneCut

EllipsoidOnPlaneClip
6.5. Sample Output
The following sections give a brief overview of the model classes and their corresponding methods.

7.1 Stokes Problem

The velocity field \( v \) and pressure \( p \) of an incompressible fluid is given as the solution of the Stokes problem

\[
- (\eta(v_{i,j} + v_{i,j}))_j + p_{,i} = f_i - \sigma_{ij,j} \tag{7.1}
\]

where \( \eta \) is the viscosity, \( F_i \) defines an internal force and \( \sigma_{ij} \) is an initial stress. We assume an incompressible media:

\[
-v_{i,i} = 0 \tag{7.2}
\]

Natural boundary conditions are taken in the form

\[
(\eta(v_{i,j} + v_{i,j})) n_j - n_{i} p = s_{i} + \sigma_{ij} n_{i} \tag{7.3}
\]

which can be overwritten by constraints of the form

\[
v_i(x) = v^D_i(x) \tag{7.4}
\]

at some locations \( x \) at the boundary of the domain. The index \( i \) may depend on the location \( x \) on the boundary. \( v^D \) is a given function on the domain.

7.1.1 Solution Method

In block form equation equations 7.1 and 7.2 takes the form of a saddle point problem

\[
\begin{bmatrix}
A & B^* \\
B & 0
\end{bmatrix}
\begin{bmatrix}
v \\
p
\end{bmatrix} =
\begin{bmatrix}
G \\
0
\end{bmatrix} \tag{7.5}
\]

where \( A \) is coercive, self-adjoint linear operator in a suitable Hilbert space, \( B \) is the \((-1)\)-divergence operator and \( B^* \) is its adjoint operator (=gradient operator). For more details on the mathematics see references [4, 5]. We use iterative techniques to solve this problem. To make sure that the incompressibility condition holds with sufficient accuracy we check for

\[
\|v_{k,k}\| \leq \epsilon \sqrt{\int_{\Omega} v_{j,k} v_{j,k} dx} \tag{7.6}
\]

where \( \epsilon \) is the desired relative accuracy and

\[
\|p\|^2 = \int_\Omega p^2 \ dx \tag{7.7}
\]

defines the \( L^2 \)-norm. We use the Uzawa scheme to solve the problem.

In fact the first equation in 7.5 gives for a known pressure

\[
v = A^{-1}(G - B^*p) \tag{7.8}
\]
which is inserted into the second equation leading to

\[ Sp = BA^{-1}G \]  

(7.9)

with the Schur complement \( S = BA^{-1}B^* \). This problem can be solved iteratively with the preconditioner \( \hat{S} \) defined as \( q = \hat{S}^{-1}p \) by solving

\[ \frac{1}{\eta} q = p \]  

(7.10)

see [7] for more details. Note that the residual for the current approximation \( p \) is given as

\[ r = BA^{-1}(G - B^*p) = Bv \]  

(7.11)

where \( v \) is given by 7.8.

If one uses the generalized minimal residual method (GMRES) the method is directly applied to the preconditioned system

\[ \hat{S}^{-1}Sp = \hat{S}^{-1}BA^{-1}G \]  

(7.12)

We use the norm

\[ ||p||_{GMRES} = ||\hat{S}p|| \]  

(7.13)

Notice that for the residual \( \hat{r} = \hat{S}^{-1}r \) one has

\[ ||\hat{r}||_{GMRES} \leq ATOL \]  

(7.14)

If \( p^0 \) provides an initial guess for the pressure we use 7.8 to get a first initial guess for the velocity \( v^0 \) which we use to set an absolute tolerance \( ATOL = \epsilon \sqrt{\sqrt{v_{j,k}^0 v_{j,k}^0}} \). The GMRES is terminated when

\[ ||\hat{r}||_{GMRES} \leq ATOL \]  

(7.15)

Notice that \( ||\hat{r}||_{GMRES} = ||r|| = ||Bv|| = ||v_{k,k}|| \) so we can expect that the target stopping criterion 7.6 is fulfilled. However, if \( v \) is very different from the initial choice of \( v^0 \) the value of \( ATOL \) is corrected and GMRES is restarted with a new tolerance. For time dependent problems this approach works well as value for \( p \) form a previous time step provides a good initial guess.

Alternatively, as \( S \) is symmetric and positive definite one can apply the preconditioned conjugate gradient method (PCG). PCG use the norm

\[ ||r||_{2_{PCG}} = \int_\Omega r \hat{S}^{-1}r \ dx = \int_\Omega \eta^2 \ dx \]  

(7.16)

To take the extra factor \( \eta \) into consideration when checking the stopping criterion we use the following definition for \( ATOL \):

\[ ATOL = \epsilon \sqrt{\sqrt{v_{j,k}^0 v_{j,k}^0}} ||v_{k,k}||_{PCG} \]  

(7.17)

7.1.2 Functions

class StokesProblemCartesian (domain)

opens the Stokes problem on the Domain domain. The approximation order needs to be two.

initialize ([f=Data(), fixed_u_mask=Data(), eta=1, [surface_stress=Data(), [stress=Data()]]])

assigns values to the model parameters. In any call all values must be set. \( f \) defines the external force \( f \), \( \eta \) the viscosity \( \eta \), \( surface_{stress} \) the surface stress \( s \) and \( stress \) the initial stress \( \sigma \). The locations and components where the velocity is fixed are set by the values of \( fixed_u_mask \). The method will try to cast the given values to appropriate Data class objects.

solve ([v,p], [max_iter=20, verbose=False, [usePCG=True]])

solves the problem and return approximations for velocity and pressure. The arguments \( v \) and \( p \) define initial guess. The values of \( v \) marked by \( fixed_u_mask \) remain unchanged. If \( usePCG \) is set to True reconditioned conjugate gradient method (PCG) scheme is used. Otherwise the problem is solved generalized minimal residual method (GMRES). In most cases the PCG scheme is more efficient. \( max_{-iter} \) defines the maximum number of iteration steps. If \( verbose \) is set to True informations on the progress of of the solver are printed.
setTolerance([tolerance=1.e-4])
sets the tolerance in an appropriate norm relative to the right hand side. The tolerance must be non-negative and less than 1.

gdTolerance()
returns the current relative tolerance.

setAbsoluteTolerance([tolerance=0.])
sets the absolute tolerance for the error in the relevant norm. The tolerance must be non-negative. Typically the absolute tolerance is set to 0.

gdAbsoluteTolerance()
returns the current absolute tolerance.

setSubProblemTolerance([rtol=None])
sets the tolerance to solve the involved PDEs. The subtolerance rtol should not be chosen too large in order to avoid feedback of errors in the subproblem solution into the outer iteration. On the other hand is chosen too small compute time is wasted. If rtol is set to None the sub-tolerance is set automatically depending on the tolerance chosen for the outer iteration.

gdSubProblemTolerance()
return the tolerance for the involved PDEs.

7.1.3 Example: Lit Driven Cavity

The following script `lit_driven_cavity.py` which is available in the example directory illustrates the usage of the StokesProblemCartesian class to solve the lit driven cavity problem:

```python
from esys.escript import *
from esys.finley import Rectangle
from esys.escript.models import StokesProblemCartesian
NE=25
dom = Rectangle(NE,NE,order=2)
x = dom.getX()
s=StokesProblemCartesian(dom)
m= (whereZero(x[0])*[1.,0]+whereZero(x[0]-1))*[1.,0] + 
   (whereZero(x[1])*[0.,1.]+whereZero(x[1]-1))*[1.,1]
s.initialize(eta=.1, fixed_u_mask= m)
v=Vector(0.,Solution(dom))
v[0]+=whereZero(x[1]-1.)
p=Scalar(0.,ReducedSolution(dom))
v,p=s.solve(v,p, verbose=True)
saveVTK("u.xml",velocity=v,pressure=p)
```

7.2 Darcy Flux

We want to calculate the velocity $u$ and pressure $p$ on a domain $\Omega$ solving the Darcy flux problem:

$$
\begin{align*}
  u_i + \kappa_{ij}p_j &= g_i \\
  u_{k,k} &= f
\end{align*}
$$

(7.18)

with the boundary conditions

$$
\begin{align*}
  u_i n_i &= u_i^N n_i & \text{on } & \Gamma_N \\
  p &= p^D & \text{on } & \Gamma_D
\end{align*}
$$

(7.19)

where $\Gamma_N$ and $\Gamma_D$ are a partition of the boundary of $\Omega$ with $\Gamma_D$ non empty, $n_i$ is the outer normal field of the boundary of $\Omega$, $u_i^N$ and $p^D$ are given functions on $\Omega$, $g_i$ and $f$ are given source terms and $\kappa_{ij}$ is the given permability. We assume that $\kappa_{ij}$ is symmetric (which is not really required) and positive definite, i.e there are positive constants $\alpha_0$ and $\alpha_1$ which are independent from the location in $\Omega$ such that

$$
\alpha_0 x_i x_i \leq \kappa_{ij} x_i x_j \leq \alpha_1 x_i x_i
$$

(7.20)

for all $x_i$. 

7.2. Darcy Flux
7.2.1 Solution Method

In practical applications it is an advantage to calculate the pressure $p$ as a correction of a 'static' pressure $p^{ref}$ which is the solution of

$$-(\kappa_{ki}\kappa_{kj}p^{ref})_{ij} = -(\kappa_{ki}(g_k - u^N_k))_{ij} \quad \text{with } p^{ref} = p^D \text{ on } \Gamma_D$$

(7.21)

With setting $u \leftarrow u - u^N$ and $p \leftarrow p - p^{ref}$ and

$$g_i \leftarrow g_i - u^N_i - \kappa_{ij}p^{ref}$$

$$f \leftarrow f - u^N_k$$

(7.22)

we can assume that $u^N_i n_i = 0$ and $p^D = 0$. We set

$$V = \{ q \in H^1(\Omega) : q = 0 \text{ on } \Gamma_D \}$$

(7.23)

and

$$W = \{ v \in (L^2(\Omega))^d : v_{k,k} \in L^2(\Omega) \text{ and } u_i n_i = 0 \text{ on } \Gamma_N \}$$

(7.24)

and define the operator $Q : V \rightarrow (L^2(\Omega))^d$ defined by

$$(Qp)_i = \kappa_{ij}p_{j,i}$$

(7.25)

and the operator $D : W \rightarrow L^2(\Omega)$ defined by

$$Dv = v_{k,k}$$

(7.26)

In operator notation the Darcy problem 7.18 is written in the form

$$u + Qp = g$$

$$Du = f$$

(7.27)

We solve this equation by minimising the functional

$$J(u, p) := \|u + Qp - g\|_0^2 + \|Du - f\|_0^2$$

(7.28)

over $W \times V$ where $\|\|_0$ denotes the norm in $L^2(\Omega)$. A simple calculation shows that one has to solve

$$(v + Qq, u + Qp - g) + (Dv, Du - f) = 0$$

(7.29)

for all $v \in W$ and $q \in V$, which translates back into operator notation

$$I + D^*D)u + Qp = D^*f + g$$

$$Q^*u + Q^*Qp = Q^*g$$

(7.30)

where $D^*$ and $Q^*$ denote the adjoint operators. In [3] it has been shown that this problem is continuous and coercive in $W \times V$ and therefore has a unique solution. Also standard FEM methods can be used for discretization.

It is also possible to solve the problem is coupled form, however this approach leads in some cases to a very ill-conditioned stiffness matrix in particular in the case of a very small or large permeability ($\alpha_1 \ll 1$ or $\alpha_0 \gg 1$).

The approach we are taking is to eliminate the velocity $u$ from the problem. Assuming that $p$ is known we have

$$v = (I + D^*D)^{-1}(D^*f + g - Qp)$$

(7.31)

(notice that $(I + D^*D)$ is coercive in $W$) which is inserted into the second equation

$$Q^*(I + D^*D)^{-1}(D^*f + g - Qp) + Q^*Qp = Q^*g$$

(7.32)

which is

$$Q^*(I - (I + D^*D)^{-1})Qp = Q^*(g - (I + D^*D)^{-1}(D^*f + g))$$

(7.33)

We use the PCG method to solve this. The residual $r (\in V^*)$ is given as

$$r = Q^*(g - (I + D^*D)^{-1}(D^*f + g) - Qp + (I + D^*D)^{-1}Qp)$$

(7.34)
So in a partial implementation we use $\hat{r} = g - Qp - v$ to represent the residual. The evaluation of the iteration operator for a given $p$ is then returning $Qp + v$ where $v$ is the solution of

$$(I + D^*D)v = Qp$$

(7.35)

We use $(Q^*Q)^{-1}$ as a preconditioner for the iteration operator $Q^*(I - (I + D^*D)^{-1})Q$. So the application of the preconditioner to $\hat{r}$ representing the residual is given by solving implemented by solving

$$Q^*Qq = Q^*\hat{r}$$

(7.36)

The residual norm used in the PCG is given as

$$\|r\|_{PCG}^2 = \int r \cdot (Q^*Q)^{-1}r \, dx = \int \hat{r} \cdot Q(Q^*Q)^{-1}Q^*\hat{r} \, dx \approx \|\hat{r}\|_0^2$$

(7.37)

The iteration is terminated if

$$\|r\|_{PCG} \leq ATOL$$

(7.38)

where we set

$$ATOL = atol + rtol \cdot \left( \frac{1}{\|v\|_0} + \frac{1}{\|Qp\|_0} \right)^{-1}$$

(7.39)

where $rtol$ is a given relative tolerance and $atol$ is a given absolute tolerance (typically = 0). Notice that if $Qp$ and $v$ both are zero, the pair $(0, p)$ is a solution. The problem is that $ATOL$ is depending on the solution $p$ and $v$ calculated form 7.31). In particular one use the initial guess for $p$ to get a first value for ATOL. If the stopping criterion is met in the PCG iteration, a new $v$ is calculated from the current pressure approximation and $ATOL$ is recalculated. If 7.38 is still fulfilled the calculation is terminated and $(v, p)$ is returned. Otherwise PCG is restarted with a new ATOL.

7.2.2 Functions

class DarcyFlow (domain)

opens the Darcy flux problem on the Domain domain.

setValue ([$f=None$, $g=None$, $location_of_fixed_pressure=None$, $location_of_fixed_flux=None$, $permeability=None$])

assigns values to the model parameters. Values can be assigned using various calls - in particular in a time dependent problem only values that change over time needs to be reset. The permeability can be defined as scalar (isotropic), a vector (orthotropic) or a matrix (anisotropic). $f$ and $g$ are the corresponding parameters in 7.18. The locations and components where the flux is prescribed are set by positive values in location_of_fixed_flux. The locations where the pressure is prescribed are set by by positive values of location_of_fixed_pressure. The values of the pressure and flux are defined by the initial guess. Notice that at any point on the boundary of the domain the pressure or the normal component of the flux must be defined. There must be at least one point where the pressure is prescribed. The method will try to cast the given values to appropriate Data class objects.

setTolerance ([$rtol=1e-4$])

sets the relative tolerance $rtol$ in 7.39.

setAbsoluteTolerance ([$atol=0.$])

sets the absolute tolerance $atol$ in 7.39.

setSubProblemTolerance ([$rtol=None$])

sets the relative tolerance used to solve the involved PDEs. If no argument is given, the square of the current relative tolerance is used. The sub-problem tolerance should be choosen as large as possible to minimize the compute time. However, a too large value for the sub-problem tolerance may lead to slow convergence or even divergence in the outer iteration.

solve ($u0,p0$, $[max_iter=100$, $verbose=False$, $sub_rtol=1.e-8$])

solves the problem. And returns approximations for the flux $v$ and the pressure $p$. $u0$ and $p0$ define initial guess for flux and pressure. Values marked by positive values location_of_fixed_flux and location_of_fixed_pressure, respectively, are kept unchanged.

7.2. Darcy Flux 87
7.2.3 Example: Gravity Flow

7.3 Level Set Method

The Level Set Method is used for tracking interfaces between two different types of fluids, which may have different physical parameters for density or viscosity. The interface is represented by a signed distance function, $\phi(x)$, where the isocontour at $\phi(x) = 0$ is used to define the interface. A point in the domain can then be determined on which side of the interface it resides, based on the local sign of $\phi(x)$; for example positive $\phi(x)$ on one side of the interface and negative $\phi(x)$ on the other. Parameters values such as density and viscosity can then be defined for the two different mediums. The Level Set Method consists of two procedures, the advection and reinitialization of the signed distance function, $\phi$. The LevelSet class can be used in conjunction with the StokesProblemCartesian class for solving computational fluid dynamics problems involving the tracking of the interface. The advantage of the Level Set Method is that it can be used to track surfaces that break apart or intersect. Also, the Level Set Method avoids the need for remeshing, which is required by the Lagrangian-Eulerian (ALE) method.

7.3.1 Solution Method

The displacement of the interface at the zero isocontour of $\phi(x)$ is calculated each time-step by using the velocity field. This is achieved by solving the advection equation:

$$\frac{\partial \phi}{\partial t} + \vec{v} \cdot \nabla \phi = 0,$$  \hspace{1cm} (7.40)

where $\vec{v}$ is the velocity field. The advection equation is solved using a Taylor-Galerkin scheme with the presence of diffusion; by expanding $\phi$ into a Taylor series:

$$\phi^+ \approx \phi^- + dt \frac{\partial \phi^-}{\partial t} + \frac{dt^2}{2} \frac{\partial^2 \phi^-}{\partial t^2},$$  \hspace{1cm} (7.41)

then by inserting

$$\frac{\partial \phi^-}{\partial t} = -\vec{v} \cdot \nabla \phi^-,$$  \hspace{1cm} (7.42)

and

$$\frac{\partial^2 \phi^-}{\partial t^2} = \frac{\partial}{\partial t}(-\vec{v} \cdot \nabla \phi^-) = \vec{v} \cdot \nabla (\vec{v} \cdot \nabla \phi^-),$$  \hspace{1cm} (7.43)

into Equation (7.41), the calculation of the level set function is given by:

$$\phi^+ = \phi^- - dt\vec{v} \cdot \nabla \phi^- + \frac{dt^2}{2} \vec{v} \cdot \nabla (\vec{v} \cdot \nabla \phi^-).$$  \hspace{1cm} (7.44)

If $\nabla \cdot \vec{v} = 0$ is assumed, then the calculation of the second order derivatives in Equation (7.44) can be avoided.

As the computation of the distance function progresses, it becomes distorted, and so it needs to be updated in order to stay regular [21]. This process is known as the reinitialization procedure. The aim is to iteratively find a solution to the reinitialization equation:

$$\frac{\partial \psi}{\partial \tau} + \text{sign}(\phi)(1 - \nabla \psi) = 0,$$  \hspace{1cm} (7.45)

where $\psi$ shares the same level set with $\phi$, $\tau$ is pseudo time, and $\text{sign}(\phi)$ is the smoothed sign function. This equation is solved to meet the definition of the level set function, $|\nabla \psi| = 1$; the normalization condition. Equation (7.45) can be rewritten in a similar form to the advection equation:

$$\frac{\partial \psi}{\partial \tau} + \vec{w} \cdot \nabla \psi = \text{sign}(\phi),$$  \hspace{1cm} (7.46)
where
\[ \vec{w} = \text{sign}(\phi) \frac{\nabla \psi}{|\nabla \psi|}. \] (7.47)

\( \vec{w} \) is the characteristic velocity pointing outward from the free surface. Equation (7.46) can be solved by a similar technique to what was used in the advection step, using the Taylor-Galerkin procedure. When the distance function, \( \phi \), is calculated, the physical parameters, density and viscosity, are updated using the sign of \( \phi \). The region along the interface is assumed to be of finite thickness of \( \alpha h \), where \( h \) is the size of the elements in the computational mesh and \( \alpha \) is a smoothing parameter. The parameters are updated by the following expression:

\[ P = \begin{cases} P_1 & \text{where } \psi < -\alpha h \\ P_2 & \text{where } \psi > \alpha h \\ (P_2 - P_1) \psi / 2\alpha h + (P_1 + P_2)/2 & \text{where } |\psi| < \alpha h. \end{cases} \] (7.48)

where the subscripts 1 and 2 denote the different fluids.

### 7.3.2 Functions

**class LevelSet**(domain, func, reinit_max, reinit_each, tolerance, smooth)

opens the LevelSet on the Domain domain. `func` defines the initial Level Set function representing the interface between two fluids. `reinit_max` sets the maximum number of iterations to satisfy the normal condition, \(|\nabla \phi| = 1\), during the reinitialization of the Level Set function. `reinit_each` sets the frequency of reinitialization for a number of time-steps. `tolerance` sets the convergence tolerance to satisfy the normal condition during the reinitialization of the Level Set function. `smooth` sets the bandwidth of size \( 2\alpha h \) along the interface to smooth the physical parameters of density and viscosity; \( h \) is the size of the elements in the mesh and \( \alpha \) is the smoothing parameter, usually set to 1.

**update_parameter**(par1, par2)

updates the physical parameters using the sign of \( \phi \). `par1` and `par2` are the physical parameter values for fluid1 and fluid2 respectively. Usually this method is called twice during each time-step to update the density and viscosity of the two fluids.

**update_phi**(vel, dt, t_step)

updates the Level Set function. It performs the advection and reinitialization procedures. `vel` is the velocity field of the fluid domain, `dt` is the time-step size, and `t_step` is the current time-step to determine when to reinitialize.

### 7.4 Isotropic Kelvin Material

As proposed by Kelvin [13] material strain \( D_{ij} = v_{i,j} + v_{j,i} \) can be decomposed into an elastic part \( D_{ij}^e \) and visco-plastic part \( D_{ij}^{vp} \):

\[ D_{ij} = D_{ij}^e + D_{ij}^{vp} \] (7.49)

with the elastic strain given as

\[ D_{ij}^{el} = \frac{1}{2\mu} \sigma_{ij}'. \] (7.50)

where \( \sigma_{ij}' \) is the deviatoric stress (Notice that \( \sigma_{ii}' = 0 \)). If the material is composed by materials \( q \) the visco-plastic strain can be decomposed as

\[ D_{ij}^{vp} = \sum_q D_{ij}^{q} \] (7.51)

where \( D_{ij}^{q} \) is the strain in material \( q \) given as

\[ D_{ij}^{q} = \frac{1}{2\eta^q} \sigma_{ij}'. \] (7.52)

where \( \eta^q \) is the viscosity of material \( q \). We assume the following between the the strain in material \( q \)

\[ \eta^q = \eta_N \left( \frac{\tau}{\eta_N} \right)^{\frac{1}{n}} \] with \( \tau = \sqrt{\frac{1}{2} \sigma_{ij}' \sigma_{ij}'} \) (7.53)
for a given power law coefficients $n^q \geq 1$ and transition stresses $\tau^q_t$, see [13]. Notice that $n^q = 1$ gives a constant viscosity. After inserting equation 7.52 into equation 7.51 one gets:

$$D'_{ij} = \frac{1}{2 \eta^{vp}} \sigma'_{ij} \text{ with } \frac{1}{\eta^{vp}} = \sum_q \frac{1}{\eta^q}. \quad (7.54)$$

and finally with 7.49

$$D'_{ij} = \frac{1}{2 \eta^{vp}} \sigma'_{ij} + \frac{1}{2 \mu} \dot{\gamma}$$

The total stress $\tau$ needs to fulfill the yield condition

$$\tau \leq \tau_Y + \beta p \quad (7.56)$$

with the Drucker-Prager cohesion factor $\tau_Y$, Drucker-Prager friction $\beta$ and total pressure $p$. The deviatoric stress needs to fulfill the equilibrion equation

$$-\sigma'_{ij,j} + p_{,i} = F_i \quad (7.57)$$

where $F_j$ is a given external force. We assume an incompressible media:

$$-v_{i,i} = 0 \quad (7.58)$$

Natural boundary conditions are taken in the form

$$\sigma'_{ij} n_j - n_i p = f \quad (7.59)$$

which can be overwritten by a constraint

$$v_i(x) = 0 \quad (7.60)$$

where the index $i$ may depend on the location $x$ on the boundary.

### 7.4.1 Solution Method

By using a first order finite difference approximation wit step size $dt > 0$ 7.50 get the form

$$\dot{\gamma} = \frac{1}{dt} \left( \sigma_{ij} - \sigma_{ij}^- \right) \quad (7.61)$$

and

$$D'_{ij} = \left( \frac{1}{2 \eta^{vp}} + \frac{1}{2 \mu dt} \right) \sigma'_{ij} + \frac{1}{2 \mu dt} \sigma_{ij}' \quad (7.62)$$

where $\sigma_{ij}^-$ is the stress at the precious time step. With

$$\dot{\gamma} = \sqrt{2 \left( \frac{D'_{ij} - \frac{1}{2 \mu dt} \sigma_{ij}'}{2} \right)^2} \quad (7.63)$$

we have

$$\tau = \eta_{eff} \cdot \dot{\gamma} \quad (7.64)$$

where

$$\eta_{eff} = \min \left( \frac{1}{\mu dt} + \frac{1}{\eta^{vp}}, \eta_{max} \right) \text{ with } \eta_{max} = \begin{cases} \frac{\tau_Y + \beta p}{\dot{\gamma}} & \text{if } \dot{\gamma} > 0 \\ \infty & \text{otherwise} \end{cases} \quad (7.65)$$

The upper bound $\eta_{max}$ makes sure that yield condition 7.56 holds. With this setting the equation 7.62 takes the form

$$\sigma'_{ij} = 2 \eta_{eff} \left( D'_{ij} - \frac{1}{2 \mu dt} \sigma_{ij}' \right) \quad (7.66)$$

After inserting 7.66 into 7.57 we get

$$-(\eta_{eff}(v_{i,j} + v_{i,j})) + p_{,i} = F_i - \frac{\eta_{eff}}{\mu dt} \sigma_{ij}' \quad (7.67)$$
Combining this with the incompressibility condition 7.49 we need to solve a Stokes problem as discussed in section 7.1.1 in each time step.

If we set
\[
\frac{1}{\eta(\tau)} = \frac{\mu}{t} + \frac{1}{\eta^{(p)}}
\]  
(7.68)
we need to solve the nonlinear problem
\[
\eta_{eff} - \min(\eta(\dot{\gamma} \cdot \eta_{eff}), \eta_{max}) = 0
\]  
(7.69)
We use the Newton-Raphson Scheme to solve this problem
\[
\eta_{eff}^{(n+1)} = \min(\eta_{max}, \eta_{eff}^{(n)} - \frac{\eta_{eff} - \eta(\tau^{(n)})}{1 - \dot{\gamma} \cdot \eta'(\tau^{(n)})}) = \min(\eta_{max}, \frac{\eta(\tau^{(n)}) - \tau^{(n)} \cdot \eta'(\tau^{(n)})}{1 - \dot{\gamma} \cdot \eta'(\tau^{(n)})})
\]  
(7.70)
where \(\eta'\) denotes the derivative of \(\eta\) with respect of \(\tau\) and \(\tau^{(n)} = \dot{\gamma} \cdot \eta_{eff}^{(n)}\).

Looking at the evaluation of \(\eta\) in 7.68 it makes sense formulate the iteration 7.70 using \(\Theta = \eta^{-1}\). In fact we have
\[
\eta' = -\frac{\Theta'}{\Theta^2} \text{ with } \Theta' = \sum_q \left(\frac{1}{\eta^{(q)}}\right)'
\]  
(7.71)
As
\[
\left(\frac{1}{\eta^{(q)}}\right)' = \frac{1 - \frac{1}{\eta^{(q)}}}{\eta^{(q)}} \tau - \frac{1}{\tau} \frac{1}{\eta^{(q)}}
\]  
(7.72)
we have
\[
\Theta' = \frac{1}{\tau} \omega \text{ with } \omega = \sum_q \frac{1 - \frac{1}{\eta^{(q)}}}{\eta^{(q)}}
\]  
(7.73)
which leads to
\[
\eta_{eff}^{(n+1)} = \min(\eta_{max}, \eta_{eff}^{(n)} \frac{\Theta^{(n)} + \omega^{(n)}}{\eta_{eff}^{(n)} \Theta^{(n)} + \omega^{(n)}})
\]  
(7.74)
8.1 Einstein Notation

Compact notation is used in equations such as continuum mechanics and linear algebra; it is known as Einstein notation or the Einstein summation convention. It makes the conventional notation of equations involving tensors more compact, by shortening and simplifying them.

There are two rules which make up the convention:

Firstly, the rank of the tensor is represented by an index. For example, $a$ is a scalar; $b_i$ represents a vector; and $c_{ij}$ represents a matrix.

Secondly, if an expression contains subscripted variables, they are assumed to be summed over all possible values, from 0 to $n$. For example, for the following expression:

$$ y = a_0 b_0 + a_1 b_1 + \ldots + a_n b_n $$

(8.1)

can be represented as:

$$ y = \sum_{i=0}^{n} a_i b_i $$

(8.2)

then in Einstein notion:

$$ y = a_i b_i $$

(8.3)

Another example:

$$ \nabla p = \frac{\partial p}{\partial x_0} i + \frac{\partial p}{\partial x_1} j + \frac{\partial p}{\partial x_2} k $$

(8.4)

can be expressed in Einstein notation as:

$$ \nabla p = p_i $$

(8.5)

where the comma ',' indicates the partial derivative.

For a tensor:

$$ \sigma_{ij} = \begin{bmatrix} \sigma_{00} & \sigma_{01} & \sigma_{02} \\ \sigma_{10} & \sigma_{11} & \sigma_{12} \\ \sigma_{20} & \sigma_{21} & \sigma_{22} \end{bmatrix} $$

(8.6)

The $\delta_{ij}$ is the Kronecker $\delta$-symbol, which is a matrix with ones for its diagonal entries ($i = j$) and zeros for the remaining entries ($i \neq j$).
\[ \delta_{ij} = \begin{cases} 1, & \text{if } i = j \\ 0, & \text{if } i \neq j \end{cases} \quad (8.7) \]
INDEX

acos() (in module esys.escript), 40
acosh() (in module esys.escript), 40
addItem() (PropertySet method), 54
addItem() (Design method), 55
AMG (LinearPDE attribute), 47
Arc (class in esys.pycad), 52
asin() (in module esys.escript), 40
asinh() (in module esys.escript), 40
atan() (in module esys.escript), 40
atanh() (in module esys.escript), 40
azimuth() (Camera method), 58
backView() (Camera method), 58
backward Euler, 7
BezierCurve (class in esys.pycad), 52
BiCGSTAB (LinearPDE attribute), 47
BiCGStab, 46
boldOn () (Text2D method), 59
bottomView() (Camera method), 58
boundary condition
natural, 7, 15, 43
boundary value problem, 2
BVP, 2, 4
BSpline (class in esys.pycad), 52
Camera (class in ), 58
Carpet (class in ), 66
CGS (LinearPDE attribute), 47
characteristic function, 4, 5, 8, 43
checkSymmetry() (LinearPDE method), 45
CHOLEVSKY (LinearPDE attribute), 47
clearItems() (Design method), 55
clearItems() (PropertySet method), 54
clip() (in module esys.escript), 38
cohesion factor, 90
constraint, 7, 15, 43
ContinuousFunction() (in module esys.escript), 33
Contour (class in ), 64
ContourOnPlaneClip (class in ), 65
ContourOnPlaneCut (class in ), 65
cos() (in module esys.escript), 40
cosh() (in module esys.escript), 40
Courant condition, 14
Curve (class in esys.pycad), 52
CurveLoop (class in esys.pycad), 52
Darcy flow, 85
Darcy flux, 85, 87
DarcyFlow (class in ), 87
data sample
points, 28, 32, 34–38
DataCollector (class in ), 59
DEFAULT (LinearPDE attribute), 46
DEG (data in esys.pycad), 53
DELAUNAY (Design attribute), 54
Design (class in esys.pycad.gmsh), 54
diffusion equation, 5
dilation (class in esys.pycad), 53
DIRECT (LinearPDE attribute), 46
Dirichlet boundary condition, 4
homogeneous, 2, 5
discontinuity, 27, 43, 44
div() (in module esys.escript), 39
dolly() (Camera method), 58
Domain (class in esys.escript), 31
Druck-Prager, 90
dump() (Data method), 35
eigenvalues() (in module esys.escript), 37
eigenvalues_and_eigenvectors() (in module esys.escript), 37
elevation() (Camera method), 58
Ellipsoid (class in ), 63
EllipsoidOnPlaneClip (class in ), 63
EllipsoidOnPlaneCut (class in ), 63
empty Data, 34, 35
Environment
ESCRIPT_HOSTFILE, 24
ESCRiPT_NUM_NODES, 24
ESCRiPT_NUM_PROCS, 24
ESCRiPT_NUM_THREADS, 24
ESCRiPT_STDFILES, 24, 25
MPI_COMM_WORLD, 41
OMP_NUM_THREADS, 3
PATH, 23, 24
esys.escript (extension module), 31
esys.escript.linearPDEs (extension module), 44
esys.pycad (extension module), 52
esys.pycad.gmsh (extension module), 54
esys.pyvisi (extension module), 57
exp() (in module esys.escript), 40
explicit scheme, 14
Courant condition, 14
finite element method, 2, 3
element, 3
mesh, 3
nodes, 2
flux, 45
force, internal, 83
Function() (in module esys.escript), 33
FunctionOnBoundary() (in module esys.escript), 33
FunctionOnContactOne() (in module esys.escript), 33
FunctionOnContactZero() (in module esys.escript), 33
FunctionSpace (class in esys.escript), 32
generalized minimal residual method
GMRES, 84
generateContours() (ContourModule method), 71
getAbsoluteTolerance() (StokesProblemCartesian method), 85
getCoefficient() (LinearPDE method), 45
getCommandString() (Design method), 55
getDim() (Design method), 54
getDim() (Domain method), 31
getDim() (FunctionSpace method), 32
getDim() (LinearPDE method), 45
getDim() (PropertySet method), 53
getDomain() (Data method), 35
getDomain() (FunctionSpace method), 32
getDomain() (LinearPDE method), 45
ggetElementOrder() (Design method), 54
ggetElementSize() (Design method), 54
getEscriptParamInt() (in module ), 41
getFlux() (LinearPDE method), 45
getFunctionSpace() (Data method), 35
getFunctionSpaceForCoefficient() (LinearPDE method), 45
getItems() (Design method), 55
getItems() (PropertySet method), 54
getManifoldClass() (PropertySet method), 53
getMeshFileName() (Design method), 55
getMeshHandler() (Design method), 55
getMPIRank() (Domain method), 32
getMPIRankWorld() (in module ), 41
getMPISize() (Domain method), 32
getMPISizeWorld() (in module ), 41
getMPIWorldMax() (in module ), 41
getName() (PropertySet method), 53
getDefault() (Domain method), 31
default() (FunctionSpace method), 32
default() (LinearPDE method), 45
default() (LinearPDE method), 45
getOperator() (LinearPDE method), 46
getRank() (Data method), 35
getRightHandSide() (LinearPDE method), 46
getScriptFileName() (Design method), 55
getScriptString() (Design method), 55
getShape() (Data method), 35
getShapeOfCoefficient() (LinearPDE method), 45
gPageSize() (Domain method), 31
gPageSize() (FunctionSpace method), 32
gPageSize() (LinearPDE method), 46
gPageSizeName() (LinearPDE method), 45
gPageSizePackage() (LinearPDE method), 45
gPageSizeLinearPDEPackage() (StokesProblemCartesian method), 85
gPageSizeLinearPDE() (Design method), 46
gPageSizeDomain() (Domain method), 32
gPageSizePropertySet() (PropertySet method), 54
gPageSizeTagMap() (Design method), 55
gPageSizeTolerance() (LinearPDE method), 45
gPageSizeTolerance() (StokesProblemCartesian method), 85
gPageSizeX() (Domain method), 31
gPageSizeX() (FunctionSpace method), 32
getGlobalPosition() (class in ), 70
GMRES, 47
GMRES (LinearPDE attribute), 47
Gmsh, 51
grad() (in module esys.escript), 39
GS (LinearPDE attribute), 47
Helmholtz (class in esys.escript.linearPDEs), 48
Helmholtz equation, 5, 7, 8
identityTensor() (in module esys.escript), 36
identityTensor4() (in module esys.escript), 36
ILU0 (LinearPDE attribute), 47
ILUT, 46, 47
ILUT (LinearPDE attribute), 47
Image (class in ), 67
imageList() (Movie method), 69
imageRange() (Movie method), 69
ImageReader (class in ), 59
implicit scheme, 14
incompressible fluid, 83
inf() (in module esys.escript), 36
initialize() (StokesProblemCartesian method), 84
inner() (in module esys.escript), 38
integrate() (in module esys.escript), 39
interpolate() (in module esys.escript), 39
inverse() (in module esys.escript), 37
isEmpty() (Data method), 35
isEmpty() (Operator method), 41
isSymmetric() (LinearPDE method), 45
isUsingLumping() (LinearPDE method), 45
isValidTagName() (Domain method), 32
ITERATIVE (LinearPDE attribute), 46
JACOBI (LinearPDE attribute), 47
Jacobi, 46
jump() (in module esys.escript), 40
keepFiles() (Design method), 54
kronecker() (in module esys.escript), 36
Kronecker symbol, 8, 11
L2() (in module esys.escript), 40
Lame (class in esys.escript.linearPDEs), 49
Lame coefficients, 11
Lame equation, 15
Laplace operator, 1, 2
leftView() (Camera method), 58
Legend (class in ), 67
length() (in module esys.escript), 37
Level Set, 89
LevelSet (class in ), 89
Light (class in ), 58
Line (class in esys.pycad), 52
linear solver
  BiCGStab, 46
  GMRES, 47
  lumping, 45, 47
  minimum fill-in ordering, 46
  MINRES, 47
  nested dissection ordering, 46
  PCG, 46, 47, 86
  TFQMR, 47
LinearPDE (class in esys.escript.linearPDEs), 44
listEscriptParams() (in module ), 41
load() (in module esys.escript), 35
LocalPosition (class in ), 70
log() (in module esys.escript), 40
log10() (in module esys.escript), 40
Logo (class in ), 68
Lsup() (in module esys.escript), 36
LUMPING (LinearPDE attribute), 47
lumping, 45, 47
makeMovie() (Movie method), 69
Map (class in ), 60
MapOnPlaneClip (class in ), 61
MapOnPlaneCut (class in ), 61
MapOnScalarClip (class in ), 61
MapOnScalarClipWithRotation (class in ), 61
Matrix Market, 41
matrix_mult() (in module esys.escript), 38
matrix_transposed_mult() (in module esys.escript), 38
maximum() (in module esys.escript), 38
maxval() (in module esys.escript), 36
mayavi, 5, 20
Message Passing Interface
  MPI, 23–25, 32, 41
minimum() (in module esys.escript), 38
minimum fill-in ordering, 46
MINIMUM_FILL_IN (LinearPDE attribute), 47
MINRES, 47
MINRES (LinearPDE attribute), 47
minval() (in module esys.escript), 36
MKL (LinearPDE attribute), 46
momentum equation, 14
Movie (class in ), 68
MPIBarrier() (Domain method), 32
MPIBarrierWorld() (in module ), 41
nested dissection, 46
NESTED_DISSECTION (LinearPDE attribute), 48
netCDF, 31
NETGEN (Design attribute), 54
Neumann boundary condition
  homogeneous, 2, 4
Newton-Raphson scheme, 91
NO_REORDERING (LinearPDE attribute), 47
nonsymmetric() (in module esys.escript), 37
of() (Operator method), 41
onMasterProcessor() (Domain method), 32
OpenMP
  threading, 23, 24
Operator (class in esys.escript), 41
outer() (in module esys.escript), 38
outer normal field, 7
partial derivative, 2
partial differential equation, 1, 27, 33
PDE, 1, 3
PASO (LinearPDE attribute), 46
PCG, 46, 47, 86
PCG (LinearPDE attribute), 47
PlaneSurface (class in esys.pycad), 52
Point (class in esys.pycad), 52
Poisson, 44
Poisson (class in esys.escript.linearPDEs), 48
Poisson equation, 1–3
preconditioned conjugate gradient method
  PCG, 84
preconditioner
  ILUT, 46, 47
Jacobi, 46
PRES20 (LinearPDE attribute), 47
PropertySet (class in esys.pycad), 53
randomOn() (MaskPoints method), 73
rank, 35
Rectangle (class in ), 67
ReducedSolution() (in module esys.escript), 33
Reflection (class in esys.pycad), 53
render() (Scene method), 58
rightView() (Camera method), 58
RILU (LinearPDE attribute), 47
Rotation (class in esys.pycad), 53
rotateX() (Transform method), 72
rotateY() (Transform method), 73
rotateZ() (Transform method), 73
RuledSurface (class in esys.pycad), 52
saddle point problem, 83
saddle point problems, 27
saveDX() (in module esys.escript), 36
saveMM() (Operator method), 41
saveVTK() (in module esys.escript), 36
Scalar() (in module esys.escript), 35
Scene (class in ), 58
Schur complement, 84
scripts
'diffusion.py', 10, 16
'helmholtz.py', 9, 85
'wave.py', 14
SCSL (LinearPDE attribute), 46
setAbsoluteTolerance() (DarcyFlow method), 87
setAbsoluteTolerance() (StokesProblem-Cartesian method), 85
setActiveScalar() (DataCollector method), 59
setActiveTensor() (DataCollector method), 59
setActiveVector() (DataCollector method), 59
setAngle() (Light method), 59
setAngle() (Rotation method), 75
setBackground() (Scene method), 58
setCenter() (CubeSource method), 74
setClipValue() (Clipper method), 71
setColor() (Actor3D method), 70
setColor() (Light method), 58
color() (Text2D method), 59
setData() (DataCollector method), 59
setDebugOff() (LinearPDE method), 45
setDebugOn() (LinearPDE method), 45
dim() (Design method), 54
dimension() (Design method), 54
dimension() (Design method), 54
dimension() (Design method), 54
dimension() (Design method), 54
dimension() (Design method), 54
dimension() (Design method), 54
dimension() (Design method), 54
setIntegrationToBothDirections() (StreamLineModule method), 72
setKeepFilesOff() (Design method), 54
setKeepFilesOn() (Design method), 54
setLabelColor() (ScalarBar method), 74
setMaxScaleFactor() (TensorGlyph method), 71
setMeshFileName() (Design method), 55
setName() (PropertySet method), 53
setOpacity() (Actor3D method), 70
setOptions() (Design method), 55
setOrientationToHorizontal() (ScalarBar method), 74
setOrientationToVertical() (ScalarBar method), 74
setOrigin() (PlaneSource method), 71
setPhiResolution() (Sphere method), 72
setPlaneToXY() (Transform method), 73
setPlaneToXZ() (Transform method), 73
setPlaneToYZ() (Transform method), 73
setPoint1() (PlaneSource method), 72
setPoint2() (PlaneSource method), 72
setPointSourceCenter() (PointSource method), 72
setPointSourceNumberOfPoints() (PointSource method), 72
setPointSourceRadius() (PointSource method), 72
setPosition() (Actor2D method), 70
setPosition() (Camera method), 58
setPosition() (Light method), 59
setPosition() (ScalarBar method), 74
setRatio() (MaskPoints method), 73
setReducedOrderOff() (LinearPDE method), 46
setReducedOrderOn() (LinearPDE method), 46
setRepresentationToWireframe() (Actor3D method), 70
setScaleFactor() (Glyph3D method), 71
setScaleFactor() (Warp method), 73
setScaleModeByScalar() (Glyph3D method), 71
setScaleModeByVector() (Glyph3D method), 71
setTagMap() (Domain method), 32
setTags() (FunctionSpace method), 32
setThetaResolution() (Sphere method), 72
setTitle() (ScalarBar method), 74
setTitleColor() (ScalarBar method), 74
setTolerance() (DarcyFlow method), 87
setTolerance() (StokesProblemCartesian method), 85
setToleranceReductionFactor() (LinearPDE method), 45
setTubeRadius() (Tube method), 73
setTubeRadiusToVaryByScalar() (Tube method), 73
setTubeRadiusToVaryByVector() (Tube method), 73
setValue() (DarcyFlow method), 87
setValue() (Helmholtz method), 48
setValue() (Lame method), 49
setValue() (LinearPDE method), 44
setValue() (Operator method), 41
setValue() (Poisson method), 48
setWidth() (ScalarBar method), 74
setX() (Domain method), 31
setXLength() (CubeSource method), 74
setYLength() (CubeSource method), 74
setZLength() (CubeSource method), 75
shape, 4, 28, 29, 31, 33–35
sign() (in module esys.escript), 40
sin() (in module esys.escript), 40
sinh() (in module esys.escript), 40
slicing, 34
Solution() (in module esys.escript), 33
solution, 33, 48, 49
reduced, 33
solve() (DarcyFlow method), 87
solve() (StokesProblemCartesian method), 84
solves() (Operator method), 41
Spline (class in esys.pycad), 52
sqrt() (in module esys.escript), 40
SSOR (LinearPDE attribute), 47
Stokes problem, 83, 84
StokesProblemCartesian (class in ), 84
StreamLine (class in ), 65
stress, 11
stress, initial, 83
summation convention, 2, 6
sup() (in module esys.escript), 36
SurfaceLoop (class in esys.pycad), 53
swap_axes() (in module esys.escript), 37
symmetric() (in module esys.escript), 37
symmetric PDE, 8, 16
symmetrical, 44
tag, 53
tan() (in module esys.escript), 40
tanh() (in module esys.escript), 40
Tensor () (in module esys.escript), 35
Tensor3() (in module esys.escript), 35
Tensor4() (in module esys.escript), 35
tensor_mult () (in module esys.escript), 39
tensor_transposed_mult() (in module esys.escript), 39
TETGEN (Design attribute), 54
Text2D (class in ), 59
TFQMR, 47
TFQMR (LinearPDE attribute), 47
time integration explicit, 14
implicit, 14
Tube topView() (Camera method), 58
trace() (in module esys.escript), 37
(Tube translate() (Transform method), 72
Translation (class in esys.pycad), 53
transpose() (in module esys.escript), 37
transposed_matrix_mult() (in module esys.escript), 38
transposed_tensor_mult() (in module esys.escript), 39
UMFPACK (LinearPDE attribute), 46
unitVector() (in module esys.escript), 36
update_parameter() (LevelSet method), 89
update_phi() (LevelSet method), 89
Uzawa scheme, 83
Vector() (in module esys.escript), 35
Velocity (class in ), 61
velocity, 83
VelocityOnPlaneClip (class in ), 63
VelocityOnPlaneCut (class in ), 62
Verlet scheme, 11
visualization mayavi, 5, 20
Volume (class in esys.pycad), 53
von–Mises stress, 16
wave equation, 11
whereNegative() (in module esys.escript), 40
whereNonNegative() (in module esys.escript), 41
whereNonPositive() (in module esys.escript), 41
whereNonZero() (in module esys.escript), 41
wherePositive() (in module esys.escript), 40
whereZero() (in module esys.escript), 41
yield condition, 90


