Abstract

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

- esys.escript core library
- finite element solver esys.finley (which uses fast vendor-supplied solvers or our paso linear solver library)
- the meshing interface esys.pycad
- a model library.

The current version supports parallelization through both MPI for distributed memory and OpenMP for distributed shared memory.

*Please see Chapter 2 for changes to the way to launch esys.escript scripts.* For more info on this and other changes from previous releases see Appendix A.2.

If you use this software in your research, then we would appreciate (but do not require) a citation. Some relevant references can be found in Appendix A.3.
# CONTENTS

1 Tutorial: Solving PDEs .......................................................... 1
   1.1 Installation ........................................................................ 1
   1.2 The First Steps ................................................................... 1
   1.3 The Diffusion Problem ...................................................... 8
   1.4 3-D Wave Propagation ...................................................... 13
   1.5 Elastic Deformation ........................................................... 19
   1.6 Stokes Flow ....................................................................... 22
   1.7 Slip on a Fault ................................................................... 24

2 Execution of an escript Script ..................................................... 31
   2.1 Overview ........................................................................... 31
   2.2 Options ............................................................................. 32
   2.3 Input and Output ............................................................... 33
   2.4 Hints for MPI Programming ............................................. 33
   2.5 Lazy Evaluation ................................................................ 34

3 The Module esys.escript .............................................................. 35
   3.1 esys.escript Classes .......................................................... 39
   3.2 Physical Units .................................................................... 51
   3.3 Utilities ............................................................................. 54

4 The Module esys.escript.linearPDEs ........................................... 57
   4.1 Linear Partial Differential Equations ................................. 57
   4.2 Projection .......................................................................... 61
   4.3 Solver Options .................................................................. 62

5 The Module esys.pycad ............................................................... 69
   5.1 Introduction ....................................................................... 69
   5.2 The Unit Square .................................................................. 69
   5.3 Holes ................................................................................. 71
   5.4 A 3D example ..................................................................... 73
   5.5 Alternative File Formats .................................................. 74
   5.6 Element Sizes .................................................................... 74
   5.7 esys.pycad Classes ............................................................ 75
   5.8 Interface to the mesh generation software ......................... 78

6 Models .................................................................................... 81
   6.1 The Stokes Problem .......................................................... 81
   6.2 Darcy Flux ......................................................................... 86
   6.3 Isotropic Kelvin Material .................................................... 88
   6.4 Fault System ...................................................................... 91

7 The Module esys.finley ............................................................... 97
   7.1 Formulation ....................................................................... 97
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.

\[ \text{Figure 1.1: Domain } \Omega = [0, 1]^2 \text{ with outer normal field } n. \]

\[ \Delta \] 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

$$x_i y_i = \sum_{i=0}^{2} x_i y_i$$

(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, ie $n_i u_{,i} = n_0 u_{,0} + n_1 u_{,1}$. 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

---

1You may be more familiar with the Laplace operator being written as $\nabla^2$, and written in the form

$$\nabla^2 u = \nabla \cdot \nabla u = \frac{\partial^2 u}{\partial x_0^2} + \frac{\partial^2 u}{\partial x_1^2}$$

and Equation (1.1) as

$$-\nabla^2 u = f$$

2Some 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 [38, 13].

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 run-escript command:

```
run-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

```
>>> 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

```
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

---

3 run-escript 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.
The arguments \( n0 \) and \( n1 \) 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 7.

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 \( \text{gammaD} \) 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 \text{getRank} \) and \text{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

\[
x0=x[0]
\]

Object \( x0 \) 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 \( \text{gammaD} \) 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])
```

whereZero \( (x[0]) \) creates function which equals 1 where \( x[0] \) is (almost) equal to zero and 0 elsewhere. Similarly, whereZero \( (x[1]) \) creates function which equals 1 where \( x[1] \) is equal to zero and 0 elsewhere. The sum of the results of whereZero \( (x[0]) \) and 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 \( \text{gammaD} \) has the same rank, shape and FunctionSpace like \( x0 \) 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 setValue method of the 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 Poisson class definition from the `esys.escript.linearPDEs` module. To get the solution of the Poisson equation defined by `mypde` we just have to call its `getSolution` method.

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()
```

The question is what we do with the calculated solution \( u \). Besides postprocessing, eg. calculating the gradient or the average value, which will be discussed later, plotting the solution is one thing you might want to do. `esys.escript` offers two ways to do this, both based on external modules or packages and so data need to be converted to hand over the solution. The first option is using the `matplotlib` module which allows plotting 2D results relatively quickly, see [21]. However, there are limitations when using this tool, eg. in problem size and when solving 3D problems. Therefore `esys.escript` provides a second option based on `VTK` files which is especially designed for large scale and 3D problems and which can be read by a variety of software packages such as `mayavi` [10], `VisIt` [34].

### 1.2.1 Plotting Using matplotlib

The `matplotlib` module provides a simple and easy to use way to visualize PDE solutions (or other `Data` objects). To hand over data from `esys.escript` to `matplotlib` the values need to be mapped onto a rectangular grid. We will make use of the `numpy` module.

First we need to create a rectangular grid. We use the following statements:

```python
import numpy
x_grid = numpy.linspace(0.,1.,50)
y_grid = numpy.linspace(0.,1.,50)
```

\( x_{\text{grid}} \) is an array defining the x coordinates of the grid while \( y_{\text{grid}} \) defines the y coordinates of the grid. In this case we use 50 points over the interval [0, 1] in both directions.

Now the values created by `esys.escript` need to be interpolated to this grid. We will use the `matplotlib.mlab.griddata` function to do this. We can easily extract spatial coordinates as a list by

---

4Users of Debian 5(Lenny) please note: this example makes use of the `griddata` method in `matplotlib.mlab`. This method is not part of version 0.98.1 which is available with Lenny. If you wish to use contour plots, you may need to install a later version. Users of Ubuntu 8.10 or later should be fine.

---

1.2. The First Steps
In principle we can apply the same `toListOfTuples` method to extract the values from the PDE solution \( u \). However, we have to make sure that the `Data` object we extract the values from uses the same `FunctionSpace` as we have us when extracting \( x \) and \( y \). We apply the interpolation to \( u \) before extraction to achieve this:

\[
z = \text{interpolate}(u, \text{mydomain.getX().getFunctionSpace()})
\]

The values in \( z \) are now the values at the points with the coordinates given by \( x \) and \( y \). These values are now interpolated to the grid defined by \( x\_\text{grid} \) and \( y\_\text{grid} \) by using

```python
import matplotlib
z\_grid = matplotlib.mlab.griddata(x,y,z,xi=x\_grid, yi=y\_grid)
```

\( z\_\text{grid} \) gives now the values of the PDE solution \( u \) at the grid. The values can be plotted now using the `contourf`:

```python
matplotlib.pyplot.contourf(x\_grid, y\_grid, z\_grid, 5)
matplotlib.pyplot.savefig("u.png")
```

Here we use 5 contours. The last statement writes the plot to the file ‘u.png’ in the PNG format. Alternatively, one can use

```python
matplotlib.pyplot.contourf(x\_grid, y\_grid, z\_grid, 5)
matplotlib.pyplot.show()
```

which gives an interactive browser window.

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
import numpy
import matplotlib
import pylab

# 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()

# interpolate \( u \) to a matplotlib grid:
x_grid = numpy.linspace(0.,1.,50)
y_grid = numpy.linspace(0.,1.,50)
x=mydomain.getX()[0].toListOfTuples()
y=mydomain.getX()[1].toListOfTuples()
z=interpolate(u,mydomain.getX().getFunctionSpace())
z_grid = matplotlib.mlab.griddata(x,y,z,xi=x_grid,yi=y_grid )

# interpolate \( u \) to a rectangular grid:
matplotlib.pyplot.contourf(x_grid, y_grid, z_grid, 5)
matplotlib.pyplot.savefig("u.png")

The entire code is available as ‘poisson_matplotlib.py’ in the example directory. You can run the script using the escript environment

    run-escript poisson_matplotlib.py

This will create the ‘u.png’, see Figure 1.3. For details on the usage of the matplotlib module we refer to the documentation [21].

As pointed out, matplotlib is restricted to the two-dimensional case and should be used for small problems only. It can not be used under MPI as the toListOfTuples method is not safe under MPI\(^5\).

### 1.2.2 Visualization using VTK

As an alternative escript supports the usage of visualization tools which base on VTK, eg. mayavi [10], VisIt [34]. In this case the solution is written to a file in the VTK format. This file the can read by the tool of choice. Using VTK file is MPI safe.

To write the solution \( u \) in Poisson problem to the file ‘u.xml’ one need to add the line

    saveVTK("u.xml", sol=u)

\(^5\)The phrase ‘safe under MPI’ means that a program will produce correct results when run on more than one processor under MPI.
The solution $u$ is now available in the ‘u.xml’ tagged with the name "sol".

The Poisson problem script is now

```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_VTK.py’ in the example directory.

You can run the script using the escript environment and visualize the solution using mayavi:

```
run-escript poisson_VTK.py
mayavi2 -d u.xml -m SurfaceMap
```

The result is shown in Figure 1.4.

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 1.5 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

$$ \rho c_p T_t - (\kappa T_x)_i = q_H $$  \hspace{1cm} (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:

$$ q_H(x,t) = \begin{cases} q^c & \text{if } \|x - x^c\| \leq r \\ 0 & \text{else} \end{cases} $$ \hspace{1cm} (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 $\kappa T_i$ to be proportional to the difference of the current temperature to the surrounding temperature $T_{\text{ref}}$:

$$ \kappa T_i n_i = \eta (T_{\text{ref}} - T) $$ \hspace{1cm} (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:

$$ T(x,0) = T_{\text{ref}} $$ \hspace{1cm} (1.18)

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 $t^{(n)}$:

$$ T^{(n)} \approx T^{(n-1)} + T^{(n)}_t (t^{(n)} - t^{(n-1)}) = T^{(n-1)} + h \cdot T^{(n)}_t $$ \hspace{1cm} (1.19)

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

$$ \frac{\rho c_p}{h} T^{(n)} - (\kappa T^{(n)}_x)_i = q_H + \frac{\rho c_p}{h} T^{(n-1)} $$ \hspace{1cm} (1.20)

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

$$ \kappa T^{(n)}_i n_i = \eta (T_{\text{ref}} - T^{(n)}) $$ \hspace{1cm} (1.21)

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

$$ \omega T^{(n)} - (\kappa T^{(n)}_x)_i = f $$ \hspace{1cm} (1.22)

and we set

$$ \omega = \frac{\rho c_p}{h} \text{ and } f = q_H + \frac{\rho c_p}{h} T^{(n-1)} $$ \hspace{1cm} (1.23)
With \( g = \eta T_{\text{ref}} \) the radiation condition defined by Equation (1.21) takes the form

\[
\kappa T^{(n)}_i n_i = g - \eta T^{(n)}_i \text{ on } \Gamma
\]  

(1.24)

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:

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

(1.25)

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 \( \Gamma \):

\[
n_j A_{ij} u_{,i} + d u = y .
\]  

(1.26)

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

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

(1.27)

\( r \) 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 \( T^{(n)}_i \) 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.\(^6\) 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:\(^7\):

```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,y=g)
mypde.setValue(d=2*eta) # overwrites d=eta
u=mypde.getSolution()
```

\(^6\)There 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.

\(^7\)Please, note that this is not a complete code. The complete code can be found in “helmholtz.py".
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
\[ A_{jl} = A_{lj} . \] (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 to check if the given PDE is symmetric.

\[
\text{mypde} = \text{LinearPDE(mydomain)} \\
\text{mypde.setValue}\left(A=\kappa \ast \text{kronecker(mydomain)}, d=\eta\right) \\
\text{print mypde.checkSymmetry()} \quad \#\ \text{returns True} \\
\text{mypde.setValue}\left(B=\text{kronecker(mydomain)}[0]\right) \\
\text{print mypde.checkSymmetry()} \quad \#\ \text{returns False} \\
\text{mypde.setValue}\left(C=\text{kronecker(mydomain)}[0]\right) \\
\text{print mypde.checkSymmetry()} \quad \#\ \text{returns True}
\]

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

\[
\text{mypde} = \text{LinearPDE(mydomain)} \\
\text{mypde.setValue}(A=\kappa \ast \text{kronecker(mydomain)}) \\
\text{mypde.setSymmetryOn()}
\]

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 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 \( \kappa n_i u_i = \kappa n_0 \). 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:

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

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

\[
\text{mayavi -d u.xml -m SurfaceMap &}
\]

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

1.3. The Diffusion Problem
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:
    mypde.setValue(Y=q+rhocp/h*T)
    T=mypde.getSolution()
    t+=h
```

$kappa$, $rhocp$, $eta$ and $Tref$ 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:

$$
\|y\| = \sqrt{\sum_{i} y_i^2} = \text{esys.escript.length}(y)
$$

(1.30)

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 $length(x-xc)\cdot r$, 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 ...
```

12 Chapter 1. Tutorial: Solving PDEs
Figure 1.6: Results of the Temperature Diffusion Problem for Time Steps 16, 32 and 48.

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 1.6 shows the result for some selected time steps. An easy way to visualize the results is the command

```
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:

$$\rho u_{i,tt} - \sigma_{ij,j} = 0$$  \hspace{1cm} (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 $t > 0$.

Here we are modelling a point source at the point $x_C$ in the $x_0$-direction which is a negative pulse of amplitude $U_0$ followed by the same positive pulse. In mathematical terms we use

$$u_0(x_C, t) = U_0 \sqrt{\frac{t - t_0}{\alpha}} e^{\frac{1}{2} \left( \frac{t - t_0}{\alpha} \right)^2}$$

(1.34)

for all $t \geq 0$ where $\alpha$ is the width of the pulse and $t_0$ is the time when the pulse changes from negative to positive. In the simulations we will choose $\alpha = 0.3$ and $t_0 = 2$, see Figure 1.7 and will apply the source as a constraint in a in a sphere of small radius around the point $x_C$. 

**Figure 1.7:** Input Displacement at Source Point ($\alpha = 0.7$, $t_0 = 3$, $U_0 = 1$).

**Figure 1.8:** Input Acceleration at Source Point ($\alpha = 0.7$, $t_0 = 3$, $U_0 = 1$).
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

\[
\begin{align*}
\mathbf{u}^{(n)} &= 2\mathbf{u}^{(n-1)} - \mathbf{u}^{(n-2)} + h^2 \mathbf{a}^{(n)} \\
\mathbf{a}^{(n)} &= 2 \mathbf{a}^{(n-1)} - \mathbf{a}^{(n-2)} + h^2 \mathbf{u}^{(n)}
\end{align*}
\]

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

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

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

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

\[
\rho \mathbf{a}^{(n)}_i = \sigma^{(n-1)}_{ij,j}
\]

and boundary conditions

\[
\sigma^{(n-1)}_{ij} n_j = 0
\]

derived from Equation (1.33) where

\[
\sigma^{(n-1)}_{ij} = \lambda \mathbf{u}^{(n-1)}_{kk} \delta_{ij} + \mu (\mathbf{u}^{(n-1)}_{i,j} + \mathbf{u}^{(n-1)}_{j,i}).
\]

We also need to apply the constraint

\[
a^{(n)}_0(x_c, t) = U_0 \sqrt{\frac{2}{\alpha^2}} \left( \frac{(t - t_0)^3}{\alpha^3} - \frac{(t - t_0)^4}{\alpha^4} \right) e^{\frac{t - (t_0)^2}{\alpha^2}}
\]

which is derived from equation 1.34 by calculating the second order time derivative, see Figure 1.8. Now we have converted our problem for displacement, \( \mathbf{u}^{(n)} \), into a problem for acceleration, \( \mathbf{a}^{(n)} \), which now depends on the solution at the previous two time steps, \( \mathbf{u}^{(n-1)} \) and \( \mathbf{u}^{(n-2)} \).

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

\[
D_{ij} \mathbf{a}^{(n)}_j = -X_{ij,j}.
\]

The natural boundary condition

\[
n_j X_{ij} = 0
\]

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

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

Moreover we need to define the location \( r \) where the constraint 1.41 is applied. We will apply the constraint on a small sphere of radius \( R \) around \( x_C \) (we will use 3p.c. of the width of the domain):

\[
q_i(x) = \begin{cases} 
1 & \|x - x_c\| \leq R \\
0 & \text{otherwise}
\end{cases}
\]

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

```python
def wavePropagation(domain, h, tend, lam, mu, rho, x_c, src_radius, U0):
    # lists to collect displacement at point source which is returned to the caller
    ts, u_pc0, u_pc1, u_pc2 = [], [], [], []
    x = domain.getX()
    # ... open new PDE ...
    mypde = LinearPDE(domain)
```

1.4. 3-D Wave Propagation 15
mypde.getSolverOptions().setSolverMethod(mypde.getSolverOptions().LUMPING)
kron=identity(mypde.getDim())
dunit=numpy.array([1.,0.,0.]) # defines direction of point source
mypde.setValue(D=kron=rho, q=whereNegative(length(x-xc)-src_radius)*dunit)
# ... set initial values ....
n=0
# for first two time steps
u=Vector(0.,Solution(domain))
u_last=Vector(0.,Solution(domain))
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
# open file to save displacement at point source
u_pc_data=FileWriter('./data/U_pc.out')
ts.append(t); u_pc0.append(u_pc[0]), u_pc1.append(u_pc[1]), u_pc2.append(u_pc[2])
while t<tend:
    t+=h
    # ... get current stress ....
g=grad(u)
stress=lam*trace(g)*kron+mu*(g+transpose(g))
    # ... get new acceleration ....
amplitude=U0*(4*(t-t0)**3/alpha**3-6*(t-t0)/alpha)*sqrt(2.)/alpha**2 \
        *exp(1./2.-((t-t0)**2/alpha**2)
    mypde.setValue(X=-stress, r=dunit*amplitude)
a=mypde.getSolution()
    # ... get new displacement ...
u_new=2*u-u_last+h**2*a
    # ... shift displacements ....
u_last=u
    u=u_new
    n+=1
print n,"-th time step t ",t
u_pc=L.getValue(u)
print "u at point charge=",u_pc
# save displacements at point source to file for t > 0
if n==1 or n%10==0: saveVTK("./data/usoln.%i.vtu"%(n/10),
    acceleration=length(a)/9.81,
    displacement = length(u), \
    tensor = stress, Ux = u[0] )
    return ts, u_pc0, u_pc1, u_pc2

Notice that all coefficients of the PDE which are independent of time \( t \) are set outside the while loop. This is very efficient since it allows the LinearPDE class to reuse information as much as possible when iterating over time.

The statement

```
mypde.getSolverOptions().setSolverMethod(mypde.getSolverOptions().LUMPING)
```

switches on the use of an aggressive approximation of the PDE operator as a diagonal matrix formed from the coefficient \( D \). The approximation allows, at the cost of additional error, very fast solution of the PDE. When using lumping the presence of \( A, B \) or \( C \) will produce wrong results.

There are a few new esys.escript functions in this example: grad(\( u \)) returns the gradient \( u_{i,j} \) of \( u \) (in fact \( \text{grad}(g)[i,j]=u_{i,j} \)). There are restrictions on the argument of the grad function, for instance the statement

```
    grad(u)
```

returns the gradient \( u_{i,j} \) of \( u \) (in fact \( \text{grad}(g)[i,j]=u_{i,j} \)). There are restrictions on the argument of the grad function, for instance the statement

```
    grad(u)
```
\( \text{grad} (\text{grad} (u)) \) will raise an exception. \( \text{trace} (g) \) returns the sum of the main diagonal elements \( g[k,k] \) of \( g \) and \( \text{transpose} (g) \) returns the matrix transpose of \( g \) (ie. \( \text{transpose}(g)[i,j] = g[j,i] \)).

We initialize the values of \( u \) and \( u\_\text{last} \) to be zero. It is important to initialize both with the solution FunctionSpace FunctionSpace as they have to be seen as solutions of PDEs from previous time steps. In fact, the \( \text{grad} \) does not accept arguments with a certain FunctionSpace, for more details see Section 3.1.3.

The Locator is designed to extract values at a given location (in this case \( x_C \)) from functions such as the displacement vector \( u \). Typically the Locator is used in the following form:

\[
L = \text{Locator}(\text{domain}, xc) \\
u = ... \\
u\_pc = L.\text{getValue}(u)
\]

The return value \( u\_pc \) is the value of \( u \) at the location \( xc \). The values are collected in the lists \( u\_pc0, u\_pc1 \) and \( u\_pc2 \) together with the corresponding time marker in \( ts \). The values are handed back to the caller. Later we will show to ways to access these data.

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 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 i.e. 10 and 11). The \( \text{ne} \) 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 \( \text{ne} \) 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. \( \text{Brick}(n_0, n_1, n_2, l_0, l_1, l_2) \) is an esys.finley function which creates a rectangular mesh with \( n_0 \times n_1 \times n_2 \) elements over the brick \([0, l_0] \times [0, l_1] \times [0, l_2]\).

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 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 i.e. 10 and 11). The \( \text{ne} \) 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 \( \text{ne} \) 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. \( \text{Brick}(n_0, n_1, n_2, l_0, l_1, l_2) \) is an esys.finley function which creates a rectangular mesh with \( n_0 \times n_1 \times n_2 \) elements over the brick \([0, l_0] \times [0, l_1] \times [0, l_2]\).

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.
Figure 1.9: Selected time steps \((n = 11, 22, 32, 36)\) of a wave propagation over a \(10\text{km} \times 10\text{km} \times 3.125\text{km}\) block from a point source initially at \((5\text{km}, 5\text{km}, 0)\) with time step size \(h = 0.02083\). Color represents the displacement. Here the view is oriented onto the bottom face.

```
    print "time step size = ",h
    ts, u_pc0, u_pc1, u_pc2 = \
        wavePropagation(mydomain,h,tend,lam,mu,rho,xc, src_radius, U0)
```

The `domain.getSize()` return the local element size \(\Delta x\). The `inf` makes sure that the Courant condition \(1.46\) olds everywhere in the domain.

The script is available as ‘wave.py’ in the example directory. To visualize the results from the data directory:

```
    mayavi2 -d usoln.1.vtu -m SurfaceMap &
```

You can rotate this figure by clicking on it with the mouse and moving it around. Again use Configure Data to move backwards and forward in time, and also to choose the results (acceleration, displacement or \(u_z\)) by using Select Scalar. Figure 1.9 shows the results for the displacement at various time steps.

It remains to show some possibilities to inspect the collected data \(u_{pc0}, u_{pc1}\) and \(u_{pc2}\). One way is to write the data to a file and then use an external package such as `gnuplot` [36], excel or OpenOffice to read the data for further analysis. The following code shows one possible form to write the data to the file ‘./data/U_pc.out’:

```
    u_pc_data=FileWriter('./data/U_pc.out')
    for i in xrange(len(ts)) :
        u_pc_data.write("%f %f %f
\(ts[i],u_{pc0}[i],u_{pc1}[i],u_{pc2}[i])\)
    u_pc_data.close()
```

The `U_pc.out` stores 4 columns of data: \(t, u_x, u_y, u_z\) respectively, where \(u_x, u_y, u_z\) are the \(x_0, x_1, x_2\) components of the displacement vector \(u\) at the point source. These can be plotted easily using any plotting package. In `gnuplot` [36]the command:

```
    plot 'U_pc.out' u 1:2 title 'U_x' w l lw 2, 'U_pc.out' u 1:3 title 'U_y' w l lw 2, 'U_pc.out' u 1:4 title 'U_z' w l lw 2
```
will reproduce Figure 1.10 (As expected this is identical to the input signal shown in Figure 1.7)2. It is pointed out that we are not using the standard python open to write to the file U_pc.out as it is not safe when running esys.escript under MPI, see chapter 2 for more details.

Alternatively, one can implement plotting the results at run time rather than in a post-processing step. This avoids the generation of an intermediate data file. In escript the preferred way of creating 2D plots of time dependent data is matplotlib. The following script creates the plot and writes it into the file ‘u_pc.png’ in the PNG image format:

```python
import matplotlib.pyplot as plt
if getMPIRankWorld() == 0:
    plt.title("Displacement at Point Source")
    plt.plot(ts, u_pc0, '-', label="x_0", linewidth=1)
    plt.plot(ts, u_pc1, '-', label="x_1", linewidth=1)
    plt.plot(ts, u_pc2, '-', label="x_2", linewidth=1)
    plt.xlabel('time')
    plt.ylabel('displacement')
    plt.legend()
    plt.savefig('u_pc.png', format='png')
```

You can add the plt.show() to create a interactive browser window. Please not that through the getMPIRankWorld() == 0 statement the plot is generated on one processor only (in this case the rank 0 processor) when run under MPI.

Both options for processing the point source data are include in the example file ‘wave.py’. There other options available to process these data in particular through the SciPy[15] package, eg Fourier transformations. It is beyond the scope of this users guide to document the usage of SciPy[15] for time series analysis but is highly recommended that users use SciPy[15] to any further data processing.

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:

\[
- \sigma_{ij,j} = 0
\]  

(1.47)

where the stress \( \sigma \) is given by

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

(1.48)

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 (1.47) is similar to Equation (1.31) 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 (1.48) an extra term is introduced to bring in stress due to volume changes trough temperature dependent expansion.

Our domain is the unit cube

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

(1.49)

On the boundary the normal stress component is set to zero

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

(1.50)

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

\[
u_i(x) = 0 \quad \text{where} \quad x_i = 0
\]  

(1.51)

For the temperature distribution we use

\[
T(x) = T_0 e^{-\beta \|x-x^c\|};
\]  

(1.52)

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

When we insert Equation (1.48) 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

\[
-(A_{ijkl} u_{k,l})_{,j} = -X_{ij,j}.
\]  

(1.53)

\( A \) 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:

\[
n_j A_{ijkl} u_{k,l} = n_j X_{ij}.
\]  

(1.54)

Constraints take the form

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

(1.55)

\( r \) and \( q \) are each rank-1 Data object. We can easily identify the coefficients in Equation (1.53):

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

(1.56)

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

(1.57)

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

\[
q_i(x) = \begin{cases} 
1 & x_i = 0 \\
0 & \text{otherwise}
\end{cases}
\]  

(1.59)

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 (1.48) to zero.
Analogously to the concept of symmetry for a single PDE, we call the PDE defined by Equation (1.53) symmetric if
\[ A_{ijkl} = A_{klij} \] (1.60)

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 analyze the actual stress distribution. Typically the von–Mises stress defined by
\[
\sigma_{\text{mises}} = \sqrt{\frac{1}{2}((\sigma_{00} - \sigma_{11})^2 + (\sigma_{11} - \sigma_{22})^2 + (\sigma_{22} - \sigma_{00})^2) + 3(\sigma_{01}^2 + \sigma_{12}^2 + \sigma_{20}^2)}
\] (1.62)
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:

```python
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)*kron(mydomain)
mypde.setValue(A=C,X=sigma0,q=msk)
#... solve pde ...
u=mypde.getSolution()
#... calculate von-Misses stress
sigma=mu*(grad(u)+transpose(grad(u)))+lam*trace(grad(u))*kron(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)/2.+
    3*(sigma[0,1]**2 + sigma[1,2]**2 + sigma[2,0]**2))
#... output ...
saveVTK("deform.xml",disp=u,stress=sigma_mises)
```

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

1.5. Elastic Deformation 21
Note that the filter CellToPointData is applied to create smooth representation of the von-Mises stress. Figure 1.11 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:

\[
\nabla \cdot \left( \eta (\nabla \vec{v} + \nabla^T \vec{v}) \right) - \nabla p = -f,
\]

with the incompressibility condition

\[
\nabla \cdot \vec{v} = 0.
\]

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):

\[
-(\eta(v_{i,j} + v_{j,i}))*_j - p_i = f_i,
\]

with the incompressibility condition

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

The subscript comma \( i \) denotes the derivative of the function with respect to \( x_i \). The body force \( f \) in Equation (1.65) is the gravity acting in the \( x_3 \) direction and is given as \( f = -g \rho \delta_{3} \). The Stokes equations is a saddle point problem, and can be solved using a Uzawa scheme. A class called StokesProblemCartesian in esys.escript can be used to solve for velocity and pressure; more detail on the class can be view in Chapter 6. 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:

\[
C = \frac{v \delta t}{h}.
\]

where \( \delta t, v, \) 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. We are using $20 \times 20$ elements with piecewise linear elements for the pressure and for velocity but the element is subdivided for the velocity. This approach is called *macro elements* and needs to be applied to make sure that the discretised problem has a unique solution, see [20] for details. The fact that pressure and velocity are represented in different ways is expressed by

```python
velocity=Vector(0.0, Solution(mesh))
pressure=Scalar(0.0, ReducedSolution(mesh))
```

The gravitational force is calculated based on the fluid density and the acceleration due to gravity. The boundary conditions are set for a slip condition at the base and the left face of the domain: At the base fluid movement in the $x_0$-direction is free, but fixed in the $x_1$-direction and similar at the left face fluid movement in the $x_1$-direction is free, but fixed in the $x_0$-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 based 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 saved 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
eta=1.0
rho=100.0
g=10.0

tolerance=1.0e-4
max_iter=200
t_end=50
t=0.0
time=0
verbose=True

#define mesh
H=2.0
L=1.0
W=1.0
mesh = esys.finley.Rectangle(l0=L, l1=H, order=-1, n0=20, n1=20)
coordinates = mesh.getX()

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

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

#boundary conditions for slip at base
boundary_cond=whereZero(coordinates[1])*[0.0,1.0]+whereZero(coordinates[0])*[1.0,0.0]

#velocity and pressure vectors
velocity=Vector(0.0, Solution(mesh))
```

9Alternative, one can use second order elements for the velocity and first order for pressure on the same element. You may use `order=2` in `esys.finley.Rectangle`
pressure=Scalar(0.0, ReducedSolution(mesh))

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

while t <= t_end:
    print " ----- Time step = %s -----
    print "Time = %s seconds"%( t , time )
    solution.initialize(fixed_u_mask=boundary_cond,eta=eta,f=Y)
    velocity,pressure=solution.solve(velocity,pressure,max_iter=max_iter, \
    verbose=verbose)

    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 1.12 and 1.13 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.

1.7 Slip on a Fault

In this example we illustrate how to calculate the stress distribution around a fault in the Earth’s crust caused by a slip through an earthquake.

To simplify the presentation we assume a simple domain \( \Omega = [0, 1]^2 \) with a vertical fault in its center. We assume that the slip distribution \( s_i \) on the fault is known. We want to calculate the distribution of the displacements \( u_i \) and stress \( \sigma_{ij} \) in the domain. We assume an isotropic, linear elastic material model of the form

\[
\sigma_{ij} = \lambda u_{k,k} \delta_{ij} + \mu (u_{i,j} + u_{j,i}) \tag{1.68}
\]

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

\[
\sigma_{ij} n_j = 0 \tag{1.69}
\]
and normal displacements are set to zero:

\[ u_i n_i = 0 \]  

(1.70)

The stress needs to fulfill the momentum equation

\[ -\sigma_{ij,j} = 0 \]  

(1.71)

This problem is very similar to the elastic deformation problem presented in Section 1.5.

However, we need to address an additional challenge: The displacement \( u_i \) is in fact discontinuous across the fault but we are in the lucky situation that we know the jump of the displacements across the fault. This is in fact the given slip \( s_i \). So we can split the total distribution \( u_i \) into a component \( v_i \) which is continuous across the fault and the known slip \( s_i \)

\[ u_i = v_i + \frac{1}{2} s_i^\pm \]  

(1.72)

where \( s^\pm = s \) when right of the fault and \( s^\pm = -s \) when left of the fault. We assume that \( s^\pm = 0 \) sufficiently away from the fault.

We insert this into the stress definition 1.68

\[ \sigma_{ij} = \sigma_{ij}^c + \frac{1}{2} \sigma_{ij}^s \]  

(1.73)
with

\[ \sigma_{ij}^c = \lambda v_{k,k} \delta_{ij} + \mu (v_{i,j} + v_{j,i}) \] (1.74)

and

\[ \sigma_{ij}^s = \lambda s_{k,k}^\pm \delta_{ij} + \mu (s_{i,j}^\pm + s_{j,i}^\pm) \] (1.75)

In fact \( \sigma_{ij}^s \) defines a stress jump across the fault. In easy way to construct this function is to us a function \( \chi \) which is 1 on the right and \(-1\) on the left side from the fault. One can then set

\[ \sigma_{ij}^s = \chi \cdot (\lambda s_{k,k}^\pm \delta_{ij} + \mu (s_{i,j}^\pm + s_{j,i}^\pm)) \] (1.76)

assuming that \( s \) is extended by zero away from the fault. After inserting 1.73 into 1.71 we get the differential equation

\[ -\sigma_{ij,j}^c = \frac{1}{2} \sigma_{ij,j}^s \] (1.77)

Together with the definition 1.74 we have a differential equation for the continuous function \( vi \). Notice that the boundary condition 1.70 and 1.69 transfer to \( vi \) and \( \sigma_{ij}^c \) as \( s \) is zero away from the fault. In Section 1.5 we have discussed how this problem is solved using the LinearPDE class. We refer to this section for further details.

To define the fault we use the FaultSystem class introduced in Section 6.4. The following statements define a fault system \( fs \) and add the fault \( I \) to the system.

```python
fs = FaultSystem(dim=2)
fs.addFault(fs.addFault(V0=[0.5, 0.25], strikes=90*DEG, ls=0.5, tag=1)
```

Figure 1.13: Simulation output for Stokes flow.
The fault added starts at point \((0.5, 0.25)\) has length 0.5 and is pointing north. The main purpose of the \texttt{FaultSystem} class is to define a parameterization of the fault using a local coordinate system. One can inquire the class to get the range used to parameterize a fault.

\[
p_0, p_1 = \texttt{fs.getW0Range(tag=1)}
\]

Typically \(p_0\) is equal to zero while \(p_1\) is equal to the length of the fault. The parameterization is given as a mapping from a set of local coordinates onto parameter range (in our case the range \(p_0\) to \(p_1\)). For instance to map the entire domain \texttt{mydomain} onto the fault one can use

\[
x = \texttt{mydomain.getX()}
p, m = \texttt{fs.getParametrization(x, tag=1)}
\]

Of course there is the problem that not all location are on the fault. For those locations which are on the fault \(m\) is set to 1 otherwise 0 is used. So on return the values of \(p\) are defining the value of the fault parameterization (typically the distance from the starting point of the fault along the fault) where \(m\) is positive. On all other locations the value if \(p\) is undefined. Now \(p\) can be used to define a slip distribution on the fault via

\[
s = m \times (p - p_0) \times (p_1 - p)/(p_1 - p_0)/2)**2 \times \text{slip}_{\text{max}} \times [0., 1.]
\]

Notice that the factor \(m\) which makes sure that \(s\) is zero away from the fault. It is important that the slip is zero at the ends of the faults.

We can now put all components together to get the script:

```python
from esys.escript import *
from esys.escript.linearPDEs import LinearPDE
from esys.escript.models import FaultSystem
from esys.finley import import Rectangle
from esys.escript.unitsSI import DEG

#... set some parameters ...
lam=1.
mu=1
slip_max=1.
mydomain = Rectangle(l0=1.,l1=1.,n0=16, n1=16) # n1 need to be multiple of 4!!!
# .. create the fault system
```

1.7. Slip on a Fault
FIGURE 1.15: Total Displacement after slip event.

```
fs=FaultSystem(dim=2)
fs.addFault(V0=[0.5,0.25], strikes=90*DEG, ls=0.5, tag=1)
# ... create a slip distribution on the fault:
p, m=fs.getParametrization(mydomain.getX(),tag=1)
p0,p1= fs.getW0Range(tag=1)
s=m*(p-p0)*(p1-p)/((p1-p0)/2)**2*slip_max*[0.,1.]
# ... calculate stress according to slip:
D=symmetric(grad(s))
chi, d=fs.getSideAndDistance(D.getFunctionSpace().getX(),tag=1)
sigma_s=(mu*D+lam*trace(D)*kronecker(mydomain))*chi
#... open symmetric PDE ...
mpde=LinearPDE(mydomain)
mpde.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
# ... fix displacement in normal direction
x=mydomain.getX()
msk=whereZero(x[0])*[1.,0.0] + whereZero(x[0]-1.)*[1.,0.0] 
   +whereZero(x[1])*[0.,1.0] + whereZero(x[1]-1.)*[0.,0.1]
mpde.setValue(A=C,X=-0.5*sigma_s,q=msk)
#... solve pde ...
mpde.getSolverOptions().setVerbosityOn()
v=mpde.getSolution()
# .. write the displacement to file:
D=symmetric(grad(v))
sigma=(mu*D+lam*trace(D)*kronecker(mydomain))+0.5*sigma_s
saveVTK("slip.vtu",disp=v+0.5*chi*s, stress= sigma)
```
The script creates the 'slip.vtu' giving the total displacements and stress. These values are stored as cell centered data. See Figure 1.15 for the result.
CHAPTER

TWO

Execution of an escript Script

2.1 Overview

A typical way of starting your escript script ‘myscript.py’ is with the run-escript command. This command was renamed from escript (used in previous releases) to avoid clashing with an unrelated program installed by default on some systems. Most 3.1 releases of esys.escript allow either run-escript or escript to be used but the latter name will be removed in future releases:

```
escript myscript.py
```

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

```
run-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 run-escript using four threads (eg. if you use a multi-core processor) you can use

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

This will require escript to be compiled for OpenMP [27].

To start run-escript using MPI [23] with 8 processes you use

```
run-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

```
run-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. For example, to use 8 nodes, with 4 MPI processes per node, write

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

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

---

1 The run-escript launcher is not supported under MS Windows yet.
2 ie. not MS Windows or Ubuntu 9.10
3 For this discussion, it is assumed that run-escript is included in your PATH environment. See installation guide for details.
4 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.
This runs the script on 8 nodes, with 4 processes per node and 2 threads per process.

2.2 Options

The general form of the run-escript launcher is as follows:

```
```

where file is the name of a script, ARGS are arguments for the script. The run-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 processes being used is nn \cdot 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 \cdot 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 process. 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 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 than one process (nn \cdot 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 to set up the esys.escript environment. This option is useful if users wish to execute scripts without using the run-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 run-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 than the master process \((\text{rank}=0)\) will not be available. Error output from any processor will be redirected to the node where run-escript has been invoked. If the `-o` or ESCRIPT_STDFILES is set\(^5\), 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 be 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 run-escript is invoked.

2.4 Hints for MPI Programming

In general a script based on the esys.escript module does not require modifications to run 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\(^6\) while data exchange is performed through the esys.escript module.

This has three main implications:

1. most arguments \((\text{Data excluded})\) should the same values on all processors. eg int, float, str and numpy parameters.
2. the same operations will be called on all processors.
3. different processors may store different amounts of information.

With a few exceptions\(^7\), values of types int, float, str and numpy returned by esys.escript will have the same value on all processors. If values produced by other modules are used as arguments, the user has to make sure that the argument values are identical on all processors. For instance, the usage of a random number generator to create argument values bears the risk that the value may depend on the processor.

Some operations in esys.escript require communication with all processors executing the job. It is not always obvious which operations these are. For example, Lsup returns the largest value on all processors. getValue on Locator may refer to a value stored on another processor. For this reason it is better if scripts do not have conditional operations (which manipulate data) based on which processor the script is on. Crashing or hanging scripts can be an indication that this has happened.

It is not always possible to divide data evenly amongst processors. In fact some processors might not have any data at all. Try to avoid writing scripts which iterate over data points, instead try to describe the operation you wish to perform as a whole.

Special attention is required when using files on more than one processor as several processors access the file at the same time. Opening a file for reading is safe, however the user has to make sure that the variables which are set from reading data from files are identical on all processors.

When writing data to a file it is important that only one processor is writing to the file at any time. As all values in esys.escript are global it is sufficient to write values on the processor with MPI rank 0 only. The FileWriter class provides a convenient way to write global data to a simple file. The following script writes to the file `test.txt` on the processor with id 0 only:

```python
from esys.escript import *
f = FileWriter('test.txt')
```

\(^{5}\)That is, it has a non-empty value.

\(^{6}\)In case of OpenMP only one copy is running but esys.escript temporarily spawns threads.

\(^{7}\)getTupleForDataPoint
f.write('test message')
f.close()

We strongly recommend using this class rather than the built-in `open` function as it will guarantee a script which will run in single processor mode as well as under MPI.

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

### 2.5 Lazy Evaluation

Escript now supports lazy evaluation [18]. Lazy evaluation is when expressions are not evaluated until it is actually needed. When applied to suitable problems, it can reduce both the memory and cpu time required to perform a simulation. This implementation is designed to be as transparent as possible; so significant alterations to scripts are not required.

**How to use it**

To have lazy evaluation applied automatically, put the following command in your script after the imports.

```python
from esys.escript import setEscriptParamInt
setEscriptParamInt('AUTOLAZY',1)
```

To get greater benefit, some fine tuning may be required. If your simulation involves iterating for a number of timesteps, you will probably have some state variables which are updated in each iteration based on their value in the previous iteration. For example:

```python
x=f(x_previous)
y=g(x)
z=h(y,x,
```

Could be modified to:

```python
x=f(x_previous)
resolve(x)
y=g(x)
z=h(y,x,
```

The `resolve` command forces `x` to be evaluated immediately.

**When to use it**

We believe that problems involving large domains and complicated expressions will benefit most from lazy evaluation. In cases where lazy does provide a benefit, larger domains should provide larger benefit. If you are uncertain, try running a test on a smaller domain.
The Module \texttt{esys.escript}

\texttt{esys.escript} is a Python module that allows you to represent the values of a function at points in a \texttt{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 \texttt{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 \texttt{FunctionSpace} object. The following statement generates the object \texttt{solution_space} which is a \texttt{FunctionSpace} object and provides access to the function space of PDE solutions on the \texttt{Domain mydomain}:

\begin{verbatim}
    solution_space=Solution(mydomain)
\end{verbatim}

The following generators for function spaces on a \texttt{Domain mydomain} are commonly used:

- \texttt{Solution(mydomain)}: solutions of a PDE.
- \texttt{ReducedSolution(mydomain)}: solutions of a PDE with a reduced smoothness requirement, eg. using a lower order approximation on the same element or using macro elements.
- \texttt{ContinuousFunction(mydomain)}: continuous functions, eg. a temperature distribution.
- \texttt{Function(mydomain)}: general functions which are not necessarily continuous, eg. a stress field.
- \texttt{FunctionOnBoundary(mydomain)}: functions on the boundary of the domain, eg. a surface pressure.
- \texttt{FunctionOnContact0(mydomain)}: functions on side 0 of the discontinuity.
- \texttt{FunctionOnContact1(mydomain)}: functions on side 1 of the discontinuity.

In some cases under-integration is used. For these cases the user may use a \texttt{FunctionSpace} from the following list:

- \texttt{ReducedFunction(mydomain)}
- \texttt{ReducedFunctionOnBoundary(mydomain)}
- \texttt{ReducedFunctionOnContact0(mydomain)}
- \texttt{ReducedFunctionOnContact1(mydomain)}

In comparison to the corresponding full version they use a reduced number of integration nodes (typically one only) to represent values.
The reduced smoothness for PDE solution is often used to fulfill the Ladyzhenskaya-Babuska-Brezzi condition [20] when solving saddle point problems, e.g., 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 esys.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, e.g., 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. The user needs to be aware that an interpolation is not always possible, see Figure 3.1 for esys.finley. An alternative approach to change the representation (FunctionSpace) is projection, see Section 4.2.

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.
index. The rank in \texttt{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 \texttt{Data} object is a tuple (list) \texttt{s} of integers. The length of \texttt{s} is the rank of the \texttt{Data} object and the \texttt{i}-th index ranges between 0 and \texttt{s[i]} \text{-} 1. For instance, a stress field has rank 2 and shape \texttt{(d,d)} where \texttt{d} is the spatial dimension. The following statement creates the \texttt{Data} object \texttt{mydat} representing a continuous function with values of shape \texttt{(2,3)} and rank 2:

\begin{verbatim}
mydat=Data(value=1,what=ContinuousFunction(myDomain),shape=(2,3))
\end{verbatim}

The initial value is the constant 1 for all data sample points and all components. \texttt{Data} objects can also be created from any \texttt{numpy} array or any object, such as a list of floating point numbers, that can be converted into a \texttt{numpy.ndarray} [14]. The following two statements create objects which are equivalent to \texttt{mydat}:

\begin{verbatim}
mydat1=Data(value=numpy.ones((2,3)),what=ContinuousFunction(myDomain))
mydat2=Data(value=[[1,1],[1,1],[1,1]],what=ContinuousFunction(myDomain))
\end{verbatim}

In the first case the initial value is \texttt{numpy.ones((2,3))} which generates a \texttt{2 \times 3} matrix as a \texttt{numpy.ndarray} filled with ones. The shape of the created \texttt{Data} object it 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 \texttt{numpy.ndarray} before creating the actual \texttt{Data} object.

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

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

Here the initial value is 0 but any object that can be converted into a \texttt{numpy.ndarray} and whose shape is consistent with shape of the \texttt{Data} object to be created can be used as the initial value.

\texttt{Data} objects can be manipulated by applying unary operations (e.g. \texttt{cos}, \texttt{sin}, \texttt{log}) point and can be combined point-wise by applying arithmetic operations (e.g. \texttt{+}, \texttt{-}, \texttt{*}, \texttt{/}). It is to be emphasized that \texttt{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 \texttt{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 \texttt{Data} objects: Assume we have a displacement field \texttt{u} and we want to calculate the corresponding stress field \texttt{σ} using the linear–elastic isotropic material model

\begin{equation}
σ_{ij} = λu_{k,k}δ_{ij} + µ(u_{i,j} + u_{j,i})
\end{equation}

where \texttt{δ}_{ij} is the Kronecker symbol and \texttt{λ} and \texttt{µ} are the Lame coefficients. The following function takes the displacement \texttt{u} and the Lame coefficients \texttt{lam} and \texttt{mu} as arguments and returns the corresponding stress:

\begin{verbatim}
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
\end{verbatim}

The variable \texttt{d} gives the spatial dimension of the domain on which the displacements are defined. \texttt{kronecker} returns the Kronecker symbol with indexes \texttt{i} and \texttt{j} running from 0 to \texttt{d}-1. The call \texttt{grad(u)} requires the displacement field \texttt{u} to be in the \texttt{Solution} or continuous \texttt{FunctionSpace} function space. The result \texttt{g} as well as the returned stress will be in the general \texttt{FunctionSpace} function space. If, for example, \texttt{u} is the solution of a PDE then \texttt{getStress} might be called in the following way:

\begin{verbatim}
mydat=Dat
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 white and 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 white and 2 for rock type 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 lam:

```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 lam as a function of the general FunctionSpace function space. In the context of FEM this means the nodal values of lam are interpolated to the element centers. The interpolation is automatic and requires no special handling.

The following statements show how, for the example of Figure 3.2, the stress calculation discussed above and tagged values are used for lam:
lam=Scalar(value=2.,what=Function(mydomain))
insertTaggedValue(lam, white=30., gray=5000.)
s=getStress(u, lam, 2.)

In this example lam is set to 30 for those cells with tag white (=1) and to 5000 for those cells with tag gray (=2). The initial value 2 of lam is used as a default value for the case when a tag is encountered which has not been linked with a value. The getStress method does not need to be changed now that we are using tags. esys.escript resolves the tags when lam*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.escript 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 [25]. 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

\[
x=\text{ContinuousFunction}(\text{mydomain}).\text{getX()}
x.\text{dump}(\text{"x.nc"})
\]

mydomain.\text{dump(\text{"dom.nc"})}

To recover the object \(x\) and \(\text{mydomain}\) was a esys.finley mesh use

\[
\text{from esys.finley import LoadMesh}
\]

\[
\text{mydomain=LoadMesh(\text{"dom.nc"})}
\]

\[
x=\text{load(\text{"x.nc"}, \text{mydomain})}
\]

It possible to rerun the mechanism that was originally used to generates \(\text{mydomain}\) to recreate \(\text{mydomain}\). However in most cases using dump and load is faster in particular if optimization has been applied. In case that esys.escript is running on more than one MPI processor the dump will create an individual file for each processor containing the local data. In order to avoid conflicts the file name is extended by the MPI processor rank.

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.escript Classes

#### 3.1.1 Domain class

**class Domain()**

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.

dump (filename) dumps the Domain into the file filename.

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.

getTag (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 hyper-manifold (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 repre-

sent 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 <code>arg0</code> and <code>arg1</code></td>
</tr>
<tr>
<td>arg0 * arg1</td>
<td>multiplies <code>arg0</code> and <code>arg1</code></td>
</tr>
<tr>
<td>arg0 - arg1</td>
<td>difference <code>arg1</code> from <code>arg0</code></td>
</tr>
<tr>
<td>arg0 / arg1</td>
<td>divide <code>arg0</code> by <code>arg1</code></td>
</tr>
<tr>
<td>arg0 ** arg1</td>
<td>raises <code>arg0</code> to the power of <code>arg1</code></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 numpy 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>arg0 += arg2</td>
<td>adds <code>arg0</code> to <code>arg2</code></td>
</tr>
<tr>
<td>arg0 *= arg2</td>
<td>multiplies <code>arg0</code> with <code>arg2</code></td>
</tr>
<tr>
<td>arg0 -= arg2</td>
<td>subtracts <code>arg2</code> from <code>arg0</code></td>
</tr>
<tr>
<td>arg0 /= arg2</td>
<td>divides <code>arg0</code> by <code>arg2</code></td>
</tr>
<tr>
<td>arg0 **= arg2</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[10:u0]</code></td>
<td><code>(u0-10,)</code></td>
</tr>
<tr>
<td>2</td>
<td><code>arg[10:u0,11:u1]</code></td>
<td><code>(u0-10,u1-11)</code></td>
</tr>
<tr>
<td>3</td>
<td><code>arg[10:u0,11:u1,12:u2]</code></td>
<td><code>(u0-10,u1-11,u2-12)</code></td>
</tr>
<tr>
<td>4</td>
<td><code>arg[10:u0,11:u1,12:u2,13:u3]</code></td>
<td><code>(u0-10,u1-11,u2-12,u3-13)</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:
3.1.4 Generation of Data objects

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

Scalar (value=0., what=FunctionSpace(), expand=False)
Vector (value=0., what=FunctionSpace(), expand=False)
Tensor (value=0., what=FunctionSpace(), expand=False)
Tensor3 (value=0., what=FunctionSpace(), expand=False)
Tensor4 (value=0., what=FunctionSpace(), expand=False)

load (filename, domain)

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://esys.ssc.uq.edu.au/docs.html.

getFunctionSpace ()
getDomain ()
getShape ()

returns the FunctionSpace of the object.
returns the Domain of the object.
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 numpy.ndarray or must be convertible into a numpy.ndarray object. value (or the corresponding numpy.ndarray object) must be of rank 0 or must have the same rank like the object. If a value has already been 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(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.

3.1.6 Functions of Data objects

This section lists the most important functions for Data class objects a. A complete list and a more detailed description of the functionality can be found on http://esys.esscc.uq.edu.au/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 [26] 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

\[
kron(d)_{i,j} = \begin{cases} 
1 & \text{if } i = j \\
0 & \text{otherwise}
\end{cases}
\]  (3.2)

If d is an integer a (d, d) numpy 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

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

If d is an integer a (d, d, d, d) numpy array is returned.
unitVector \((i,d)\) returns a rank-1 Data object in FunctionSpace \(d\) such that
\[\text{identityTensor}(d)[j] = \begin{cases} 1 & \text{if } j = i \\ 0 & \text{otherwise} \end{cases}\] (3.4)

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

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

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

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

\(\text{minval}(a)\) returns at each data sample points the minimum value over all components.

\(\text{maxval}(a)\) returns at each data sample points the maximum value over all components.

\(\text{length}(a)\) 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)

\(\text{trace}(a[],\text{axis\_offset}=0)\) returns the trace of \(a\). This is the sum over components \(axis\_offset\) and \(axis\_offset+1\) 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 \(axis\_offset=1\) this is
\[\text{trace}(a,1)[i,j] = \sum_k a[i,k,k,j]\] (3.7)

\(\text{transpose}(a[],\text{axis\_offset}=\text{None})\) returns the transpose of \(a\). This swaps the first \(axis\_offset\) components of \(a\) with the rest. If \(axis\_offset\) is not present \(\text{int}(r/2)\) is used where \(r\) is the rank of \(a\), the sum over components \(axis\_offset\) and \(axis\_offset+1\) 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 \(axis\_offset=1\) this is
\[\text{transpose}(a,1)[i,j,k,l] = a[j,k,l,i]\] (3.9)

\(\text{swap\_axes}(a[],\text{axis0}=0,\text{axis1}=1)\) returns \(a\) but with swapped components \(axis0\) and \(axis1\). The argument \(a\) must be at least of rank-2 Data object. For instance in the for a rank-4 Data object argument, \(axis0=1\) and \(axis1=2\) this is
\[\text{swap\_axes}(a,1,2)[i,j,k,l] = a[i,k,j,l]\] (3.10)

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

\(\text{nonsymmetric}(a)\) returns the non–symmetric part of \(a\). This is \((a-\text{transpose}(a))/2\).
The current implementation is restricted to arguments of shape (2, 2) and (3, 3).

eigenvalues (a)
return the eigenvalues of a. This is
matrix_mult(a,V)=e[i]*V
(3.12)
where e=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, i.e. a=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
matrix_mult(a,V[:,i])=e[i]*V[:,i]
(3.13)
where e,V=eigenvalues_and_eigenvectors(a). The eigenvectors V are orthogonal and normalized, i.e.
matrix_mult(transpose(V),V)=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, i.e. a=symmetric(a). The current implementation is restricted to arguments of shape (2, 2) and (3, 3).

maximum (*a)
returns the maximum value over all arguments at all data sample points and for each component. For instance
maximum(a0,a1) [i,j] = max(a0[i,j],a1 [i,j])
(3.15)
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
minimum(a0,a1) [i,j] = min(a0[i,j],a1 [i,j])
(3.16)
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
inner(a) = ∑^ij a0[j,i] · a1 [j,i]
(3.17)
and for a rank-4 Data object arguments this is
inner(a) = ∑^ijkl a0[i,j,k,l] · a1 [j,i,k,l]
(3.18)

matrix_mult (a0,a1)
returns the matrix product of a0 and a1. If a1 is rank-1 Data object this is
matrix_mult(a) [i] = ∑_k a0 · [i,k] a1 [k]
(3.19)
and if a1 is rank-2 Data object this is
matrix_mult(a) [i,j] = ∑_k a0 · [i,k] a1 [k,j]
(3.20)
transposed_matrix.mult(a0,a1)
returns the matrix product of the transposed of a0 and a1. The function is equivalent to
matrix.mult(transpose(a0), a1). If a1 is rank-1 Data object this is
\[
\text{transposed_matrix.mult}(a)[i] = \sum_k a[i,k] a1[k]
\] (3.21)
and if a1 is rank-2 Data object this is
\[
\text{transposed_matrix.mult}(a)[i,j] = \sum_k a[i, k] a1[k, j]
\] (3.22)

matrix.transposed_mult(a0,a1)
returns the matrix product of a0 and the transposed of a1. The function is equivalent to
matrix.mult(a0, transpose(a1)). If a1 is rank-2 Data object this is
\[
\text{matrix.transposed_mult}(a)[i,j] = \sum_k a[i, k] a1[j, k]
\] (3.23)

outer(a0,a1)
returns the outer product of a0 and a1. For instance if a0 and a1 both are rank-1 Data object then
\[
\text{outer}(a)[i,j] = a0[i] \cdot a1[j]
\] (3.24)
and if a0 is rank-1 Data object and a1 is rank-3 Data object
\[
\text{outer}(a)[i,j,k] = a0[i] \cdot a1[j,k]
\] (3.25)

tensor.mult(a0,a1)
returns the tensor product of a0 and a1. If a1 is rank-2 Data object this is
\[
\text{tensor.mult}(a)[i,j] = \sum_{kl} a0[i, j, k, l] \cdot a1[k, l]
\] (3.26)
and if a1 is rank-4 Data object this is
\[
\text{tensor_mult}(a)[i,j,m,n] = \sum_{kl} a0[i, j, m, n] \cdot a1[k, l]
\] (3.27)

transposed.tensor.mult(a0,a1)
returns the tensor product of the transposed of a0 and a1. The function is equivalent to
tensor.mult(transpose(a0), a1). If a1 is rank-2 Data object this is
\[
\text{transposed.tensor.mult}(a)[i,j] = \sum_{kl} a0[k, l, i, j] \cdot a1[k, l]
\] (3.28)
and if a1 is rank-4 Data object this is
\[
\text{transposed.tensor.mult}(a)[i,j,m,n] = \sum_{mn} a0[i, j, m, n] \cdot a1[k, l]
\] (3.29)

tensor.transposed_mult(a0,a1)
returns the tensor product of a0 and the transposed of a1. The function is equivalent to
tensor.mult(a0, transpose(a1)). If a1 is rank-2 Data object this is
\[
\text{tensor.transposed_mult}(a)[i,j] = \sum_{kl} a0[i, j, k, l] \cdot a1[l, k]
\] (3.30)
and if a1 is rank-4 Data object this is
\[
\text{tensor.transposed_mult}(a)[i,j,m,n] = \sum_{mn} a0[i, j, m, n] \cdot a1[k, l, m, n]
\] (3.31)
\textbf{grad}(a[, \text{where}=\text{None}])

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

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

(3.32)

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

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

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

(3.34)

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

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

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

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

returns the divergence of \(a\). This

\[
\text{div}(a) = \text{trace} (\text{grad}(a), \text{where})
\]  

(3.35)

\textbf{jump}(a[, \text{domain}=\text{None}])

returns the jump of \(a\) over the discontinuity in its domain or if \text{Domain} \text{domain} is present in \text{domain}.

\[
\text{jump}(a) = \text{interpolate}(a, \text{FunctionOnContactOne}(\text{domain})) - \text{interpolate}(a, \text{FunctionOnContactZero}(\text{domain}))
\]  

(3.36)

\textbf{L2}(a)

returns the \(L^2\)-norm of \(a\) in its function space. This is

\[
\text{L2}(a) = \text{integrate}(\text{length}(a)^2)
\]  

(3.37)

The following functions operate “point-wise”. That is, the operation is applied to each component of each point individually.

\textbf{sin}(a)

applies sine function to \(a\).

\textbf{cos}(a)

applies cosine function to \(a\).

\textbf{tan}(a)

applies tangent function to \(a\).

\textbf{asin}(a)

applies arc (inverse) sine function to \(a\).

\textbf{acos}(a)

applies arc (inverse) cosine function to \(a\).

\textbf{atan}(a)

applies arc (inverse) tangent function to \(a\).

\textbf{sinh}(a)

applies hyperbolic sine function to \(a\).
\( \cosh(a) \)

applies hyperbolic cosine function to \( a \).

\( \tanh(a) \)

applies hyperbolic tangent function to \( a \).

\( \text{asinh}(a) \)

applies arc (inverse) hyperbolic sine function to \( a \).

\( \text{acosh}(a) \)

applies arc (inverse) hyperbolic cosine function to \( a \).

\( \text{atanh}(a) \)

applies arc (inverse) hyperbolic tangent function to \( a \).

\( \exp(a) \)

applies exponential function to \( a \).

\( \sqrt{a} \)

applies square root function to \( a \).

\( \log(a) \)

applies the natural logarithm to \( a \).

\( \log10(a) \)

applies the base-10 logarithm to \( a \).

\( \text{sign}(a) \)

applies the sign function to \( a \), that is \( 1 \) where \( a \) is positive, \( -1 \) where \( a \) is negative and 0 otherwise.

\( \text{wherePositive}(a) \)

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

\( \text{whereNegative}(a) \)

returns a function which is 1 where \( a \) is negative and 0 otherwise.

\( \text{whereNonNegative}(a) \)

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

\( \text{whereNonPositive}(a) \)

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

\( \text{whereZero}(a, [\text{tol=None}, \text{rtol=1.e-8}]) \)

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 \( Ca \) times \( Crtol \) is used.

\( \text{whereNonZero}(a, [\text{tol=None}, \text{rtol=1.e-8}]) \)

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 \( Ca \) times \( Crtol \) is used.

### 3.1.7 Interpolating Data

In some cases, it may be useful to produce Data objects which fit some user defined function. Manually modifying each value in the Data object is not a good idea since it depends on knowing the location and order of each datapoint in the domain. Instead, \texttt{esys.escript} can use an interpolation table to produce a Data object.

The following example is available as ‘\texttt{int\_save.py}’ in the examples directory. We will produce a Data object which approximates a sine curve.

```python
from esys.escript import saveDataCSV, sup
import numpy
from esys.finley import Rectangle

n=4
r=Rectangle(n,n)
x=r.getX()
x0=x[0]
```
First we produce an interpolation table.

\[
\text{sine_table} = [0, 0.70710678118654746, 1, 0.70710678118654746, 0, -0.70710678118654746, -1, -0.70710678118654746, 0]
\]

We wish to identify 0 and 1 with the ends of the curve. That is, with the first and eighth values in the table.

\[
\text{numslices} = \text{len(sine_table)} - 1
\]

\[
\text{minval} = 0
\]

\[
\text{maxval} = 1
\]

\[
\text{step} = \frac{\text{maxval} - \text{minval}}{\text{numslices}}
\]

So the values \( v \) from the input lie in the interval \( \text{minval} \leq v < \text{maxval} \). \( \text{step} \) represents the gap (in the input range) between entries in the table. By default values of \( v \) outside the table argument range (\( \text{minval}, \text{maxval} \)) will be pushed back into the range, i.e. if \( v < \text{minval} \) the value \( \text{minval} \) will be used to evaluate the table. Similarly, for values \( v > \text{maxval} \) the value \( \text{maxval} \) is used.

Now we produce our new Data object.

\[
\text{result} = \text{x0.interpolateTable(sine_table, minval, step, toobig)}
\]

Any values which interpolate to larger than \( \text{toobig} \) will raise an exception. You can switch on boundary checking by adding “check_boundaries=True” the argument list.

Now for a 2D example. We will interpolate a surface such that the bottom edge is the sine curve described above. The amplitude of the curve decreases as we move towards the top edge.

Our interpolation table will have three rows.

\[
\text{st} = \text{numpy.array(sine_table)}
\]

\[
\text{table} = [\text{st}, 0.5 \times \text{st}, 0 \times \text{st}]
\]

The use of numpy and multiplication here is just to save typing.

\[
\text{result2} = \text{x1.interpolateTable(table, 0, 0.55, x0, minval, step, toobig)}
\]

In the 2D case, the parameters for the x1 direction (\( \text{min}=0, \text{step}=0.55 \)) come first followed by the x0 data object and its parameters. By default, if a point is specified which is outside the boundary, then \text{interpolateTable} will operate as if the point was on the boundary. Passing \text{check_boundaries=True} will \text{interpolateTable} to reject any points outside the boundaries.

### 3.1.8 Saving Data as CSV

For simple post-processing, Data objects can be saved in comma separated value format. If \( \text{mydata1} \) and \( \text{mydata2} \) are scalar data, the following command:

\[
\text{saveDataCSV('output.csv', U=mydata1, V=mydata2)}
\]

will record the values of \( \text{mydata} \) in \text{output.csv} in the following format:

\[
\begin{align*}
U, & V \\
1.00000000e+0, & 2.00000000e+1 \\
5.00000000e-0, & 1.00000000e+1 \\
\ldots
\end{align*}
\]
The names of the keyword parameters form the names of columns in the output. If the data objects are over different function spaces, then saveDataCSV will attempt to interpolate to a common function space. If this is not possible, then an exception will be raised.

Output can be restricted using a scalar mask.

```
saveDataCSV('outfile.csv', U=mydata1, V=mydata2, mask=myscalar)
```

Will only output those rows which correspond to to positive values of myscalar. Some aspects of the output can be tuned using additional parameters.

```
saveDataCSV('data.csv', append=True, sep=' ', csep='/', mask=mymask, e=mat1)
```

- `append` - specifies that the output should be written to the end of an existing file.
- `sep` - defines the separator between fields.
- `csep` - defines the separator between components in the header line. For example between the components of a matrix.

The above command would produce output like this:

```
e/0/0 e/1/0 e/0/1 e/1/1
1.0000000000e+00 2.0000000000e+00 3.0000000000e+00 4.0000000000e+00
...
```

Note that while the order in which rows are output can vary, all the elements in a given row always correspond to the same input.

### 3.1.9 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.

```
class Operator():
    creates an empty Operator object.

isEmpty (fileName)
    returns True if the object is empty. Otherwise True is returned.

setValue (value)
    resets all entries in the object representation to value

solves (rhs)
    solves the operator equation with right hand side rhs

of (u)
    applies the operator to the Data object u

saveMM (fileName)
    saves the object to a matrix market format file of name fileName, see http://maths.nist.gov/MatrixMarket
```

### 3.2 Physical Units

`esys.escript` provides support for physical units in the SI system including unit conversion. So the user can define variables in the form
from esys.escript.unitsSI import *
l=20*m
w=30*kg
w2=40*lb
T=100*Celsius

In the two latter cases an conversion from pounds and degree Celsius is performed into the appropriate SI units kg and Kelvin is performed. In addition composed units can be used, for instance

from esys.escript.unitsSI import *
rho=40*lb/cm**3

to define the density in the units of pounds per cubic centimeter. The value 40 will be converted into SI units, in this case kg per cubic meter. Moreover unit prefixes are supported:

from esys.escript.unitsSI import *
p=40*Mega*Pa

to the the pressure to 40 Mega Pascal. Units can also be converted back from the SI system into a desired unit, e.g

from esys.escript.unitsSI import *
print p/atm

can be used print the pressure in units of atmosphere.

This is an incomplete list of supported physical units:

- km unit of kilo meter
- m unit of meter
- cm unit of centi meter
- mm unit of milli meter
- sec unit of second
- minute unit of minute
- h unit of hour
- day unit of day
- yr unit of year
- gram unit of gram
- kg unit of kilo gram
- lb unit of pound
- ton metric ton
- A unit of Ampere
Hz
unit of Hertz
N
unit of Newton
Pa
unit of Pascal
atm
unit of atmosphere
J
unit of Joule
W
unit of Watt
C
unit of Coulomb
V
unit of Volt
F
unit of Farad
Ohm
unit of Ohm
K
unit of Kelvin
Celsius
unit of Celsius
Fahrenheit
unit of Fahrenheit

Moreover unit prefixes are supported:

Yotta
prefix yotta = $10^{24}$.
Zetta
prefix zetta= $10^{21}$.
Exa
prefix exa= $10^{18}$.
Peta
prefix peta= $10^{15}$.
Tera
prefix tera= $10^{12}$.
Giga
prefix giga= $10^{9}$.
Mega
prefix mega= $10^{6}$.
Kilo
prefix kilo= $10^{3}$.
Hecto
prefix hecto= $10^{2}$.
Deca
prefix deca= $10^{1}$. 

3.2. Physical Units 53
Deci
prefix deci = \(10^{-1}\).

Centi
prefix centi = \(10^{-2}\).

Milli
prefix milli = \(10^{-3}\).

Micro
prefix micro = \(10^{-6}\).

Nano
prefix nano = \(10^{-9}\).

Pico
prefix pico = \(10^{-12}\).

Femto
prefix femto = \(10^{-15}\).

Atto
prefix atto = \(10^{-18}\).

Zepto
prefix zepto = \(10^{-21}\).

Yocto
prefix yocto = \(10^{-24}\).

3.3 Utilities

The FileWriter provides a mechanism to write data to a file. In essence, this class wraps the standard file class to write data that are global in MPI to a file. In fact, data are written on the processor with MPI rank 0 only. It is recommended to use FileWriter rather than open in order to write code that is running with and without MPI. It is save to use open under MPI to read data which are global under MPI.

class FileWriter (fn[, append=False, [createLocalFiles=False]])

Opens a file of name fn for writing. If append is set to True written data are append at the end of the file. If running under MPI only the first processor with rank==0 will open the file and write to it. If createLocalFiles is set each individual processor will create a file where for any processor with rank>0 the file name is extended by its rank. This option is normally used for debug purposes only.

The following methods are available:

**close()**

closes the file.

**flush()**

flushes the internal buffer to disk.

**write(txt)**

Write string txt to file. Note that newline is not added.

**writelines(txts)**

Write the list txts of strings to the file. Note that newlines are not added. This method is equivalent to call write() for each string.

**closed**

True if file is closed.

**mode**

access mode.

**name**

file name.
newlines
    line separator

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

getEscriptParamInt (name)
    returns the current value of integer parameter name.

listEscriptParams (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 (a)
    returns the maximum value of the integer a across all processors within MPI_COMM_WORLD.
The Module `esys.escript.linearPDEs`

4.1 Linear Partial Differential Equations

The `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 `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:

$$-(A_{jl}u_{,l})_{,j} - (B_{j}u)_{,j} + C_{l}u_{,l} + D_{l}u = -X_{j,j} + Y.$$ (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 `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, $B$, $C$ and $X$ are rank-1 `Data` objects and $D$ and $Y$ are scalar. The following natural boundary conditions are considered on $\Gamma$:

$$n_{j}(A_{jl}u_{,l} + B_{j}u) + du = n_{j}X_{j} + y.$$ (4.2)

Notice that the coefficients $A$, $B$ and $X$ are defined in the PDE. The coefficients $d$ and $y$ are each a scalar `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

$$u = r \text{ where } q > 0$$ (4.3)

$r$ and $q$ are each scalar `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

$$-(A_{ijkl}u_{k,l})_{,j} - (B_{jik}u_{k})_{,j} + C_{ikl}u_{k,l} + D_{ik}u_{k} = -X_{ij,j} + Y_{i}.$$ (4.4)

$A$ is a rank-4 `Data` object, $B$ and $C$ are each a rank-3 `Data` object, $D$ and $X$ are each a rank-2 `Data` object and $Y$ is a rank-1 `Data` object. The natural boundary conditions take the form:

$$n_{j}(A_{ijkl}u_{k,l} + B_{jik}u_{k}) + d_{ik}u_{k} = n_{j}X_{ij} + y_{i}.$$ (4.5)

The coefficient $d$ is a rank-2 `Data` object and $y$ is a rank-1 `Data` object both in the boundary `FunctionSpace`. Constraints take the form

$$u_{i} = r_{i} \text{ where } q_{i} > 0$$ (4.6)

$r$ and $q$ are each rank-1 `Data` object. Notice that not necessarily all components must have a constraint at all locations.

`LinearPDE` also supports solution discontinuities over contact region $\Gamma^{\text{contact}}$ in the domain $\Omega$. To specify the conditions across the discontinuity we are using the generalised flux $J^{1}$ which is in the case of a systems of PDEs

---

1In some applications the definition of flux used here can be different from the commonly used definition. For instance, if $T$ is a temperature field the heat flux $q$ is defined as $q_{i} = -\kappa T_{,i}$ ($\kappa$ is diffusivity) which differs from the definition used here by the sign. This needs to be kept in mind when defining natural boundary conditions.
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
\[ J_j = A_{jl} u_{l} + B_{j} u_k - X_{j} \] (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
\[ n_j J^0_{ij} = n_j J^1_{ij} = y_{i}^{contact} - d_{ik}^{contact}[u]_k. \] (4.9)
where \( J^0 \) and \( J^1 \) are the fluxes on side 0 and side 1 of the discontinuity \( \Gamma^{contact} \), respectively. \([u]_i \) which is the difference of the solution at side 1 and at side 0, denotes the jump of \( u \) across \( \Gamma^{contact} \). The coefficient \( d_{ik}^{contact} \) is a rank-2 Data object and \( y_{i}^{contact} \) 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
\[ n_j J^0_j = n_j J^1_j = y_{j}^{contact} - d_{kj}^{contact}[u] \] (4.10)
In this case the the coefficient \( d_{kj}^{contact} \) and \( y_{j}^{contact} \) are each scalar Data object both in the contact FunctionSpace on side 0 or contact FunctionSpace on side 1.

The PDE is symmetrical if
\[ A_{jl} = A_{lj} \text{ and } B_{j} = C_{j} \] (4.11)
The system of PDEs is symmetrical if
\[
\begin{align*}
A_{ijkl} &= A_{klij} \\
B_{ijk} &= C_{kij} \\
D_{ik} &= D_{ki} \\
d_{ik} &= d_{ki} \\
d_{ik}^{contact} &= d_{ki}^{contact}
\end{align*}
\] (4.12-4.16)
Note that in contrast with the scalar case Equation (4.11) now the coefficients \( D, d \) and \( d^{contact} \) have to be inspected.

The following example illustrates the typical usage of the LinearPDE class:
```
from esys.escript import *
from esys.escript.linearPDEs import LinearPDE
from esys.finley import Rectangle
mydomain = Rectangle(l0=1.,l1=1.,n0=40, n1=20)
mypde=LinearPDE(mydomain)
mypde.setSymmetryOn()
mypde.setValue(A=kappa*kronecker(mydomain),D=1,Y=1)
u=mypde.getSolution()
```

We refer to chapter 1 for more details.

An instance of the SolverOptions class is attached to the LinearPDE class object. It is used to set options of the solver used to solve the PDE. In the following code the getSolverOptions is used to access the SolverOptions attached to mypde:
```
from esys.escript import *
from esys.escript.linearPDEs import LinearPDE, SolverOptions
from esys.finley import Rectangle
mydomain = Rectangle(l0=1.,l1=1.,n0=40, n1=20)
mypde=LinearPDE(mydomain)
mypde.setValue(A=kappa*kronecker(mydomain),D=1,Y=1)
mypde.getSolverOptions().setVerbosityOn()
mypde.getSolverOptions().setSolverMethod(SolverOptions.PCG)
mypde.getSolverOptions().setPreconditioner(SolverOptions.AMG)
mypde.getSolverOptions().setTolerance(1e-8)
mypde.getSolverOptions().setIterMax(1000)
u=mypde.getSolution()
```
In this code the preconditioned conjugate gradient method `SolverOptions.PCG` with preconditioner `SolverOptions.AMG`. The relative tolerance is set to $10^{-8}$ and the maximum number of iteration steps to 1000.

Moreover, after a completed solution call the attached `SolverOptions` object gives access to diagnostic informations:

```python
u = mypde.getSolution()
print 'Number of iteration steps =', mypde.getDiagnostics('num_iter')
print 'Total solution time =', mypde.getDiagnostics('time')
print 'Set-up time =', mypde.getDiagnostics('set_up_time')
print 'Net time =', mypde.getDiagnostics('net_time')
print 'Residual norm of returned solution =', mypde.getDiagnostics('residual_norm')
```

Typically a negative value for a diagnostic value indicates that the value is undefined.

### 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, eg. `esys.finley`. The general interface is provided through the `LinearPDE` class. 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://esys.esscc.uq.edu.au/docs.html](http://esys.esscc.uq.edu.au/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`, `d`, `y`, `d_contact`, `y_contact`, `q`, `r`])
  - assigns new values to coefficients. By default all values are assumed to be zero. If the new coefficient value is not a `Data` object, it is converted into a `Data` object in the appropriate `FunctionSpace`.

- `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.

- `getSolverOptions` ()
  - returns the solver options for solving the PDE. In fact the method returns a `SolverOptions` class object which can be used to modify the tolerance, the solver or the preconditioner, see Section 4.3 for details.

---

2In 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.
setSolverOptions(options=None)
sets the solver options for solving the PDE. If argument options is present it must be a SolverOptions class object, see Section 4.3 for details. Otherwise the solver options are reset to the default.

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

domain
returns the Domain of the PDE.

dim
returns the spatial dimension of the PDE.

getNumEquations()
returns the number of equations.

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

checkSymmetry(verbos=True)
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.

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()
returns (an approximation of) the solution of the PDE. This call will invoke the discretization of the PDE and the solution of the resulting system of linear equations. Keep in mind that this call is typically computational expensive and can - depending on the PDE and the discretization - take a long time to complete.

4.1.3 The Poisson Class

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

$$-u_{,ii} = f.$$  \hfill (4.17)

with homogeneous boundary conditions

$$n_i u_{,i} = 0$$  \hfill (4.18)
and homogeneous constraints
\[ u = 0 \text{ where } q > 0 \] (4.19)

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

**class Poisson (domain)**

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

**setValue** (\( f=\text{escript.Data()}, q=\text{escript.Data()} \))

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

### 4.1.4 The Helmholtz Class

The Helmholtz class defines the Helmholtz problem
\[ \omega u - (k u_{j,j}) = f \] (4.20)

with natural boundary conditions
\[ k u_{j,j}n_j = g - \alpha u \] (4.21)

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

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

**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:
\[ -\left( \mu (u_{i,j} + u_{j,i}) + \lambda u_{k,k} \delta_{ij} \right) = F_i - \sigma_{ij,j} \] (4.23)

with natural boundary conditions:
\[ n_j \left( \mu (u_{i,j} + u_{j,i}) + \lambda u_{k,k} \delta_{ij} \right) = f_i + n_j \sigma_{ij} \] (4.24)

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

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

**class Lame (domain)**

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

**setValue** (\( \mu, \lambda, F, \sigma \))

assigns new values to \( \mu, \lambda, F, \sigma \). By default all values are set to be zero.

### 4.2 Projection

4.2. Projection
Using the LinearPDE class provides an option to change the FunctionSpace attribute in addition to the standard interpolation mechanism as discussed on in Chapter 3. If one looks the stripped down version

\[ u = Y \]  

(4.26)

of the general scalar PDE 4.1 (boundary conditions are irrelevant) one can see the solution \( u \) of this PDE as a project of the input function \( Y \) which has the general FunctionSpace attribute to a function with the solution FunctionSpace or reduced solution FunctionSpace attribute. In fact, the solution maps values defined at element centers representing a possibly discontinuous function onto a continuous function represented by its values at the nodes of the FEM mesh. This mapping is called a projection. Projection can be a useful tool but needs to be applied with some care due to the fact that a potentially discontinuous function is projected onto a continuous function but it can also be a desirable effect for instance to smooth a function. The projection of the gradient of a function typically calculated on the element center to the nodes of a FEM mesh can be evaluated on the domain boundary and so projection provides a tool to extrapolate the gradient from the internal to the boundary. This is only a reasonable procedure in the absence of singularities at the boundary.

As projection is used often in simulations esys.escript provides an easy to use class Projector which is part of the esys.escript.pdetools module. The following script demonstrates the usage of the class to project the piecewise constant function \( x_0 \geq 0 \rightarrow 1 \) \( x_0 < 0 \rightarrow -1 \) to a function with the reduced solution FunctionSpace attribute (default target)

```python
from esys.escript.pdetools import Projector
proj=Projector(domain)
x0=domain.getX()[0]
jmp=1.-2.*wherePositive(x0-0.5)
u=proj.getValue(jmp)
# alternative call:
u=proj(jmp)
```

By default the class uses lumping to solve the PDE 4.26. This technique is faster then using the standard solver techniques of PDEs. In essence it leads to using the average of neighbor element values to calculate the value at each FEM node.

The following script illustrate how to evaluate the normal stress on the boundary from a given displacement field \( u \):

```python
from esys.escript.pdetools import Projector
proj=Projector(u.getDomin())
e=symmetric(grad(u))
stress = G*e+ (K-2./3.*G)*trace(e)*kronecker(u.getDomin())
normal_stress = inner(u.getDomin().getNormal(), proj(stress))
```

**class Projector** (domain[, reduce=True[, fast=True]])

This class defines the projector on the domain domain. If reduce is set to True the projection will be returned as a reduced solution FunctionSpace Data object. Otherwise solution FunctionSpace representation is returned. If reduce is set to True lumping is used when the equation 4.26 is solved. Otherwise the standard PDE solver is used. Notice, that lumping is requires significant less compute time and memory. The class is callble.

**getSolverOptions()**

returns the solver options for solving the PDE. In fact the method returns a SolverOptions class object which can be used to modify the tolerance, the solver or the preconditioner, see Section 4.3 for details.

**getValue(input_data)**

projects the input_data. This method is equivalent to call an instance of the class with argument input_data:

### 4.3 Solver Options

**class SolverOptions()**

This class defines the solver options for a linear or non-linear solver. The option also supports the handling of diagnostic informations.
getSummary()  
Returns a string reporting the current settings

getName(key)  
Returns the name as a string of a given key

setSolverMethod(method=SolverOptions.DEFAULT)  
Sets the solver method to be used. Use method=SolverOptions.DIRECT to indicate that a direct  
rather than an iterative solver should be used and use method=SolverOptions.ITERATIVE  
to indicate that an iterative rather than a direct solver should be used. The value of method  
must be one of the constants SolverOptions.DEFAULT, SolverOptions.DIRECT,  
SolverOptions.CHOLEVSKY, SolverOptions.PCG, SolverOptions.CR,  
SolverOptions.CGSL, SolverOptions.BICGSTAB, SolverOptions.SSOR,  
SolverOptions.GMRES, SolverOptions.PRES20, SolverOptions.LUMPING,  
SolverOptions.ITERATIVE, SolverOptions.DEFAULT, SolverOptions.OUTLIERGMRES,  
SolverOptions.TFQMR, SolverOptions.MINRES, or SolverOptions.GAUSS_SEIDEL.  
Not all packages support all solvers. It can be assumed that a package makes a reasonable choice if it  
encounters. See Table 7.2 for the solvers supported by esys.finley.

getsolverMethod()  
Returns key of the solver method to be used.

setPreconditioner(preconditioner=SolverOptions.JACOBI)  
Sets the preconditioner to be used. The value of preconditioner must  
be one of the constants SolverOptions.SSOR, SolverOptions.ILU0,  
SolverOptions.ILUT, SolverOptions.JACOBI, SolverOptions.AMG,  
SolverOptions.REC_ILU, SolverOptions.GAUSS_SEIDEL, SolverOptions.RILU,  
or SolverOptions.NO_PRECONDITIONER. Not all packages support all preconditioner. It can  
be assumed that a package makes a reasonable choice if it encounters an unknown preconditioner. See  
Table 7.3 for the solvers supported by esys.finley.

getPreconditioner()  
Returns key of the preconditioner to be used.

setPackage(package=SolverOptions.DEFAULT)  
Sets the solver package to be used as a solver. The value of method must be one of the constants  
in SolverOptions.DEFAULT, SolverOptions.PASO, SolverOptions.SUPER_LU,  
SolverOptions.PASTIX, SolverOptions.MKL, SolverOptions.UMFPACK,  
SolverOptions.TRILINOS. Not all packages support all implementation. An exception  
may be thrown on some platforms if a particular package is requested. Currently esys.finley  
supports SolverOptions.PASO (as default) and, if available, SolverOptions.MKL 3 and  
SolverOptions.UMFPACK

getPackage()  
Returns the solver package key

resetDiagnostics(all=False)  
resets the diagnostics. If all is True all diagnostics including accumulative counters are reset.

getDiagnostics(name)  
Returns the diagnostic information name. The following keywords are supported:

• "num_iter": the number of iteration steps
• "cum_num_iter": the cumulative number of iteration steps
• "num_level": the number of level in multi level solver
• "num_inner_iter": the number of inner iteration steps
• "cum_num_inner_iter": the cumulative number of inner iteration steps
• "time": execution time
• "cum_time": cumulative execution time

---

3If the stiffness matrix is non-regular MKL may return without returning a proper error code. If you observe suspicious solutions when  
using MKL, this may cause by a non-invertible operator.
• "set_up_time": time to set up of the solver, typically this includes factorization and reordering
• "cum_set_up_time": cumulative time to set up of the solver
• "net_time": net execution time, excluding setup time for the solver and execution time for preconditioner
• "cum_net_time": cumulative net execution time
• "residual_norm": norm of the final residual
• "converged": return self.__converged

hasConverged()
Returns True if the last solver call has been finalized successfully. If an exception has been thrown by the solver the status of this flag is undefined.

setCoarsening([method=SolverOptions.DEFAULT])
Sets the key of the coarsening method to be applied in SolverOptions.AMG. The value of method must be one of the constants SolverOptions.DEFAULT, SolverOptions.STANDARD_COARSENING, SolverOptions.YAIR_SHAPIRA_COARSENING, SolverOptions.RUGE_STUEBEN_COARSENING\(^4\), or SolverOptions.AGGREGATION_COARSENING.

goCoarsening()
Returns the key of the coarsening algorithm to be applied SolverOptions.AMG.

setReordering([ordering=SolverOptions.DEFAULT_REORDERING])
Sets the key of the reordering method to be applied if supported by the solver. Some direct solvers support reordering to optimize compute time and storage use during elimination. The value of ordering must be one of the constants SolverOptions.NO_REORDERING, SolverOptions.MINIMUM_FILL_IN, SolverOptions.NESTED_DISSECTION, or SolverOptions.DEFAULT_REORDERING.

goReordering()
Returns the key of the reordering method to be applied if supported by the solver.

setRestart([restart=None])
Sets the number of iterations steps after which SolverOptions.GMRES is performing a restart. If restart is equal to None no restart is performed.

getRestart()
Returns the number of iterations steps after which SolverOptions.GMRES is performing a restart.

setTruncation([truncation=20])
Sets the number of residuals in SolverOptions.GMRES to be stored for orthogonalization. The more residuals are stored the faster SolverOptions.GMRES converged but

goTruncation()
Returns the number of residuals in SolverOptions.GMRES to be stored for orthogonalization

setIterMax([iter_max=10000])
Sets the maximum number of iteration steps

getIterMax()
Returns maximum number of iteration steps

setLevelMax([level_max=10])
Sets the maximum number of coarsening levels to be used in the SolverOptions.AMG solver or preconditioner.

getLevelMax()
Returns the maximum number of coarsening levels to be used in an algebraic multi level solver or preconditioner

setCoarseningThreshold([theta=0.25])
Sets the threshold for coarsening in the SolverOptions.AMG solver or preconditioner

\(^4\)The Ruge-Stueben and aggregation coarsening algorithms used for measuring the strength of connection only, but splitting is done with greedy algorithm.
getCoarseningThreshold()
Returns the threshold for coarsening in the SolverOptions.AMG solver or preconditioner.

setMinCoarseMatrixSize([size=500])
Sets the minimum size of the coarsest level matrix in SolverOptions.AMG.

getMinCoarseMatrixSize()
Returns the minimum size of the coarsest level matrix in SolverOptions.AMG.

setSmotherer([smoother=SolverOptions.GAUSS_SEIDEL])
Sets the SolverOptions.JACOBI or SolverOptions.GAUSS_SEIDEL smoother to be used in SolverOptions.AMG.

getSmotherer()
Returns the key for SolverOptions.JACOBI or SolverOptions.GAUSS_SEIDEL smoother used in SolverOptions.AMG.

setNumSweeps([sweeps=2])
Sets the number of sweeps in a SolverOptions.JACOBI or SolverOptions.GAUSS_SEIDEL preconditioner.

getNumSweeps()
Returns the number of sweeps in a SolverOptions.JACOBI or SolverOptions.GAUSS_SEIDEL preconditioner.

setNumPreSweeps([sweeps=2])
Sets the number of sweeps in the pre-smoothing step of SolverOptions.AMG.

getNumPreSweeps()
Returns the number of sweeps in the pre-smoothing step of SolverOptions.AMG.

setNumPostSweeps([sweeps=2])
Sets the number of sweeps in the post-smoothing step of SolverOptions.AMG.

getNumPostSweeps()
Returns the number of sweeps in the post-smoothing step of SolverOptions.AMG.

setTolerance([rtol=1.e-8])
Sets the relative tolerance for the solver. 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.

getTolerance()
Returns the relative tolerance for the solver.

setAbsoluteTolerance([atol=0.])
Sets the absolute tolerance for the solver. 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.

getAbsoluteTolerance()
Returns the absolute tolerance for the solver.

setInnerTolerance([rtol=0.9])
Sets the relative tolerance for an inner iteration scheme for instance on the coarsest level in a multi-level scheme.

getInnerTolerance()
Returns the relative tolerance for an inner iteration scheme.

setDropTolerance([drop_tol=0.01])
Sets the relative drop tolerance in ILUT.

getDropTolerance()
Returns the relative drop tolerance in SolverOptions.ILUT.

setDropStorage([storage=2.])
Sets the maximum allowed increase in storage for SolverOptions.ILUT. storage=2 would mean that
a doubling of the storage needed for the coefficient matrix is allowed in the SolverOptions.ILUT factorization.

getDropStorage()
Returns the maximum allowed increase in storage for SolverOptions.ILUT

setRelaxationFactor([factor=0.3])
Sets the relaxation factor used to add dropped elements in SolverOptions.RILU to the main diagonal.

getRelaxationFactor()
Returns the relaxation factor used to add dropped elements in RILU to the main diagonal.

isSymmetric()
Returns True if the discrete system is indicated as symmetric.

setSymmetryOn()
Sets the symmetry flag to indicate that the coefficient matrix is symmetric.

setSymmetryOff()
Clears the symmetry flag for the coefficient matrix.

isVerbose()
Returns True if the solver is expected to be verbose.

setVerbosityOn()
Switches the verbosity of the solver on.

setVerbosityOff()
Switches the verbosity of the solver off.

adaptInnerTolerance()
Returns True if the tolerance of the inner solver is selected automatically. Otherwise the inner tolerance set by setInnerTolerance is used.

setInnerToleranceAdaptionOn()
Switches the automatic selection of inner tolerance on

setInnerToleranceAdaptionOff()
Switches the automatic selection of inner tolerance off.

setInnerIterMax([iter_max=10])
Sets the maximum number of iteration steps for the inner iteration.

getInnerIterMax()
Returns maximum number of inner iteration steps.

acceptConvergenceFailure()
Returns True if a failure to meet the stopping criteria within the given number of iteration steps is not raising in exception. This is useful if a solver is used in a non-linear context where the non-linear solver can continue even if the returned solution does not necessarily meet the stopping criteria. One can use the hasConverged method to check if the last call to the solver was successful.

setAcceptanceConvergenceFailureOn()
Switches the acceptance of a failure of convergence on.

setAcceptanceConvergenceFailureOff()
Switches the acceptance of a failure of convergence off.

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.

MKL
the MKL library by Intel, Reference [22].

UMFPACK
the UMFPACK, Reference [33]. Remark: UMFPACK is not parallelized.

5The MKL library will only be available when the Intel compilation environment is used.
PASO

PASO is the solver library of esys.finley, see Section 7.

ITERATIVE

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

DIRECT

the default direct linear solver.

CHOLESKY

direct solver based on Cholesky factorization (or similar), see Reference [29]. The solver will require a symmetric PDE.

PCG

preconditioned conjugate gradient method, see Reference [35]. The solver will require a symmetric PDE.

TFQMR

transpose-free quasi-minimal residual method, see Reference [35].

GMRES

the GMRES method, see Reference [35]. 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 [35].

CGS

conjugate gradient squared method, see Reference [35].

BICGSTAB

stabilized bi-conjugate gradients methods, see Reference [35].

SSOR

symmetric successive over-relaxation method, see Reference [35]. 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 [29].

ILUT

the incomplete LU factorization preconditioner with fill-in, see Reference [29]. During the LU-factorization element with relative size less then getDropTolerance are dropped. Moreover, the size of the LU-factorization is restricted to the getDropStorage-fold of the stiffness matrix. getDropTolerance and getDropStorage are both set in the getSolution call.

JACOBI

the Jacobi preconditioner, see Reference [29].

AMG

the algebraic–multi grid method, see Reference [30]. This method can be used as linear solver method but is more robust when used in a preconditioner.

GAUSS_SEIDEL

the symmetric Gauss-Seidel preconditioner, see Reference [29]. getNumSweeps() is the number of sweeps used.

4.3. Solver Options
RILU
relaxed incomplete LU factorization preconditioner, see Reference [?]. This method is similar to SolverOptions.ILU0 but dropped elements are added to the main diagonal with the relaxation factor getRelaxationFactor.

REC_ILU
recursive incomplete LU factorization preconditioner, see Reference [37]. This method is similar to SolverOptions.ILU0 but applies reordering during the factorization.

NO_REORDERING
no ordering is used during factorization.

DEFAULT_REORDERING
the default reordering method 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.

TRILINOS
the Trilinos library is used as a solver Reference [?]

SUPER_LU
the SuperLU library is used as a solver Reference [?]

PASTIX
the Pastix library is used as a solver Reference [?]

STANDARD_COARSENING
SolverOptions.AMG coarsening method by Ruge and Stueben using measure of importance principle [32].

YAIR_SHAPIRA_COARSENING
SolverOptions.AMG coarsening method by Yair-Shapira

RUGE_STUEBEN_COARSENING
SolverOptions.AMG coarsening method by Ruge and Stueben using greedy algorithm for splitting.

AGGREGATION_COARSENING
SolverOptions.AMG coarsening using (symmetric) aggregation using greedy algorithm for splitting.

NO_PRECONDITIONER
no preconditioner is applied.
CHAPTER
FIVE

The Module esys.pycad

5.1 Introduction

esys.pycad provides a simple way to build a mesh for your finite element simulation. You begin by building what we call a Design using primitive geometric objects, and then to go on to build a mesh from this. The final step of generating the mesh from a Design uses freely available mesh generation software, such as Gmsh[19].

A 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 Design, you can build meshes for arbitrarily complex 2-D and 3-D structures.

5.2 The Unit Square

So the simplest geometry is the unit square. First we generate the corner points

```python
from esys.pycad import *
p0=Point(0.,0.,0.)
p1=Point(1.,0.,0.)
p2=Point(1.,1.,0.)
p3=Point(0.,1.,0.)
```

which are then linked to define the edges of the square

```python
l01=Line(p0,p1)
l12=Line(p1,p2)
l23=Line(p2,p3)
l30=Line(p3,p0)
```

The lines are put together to form a loop

```python
c=CurveLoop(l01,l12,l23,l30)
```

The orientation of the line defining the CurveLoop is important. It is assumed that the surrounded area is to the left when moving along the lines from their starting points towards the end points. Moreover, the line need to form a closed loop.

We use the CurveLoop to define a surface

```python
s=PlaneSurface(c)
```

Notice there is difference between the CurveLoop defining the boundary of the surface and the actually surface PlaneSurface. This difference becomes clearer in the next example with a hole. The direction of the lines is important. Now we are ready to define the geometry which described by an instance of Design class:

```python
d=Design(dim=2,element_size=0.05)
```

Here we use the two dimensional domain with a local element size in the finite element mesh of 0.05. We then add the surface s to the geometry
d.addItems(s)

This will automatically import all items used to construct s into the Design d. Now we are ready to construct a
esys.finley FEM mesh and then write it to the file ‘quad.fly’:

```python
from esys.finley import MakeDomain
dom=MakeDomain(d)
dom.write("quad.fly")
```

In some cases it is useful to access the script used to generate the geometry. You can specify a specific name for
the script file. In our case we use

```python
d.setScriptFileName("quad.geo")
```

It is also useful to check error messages generated during the mesh generation process. Gmsh writes messages
to the `.gmsh-errors` in your home directory.

If we put everything together we get the script

```python
from esys.pycad import *
from esys.pycad.gmsh import Design
from esys.finley import MakeDomain
p0=Point(0.,0.,0.)
p1=Point(1.,0.,0.)
p2=Point(1.,1.,0.)
p3=Point(0.,1.,0.)
l01=Line(p0,p1)
l12=Line(p1,p2)
l23=Line(p2,p3)
l30=Line(p3,p0)
c=CurveLoop(l01,l12,l23,l30)
s=PlaneSurface(c)
d=Design(dim=2,element_size=0.05)
d.setScriptFileName("quad.geo")
d.addItems(s)
pl1=PropertySet("sides",l01,l23)
pl2=PropertySet("top_and_bottom",l12,l30)
d.addItems(pl1, pl2)
dom=MakeDomain(d)
dom.write("quad.fly")
```

This example is included with the software in `quad.py` in the example directory.

There are three extra statements which we have not discussed yet: By default the mesh used to subdivide the
boundary are not written into the mesh file mainly to reduce the size of the data file. One need to explicitly add
the lines to the Design which should be present in the mesh data. Here we additionally labeled the lines on the
top and the bottom with the name “top_and_bottom” and the lines on the left and right hand side with the name
“sides” using PropertySet objects. The labeling is convenient when using tagging, see Chapter 3.

If you have Gmsh installed you can run the example and view the geometry and mesh with:

```bash
run-escript quad.py
gmsh quad.geo
gmsh quad.msh
```

You can access error messages from Gmsh in the `.gmsh-errors` in your home directory. See Figure 5.1 for a
result.

In most cases it is best practice to generate the mesh and to solve the mathematical model in to different scripts.
In our example you can read the esys.finley mesh into your simulation code¹ using

```python
from finley import ReadMesh
mesh=ReadMesh("quad.fly")
```

¹ Gmsh files can be directly read using the ReadGmsh, see Chapter 7
Note that the underlying mesh generation software will not accept all the geometries you can create with esys.pycad. For example, esys.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\cite{gmsh} will fail to create a mesh for it.

5.3 Holes

The example included below shows how to use esys.pycad to create a 2-D mesh in the shape of a trapezoid with a cut-out area, see Figure 5.2:

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

Figure 5.1: Trapozid with triangle Hole.

Figure 5.2: Trapozid with triangle Hole.
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.
5.4 A 3D example

In this section we discuss the definition of 3D geometries. The example is the unit cube, see Figure 5.3. First we generate the vertices of the cube:

```python
from esys.pycad import *
p0=Point(0.,0.,0.)
p1=Point(1.,0.,0.)
p2=Point(0.,1.,0.)
p3=Point(1.,1.,0.)
p4=Point(0.,0.,1.)
p5=Point(1.,0.,1.)
p6=Point(0.,1.,1.)
p7=Point(1.,1.,1.)
```

We connect the points to form the bottom and top surfaces of the cube:

```python
l01=Line(p0,p1)
l13=Line(p1,p3)
l32=Line(p3,p2)
l20=Line(p2,p0)
bottom=PlaneSurface(CurveLoop(l01,l13,l32,l20))
```

and

```python
l45=Line(p4,p5)
l57=Line(p5,p7)
l76=Line(p7,p6)
l64=Line(p6,p4)
top=PlaneSurface(CurveLoop(l45,l57,l76,l64))
```

To form the front face we introduce the two additional lines connecting the left and right front points of the top and bottom face:

```python
l15=Line(p1,p5)
l40=Line(p4,p0)
```

To form the front face we encounter the problem as the line \( l_{45} \) used to define the top face is pointing the wrong direction. In `esys.pycad` you can reversing direction of an object by changing its sign. So we write \(-l_{45}\) to indicate that the direction is to be reversed. With this notation we can write

```python
front=PlaneSurface(CurveLoop(l01,l15,-l45,l40))
```

Keep in mind that if you use `Line(p4,p5)` instead of \(-l_{45}\) both objects are treated as different although the connecting the same points with a straight line in the same direction. The resulting geometry would include an opening along the \( p_4-p_5 \) connection. This will lead to an inconsistent mesh and may result in a failure of the volumetric mesh generator. Similarly we can define the other sides of the cube:

```python
l37=Line(p3,p7)
l62=Line(p6,p2)
back=PlaneSurface(CurveLoop(l32,-l62,-l76,-l37))
left=PlaneSurface(CurveLoop(-l40,-l64,l62,l20))
right=PlaneSurface(CurveLoop(-l15,l13,l37,-l57))
```

We can now put the six surfaces together to form a `SurfaceLoop` defining the boundary of the volume of the cube:

```python
sl=SurfaceLoop(top,-bottom,front,back,left,right)
v=Volume(sl)
```

Similar to the definition of a `CurvedLoop` the orientation of the surfaces `SurfaceLoop` is relevant. In fact the surface normal direction defined by the the right hand rule needs to point outwards as indicated by the surface normals in Figure 5.3. As the bottom face is directed upwards it is inserted with the minus sign into the `SurfaceLoop` in order to adjust the orientation of the surface.

As in the 2D case, the `Design` class is used to define the geometry:
from esys.pycad.gmsh import Design
from esys.finley import MakeDomain

des=Design(dim=3, element_size = 0.1, keep_files=True)
des.setScriptFileName("brick.geo")
des.addItems(v, top, bottom, back, front, left, right)

dom=MakeDomain(des)
dom.write("brick.fly")

Note that the esys.finley mesh file ‘brick.fly’ will contain the triangles used to define the surfaces as they are added to the Design. The example script of the cube is included with the software in ‘brick.py’ in the example directory.

5.5 Alternative File Formats

pycad supports other file formats in including I-DEAS universal file, VRML, Nastran and STL. The following example shows how to generate the STL file ‘brick.stl’:

from esys.pycad.gmsh import Design

des=Design(dim=3, element_size = 0.1, keep_files=True)
des.addItems(v, top, bottom, back, front, left, right)

des.setFileFormat(des.STL)
des.setMeshFileName("brick.stl")
des.generate()

The example script of the cube is included with the software in ‘brick_stl.py’ in the example directory.

5.6 Element Sizes

The element size used globally is defined by the element_size argument of the Design. The mesh generator will try to use this mesh size everywhere in the geometry. In some cases it can be desirable to use locally a finer mesh. A local refinement can be defined at each Point:

\[
p0=\text{Point}(0.,0.,0.,\text{local\_scale}=0.01)
\]
Here the mesh generator will create a mesh with an element size which is by the factor 0.01 times smaller than the global mesh size \( \text{element}_\text{size}=0.3 \), see Figure 5.4. The point where a refinement is defined must be a point of curve used to define the geometry.

Alternatively, one can define a mesh size along a curve by defining the number of elements to be used to subdivide the curve. For instance, to use 20 element on line 123 on uses:

```python
l23=Line(p2, p3)
l23.setElementDistribution(20)
```

Setting the number of elements on a curve overwrites the global mesh size \( \text{element}_\text{size} \). The result is shown in Figure 5.4.

## 5.7 esys.pycad Classes

### 5.7.1 Primitives

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

**class Point** \((x=0., y=0., z=0., \text{local}_\text{scale}=1.)\)

Create a point with from coordinates with local characteristic length \( \text{local}_\text{scale} \)

**class CurveLoop** (list)

Create a closed curve from the list of Line, Arc, Spline, BSpline, BrezierSpline.

**class SurfaceLoop** (list)

Create a loop of PlaneSurface or RuledSurface, which defines the shell of a volume.

**Lines**

**class Line** (point1, point2)

Create a line with between starting and ending points.

**setElementDistribution** (\(n[\text{progression}=1, \text{createBump}=False]\))

Defines the number of elements on the line. If set it overwrites the local length setting which would be applied. The progression factor \( \text{progression} \) defines the change of element size between neighboured elements. If \( \text{createBump} \) is set progression is applied towards the centre of the line.

**resetElementDistribution** ()

removes a previously set element distribution from the line.

**getElementofDistribution** ()

Returns the element distribution as tuple of number of elements, progression factor and bump flag. If no element distribution is set None is returned.

**Splines**

**class Spline** (point0, point1, ...)

A spline curve defined by a list of points point0, point1, ...

**setElementDistribution** (\(n[\text{progression}=1, \text{createBump}=False]\))

Defines the number of elements on the line. If set it overwrites the local length setting which would be applied. The progression factor \( \text{progression} \) defines the change of element size between neighboured elements. If \( \text{createBump} \) is set progression is applied towards the centre of the line.

**resetElementDistribution** ()

removes a previously set element distribution from the line.

**getElementofDistribution** ()

Returns the element distribution as tuple of number of elements, progression factor and bump flag. If no element distribution is set None is returned.

5.7. esys.pycad Classes
BSplines

class BSpline (point0, point1, ...)
A B-spline curve defined by a list of points point0, point1, ...

setElementDistribution (n, progression=1, createBump=False)
Defines the number of elements on the line. If set it overwrites the local length setting which would be applied. The progression factor progression defines the change of element size between neighboured elements. If createBump is set progression is applied towards the centre of the line.

resetElementDistribution ()
removes a previously set element distribution from the line.

getElemenofDistribution ()
Returns the element distribution as tuple of number of elements, progression factor and bump flag. If no element distribution is set None is returned.

Brezier Curves

classBezierCurve (point0, point1, ...)
A Brezier spline curve defined by a list of points point0, point1, ...

setElementDistribution (n, progression=1, createBump=False)
Defines the number of elements on the line. If set it overwrites the local length setting which would be applied. The progression factor progression defines the change of element size between neighboured elements. If createBump is set progression is applied towards the centre of the line.

resetElementDistribution ()
removes a previously set element distribution from the line.

getElemenofDistribution ()
Returns the element distribution as tuple of number of elements, progression factor and bump flag. If no element distribution is set None is returned.

Arcs

classArc (centre_point, start_point, end_point)
Create an arc by specifying a centre for a circle and start and end points. An arc may subtend an angle of at most π radians.

setElementDistribution (n, progression=1, createBump=False)
Defines the number of elements on the line. If set it overwrites the local length setting which would be applied. The progression factor progression defines the change of element size between neighboured elements. If createBump is set progression is applied towards the centre of the line.

resetElementDistribution ()
removes a previously set element distribution from the line.

getElemenofDistribution ()
Returns the element distribution as tuple of number of elements, progression factor and bump flag. If no element distribution is set None is returned.

Plain surfaces

class PlaneSurface (loop, [holes=[list]])
Create a plane surface from a CurveLoop, which may have one or more holes described by list of CurveLoop.

setElementDistribution (n, progression=1, createBump=False)
Defines the number of elements on all lines.

setRecombination ([max_deviation=45 * DEG ])
the mesh generator will try to recombine triangular elements into quadrilateral elements. max_deviation
(in radians) defines the maximum deviation of any angle in the quadrilaterals from the right angle. Set `max_deviation=None` to remove recombination.

```python
setTransfiniteMeshing ([orientation="Left"])
```

applies 2D transfinite meshing to the surface. `orientation` defines the orientation of triangles. Allowed values are “Left”, “Right” or “Alternate”. The boundary of the surface must be defined by three or four lines where an element distribution must be defined on all faces where opposite faces uses the same element distribution. No holes must be present.

### Ruled Surfaces

**class RuledSurface (list)**

Create a surface that can be interpolated using transfinite interpolation. `list` gives a list of three or four lines defining the boundary of the surface.

```python
setRecombination ([max_deviation=45 * DEG ])
```

the mesh generator will try to recombine triangular elements into quadrilateral elements. `max_deviation` (in radians) defines the maximum deviation of any angle in the quadrilaterals from the right angle. Set `max_deviation=None` to remove recombination.

```python
setTransfiniteMeshing ([orientation="Left"])
```

applies 2D transfinite meshing to the surface. `orientation` defines the orientation of triangles. Allowed values are “Left”, “Right” or “Alternate”. The boundary of the surface must be defined by three or four lines where an element distribution must be defined on all faces where opposite faces uses the same element distribution. No holes must be present.

```python
setElementDistribution (n[,progression=1[,createBump=False ]])
```

Defines the number of elements on all lines.

### Volumes

**class Volume (loop, [holes=[list]])**

Create a volume given a `SurfaceLoop`, which may have one or more holes define by the list of `SurfaceLoop`.

```python
setElementDistribution (n[,progression=1[,createBump=False ]])
```

Defines the number of elements on all lines.

```python
setRecombination ([max_deviation=45 * DEG ])
```

the mesh generator will try to recombine triangular elements into quadrilateral elements. These meshes are then used to generate the volume mesh if possible. Together with transfinite meshing one can construct rectangular meshes. `max_deviation` (in radians) defines the maximum deviation of any angle in the quadrilaterals from the right angle. Set `max_deviation=None` to remove recombination.

```python
setTransfiniteMeshing ([orientation="Left"])
```

applies transfinite meshing to the volume and all surfaces (if `orientation` is not equal to `None`) `orientation` defines the orientation of triangles. Allowed values are “Left”, “Right” or “Alternate”. The boundary of the surface must be defined by three or four lines where an element distribution must be defined on all faces where opposite faces uses the same element distribution. If `orientation` is equal to `None` transfinite meshing is not switched on for the surfaces but needs to be set by the user. No holes must be present. **Warning:** The functionality of transfinite meshing without recombination is not entirely clear in Gmsh[19]. So please apply this method with care.

### 5.7.2 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=[0.0,0.0]])**

defines a translation $x \rightarrow x + b$. $b$ can be any object that can be converted into a `numpy` object of shape (3, ).
class Rotation(\([\text{axis}=[1,1,1], \text{point} = [0,0,0], \text{angle}=0^\text{*RAD}] \])
defines a rotation by \text{angle} around \text{axis} through \text{point} \text{point} and direction \text{axis}. \text{axis} and \text{point} can be any object that can be converted into a \text{numpy} object of shape (3,). \text{axis} does not have to be normalised but must have positive length. The right hand rule \cite{7} applies.

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

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

\text{DEG}

A constant to convert from degrees to an internal angle representation in radians. For instance use \(90^\text{DEG}\) for 90 degrees.

5.7.3 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 \text{items} must all be \text{Manifold1D}, \text{Manifold2D} or \text{Manifold3D} objects.

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

getchDim()
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 \text{Manifold1D}, \text{Manifold2D} or \text{Manifold3D}.

getItems()
returns the list of items

clearItems()
clears the list of items

getTag()
returns the tag used for this property set

5.8 Interface to the mesh generation software

The class and methods described here provide an interface to the mesh generation software, which is currently \text{Gmsh}\cite{19}. This interface could be adopted to triangle or another mesh generation package if this is deemed to be
desirable in the future.

```python
class Design ( [ dim=3, [ element_size=1., [ order=1, [ keep_files=False ] ] ] ])
```

The `Design` class 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.

**GMSH**
- gms file format [19].

**IDEAS**
- I-DEAS universal file format [3].

**VRML**
- VRML file format, [9].

**STL**
- STL file format [8].

**NASTRAN**
- NASTRAN bulk data format [5].

**MEDIT**
- Medit file format [4].

**CGNS**
- CGNS file format [1].

**PLOT3D**
- Plot3D file format [6].

**DIFFPACK**
- Diffpack 3D file format [2].

**DELAUNAY**
- the gmsh Delaunay triangulator.

**TETGEN**
- the TetGen [31] triangulator.

**NETGEN**
- the NETGEN [17] triangulator.

**generate()**
- generates the mesh file. The data are are written to the file `Design.getMeshFileName`.

**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])**
- set 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.

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

5.8. Interface to the mesh generation software
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 file name 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 mesh file. if no name is given a name is generated. The format is set by Design.setFileFormat.

getMeshFileName()
returns the name of the mesh file

addItems (*items)
adds the tuple of variitems. 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 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.

setFileFormat ([format=Design.GMSH])
set the file format. format must be one of the values Design.GMSH, Design.IDEAS, Design.VRML, Design.STL, Design.NASTRAN, Design.MEDIT, Design.CGNS, Design.PLOT3D or Design.DIFFPACK.

getFileFormat()
returns the file format.
The following sections give a brief overview of the model classes and their corresponding methods.

6.1 The Stokes Problem

In this section we discuss how to solve the Stokes problem which is defined as follows:

We want to calculate the velocity field \( v \) and pressure \( p \) of an incompressible fluid. They are given as the solution of the Stokes problem

\[
- (\eta(v_{i,j} + v_{j,i}))_j + p_i = f_i - \sigma_{ij,j}
\] (6.1)

where \( f_i \) defines an internal force and \( \sigma_{ij} \) is an initial stress. The viscosity \( \eta \) may weakly depend on pressure and velocity. If relevant we will use the notation \( \eta(v, p) \) to express this dependency.

We assume an incompressible media:

\[-v_{i,i} = 0\] (6.2)

Natural boundary conditions are taken in the form

\[(\eta(v_{i,j} + v_{j,i}))n_j - n_i p = s_i - \alpha \cdot n_i n_j v_j + \sigma_{ij} n_j\] (6.3)

which can be overwritten by constraints of the form

\[v_i(x) = v^D_i(x)\] (6.4)

at some locations \( x \) at the boundary of the domain. \( s_i \) defines a normal stress and \( \alpha \geq 0 \) the spring constant for restoring normal force. The index \( i \) may depend on the location \( x \) on the boundary. \( v^D_i \) is a given function on the domain.

6.1.1 Solution Method

If we assume that \( \eta \) is independent from the velocity and pressure equations 6.1 and 6.2 can be written in the block form

\[
\begin{bmatrix}
A & B^* \\
B & 0
\end{bmatrix}
\begin{bmatrix}
v \\
p
\end{bmatrix}
= 
\begin{bmatrix}
G \\
0
\end{bmatrix}
\] (6.5)

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 [11, 12].

If \( v_0 \) and \( p_0 \) are given initial guesses for velocity and pressure we calculate a correction \( dv \) for the velocity by solving the first equation of equation 6.5

\[Adv_1 = G - Av_0 - B^* p_0\] (6.6)

We then insert the new approximation \( v_1 = v_0 + dv_1 \) to calculate a correction \( dp_2 \) for the pressure and an additional correction \( dv_2 \) for the velocity by solving

\[
BA^{-1}B^* dp_2 = Bv_1
\]

\[Adv_2 = B^* dp_2\] (6.7)
The new velocity and pressure are then given by \( v_2 = v_1 - dv_2 \) and \( p_2 = p_0 + dp_2 \) which will fulfill the block system 6.5. This solution strategy is called the Uzawa scheme.

There is a problem with this scheme: In practice we will use an iterative scheme to solve any problem for operator \( A \). So we will be unable to calculate the operator \( BA^{-1}B^* \) required for \( dp_2 \) explicitly. In fact, we need to use another iterative scheme to solve the first equation in 6.7 where in each iteration step an iterative solver for \( A \) over the three equations 6.6 and 6.7.

Another issue is the fact that the viscosity \( \eta \) may depend on velocity or pressure and so we need to iterate over the three equations 6.6 and 6.7.

In the following we will use the two norms
\[
\|v\|_1^2 = \int_{\Omega} v_{j,k} v_{j,k} \, dx \quad \text{and} \quad \|p\|_0^2 = \int_{\Omega} p^2 \, dx. \tag{6.8}
\]
for velocity \( v \) and pressure \( p \). The iteration is terminated if the stopping criteria
\[
\max(\|Bv_1\|_0, \|v_2 - v_0\|_1) \leq \tau \cdot \|v_2\|_1 \tag{6.9}
\]
for a given given tolerance \( 0 < \tau < 1 \) is meet. Notice that because of the first equation of 6.7 we have that \( \|Bv_1\|_0 \) equals the norm of \( BA^{-1}B^* dp_2 \) and consequently provides a norm for the pressure correction.

We want to optimize the tolerance choice for solving 6.6 and 6.7. To do this we write the iteration scheme as a fixed point problem. Here we consider the errors produced by the iterative solvers being used. From Equation 6.6 we have
\[
v_1 = e_1 + v_0 + A^{-1}(G - Av_0 - B^*p_0) \tag{6.10}
\]
where \( e_1 \) is the error when solving 6.6. We will use a sparse matrix solver so we have not full control on the norm \( \|\cdot\|_s \) used in the stopping criteria for this equation. In fact we will have a stopping criteria of the form
\[
\|Ac_1\|_s = \|G - Av_1 - B^*p_0\|_s \leq \tau_1 \|G - Av_0 - B^*p_0\|_s \tag{6.11}
\]
where \( \tau_1 \) is the tolerance which we need to choose. This translates into the condition
\[
\|e_1\|_1 \leq K\tau_1 \|dv_1 - e_1\|_1 \tag{6.12}
\]
The constant \( K \) represents some uncertainty combining a variety of unknown factors such as the norm being used and the condition number of the stiffness matrix. From the first equation of 6.7 we have
\[
p_2 = p_0 + (BA^{-1}B^*)^{-1}(e_2 + Bv_1) \tag{6.13}
\]
where \( e_2 \) represents the error when solving 6.7. We use an iterative preconditioned conjugate gradient method (PCG) with iteration operator \( BA^{-1}B^* \) using the \( \|\cdot\|_0 \) norm. As suitable preconditioner for the iteration operator we use \( \frac{1}{\eta} \) [16], ie the evaluation of the preconitioner \( P \) for a given pressure increment \( q \) is the solution of
\[
\frac{1}{\eta}(Pq) = q. \tag{6.14}
\]
Note that in each evaluation of the iteration operator \( q = BA^{-1}B^*s \) one needs to solve the problem
\[
Aw = B^*s \tag{6.15}
\]
with sufficient accuracy to return \( q = Bw \). We assume that the desired tolerance is sufficiently small, for instance one can take \( \tau_2^2 \) where \( \tau_2 \) is the tolerance for 6.7.

In an implementation we use the fact that the residual \( r \) is given as
\[
r = B(v_1 - A^{-1}B^*dp) = B(v_1 - A^{-1}B^*dp) = B(v_1 - dv_2) = Bv_2 \tag{6.16}
\]
In particular we have \( e_2 = Bv_2 \) So the residual \( r \) is represented by the updated velocity \( v_2 = v_1 - dv_2 \). In practice, if one uses the velocity to represent the residual \( r \) there is no need to recover \( dv_2 \) in 6.7 after \( dp_2 \) has been calculated. In PCG the iteration is terminated if
\[
\|P^{1/2}Bv_2\|_0 \leq \tau_2 \|P^{1/2}Bv_1\|_0 \tag{6.17}
\]
where \( \tau_2 \) is the given tolerance. This translates into
\[
\|e_2\|_0 = \|Bv_2\|_0 \leq M\tau_2 \|Bv_1\|_0 \tag{6.18}
\]
where $M$ is taking care of the fact that $P^\perp$ is dropped.

As we assume that there is no significant error from solving with the operator $A$ we have

$$v_2 = v_1 - dv_2 = v_1 - A^{-1} B^* dp$$  \hspace{1cm} (6.19)

Combining the equations 6.10, 6.13 and 6.19 and setting the errors to zero we can write the solution process as a fix point problem

$$v = \Phi(v, p) \text{ and } p = \Psi(u, p)$$  \hspace{1cm} (6.20)

with suitable functions $\Phi(v, p)$ and $\Psi(v, p)$ representing the iteration operator without errors. In fact for a linear problem, $\Phi$ and $\Psi$ are constant. With this notation we can write the update step in the form $p_2 = \delta p + \Psi(v_0, p_0)$ and $v_2 = \delta v + \Phi(v_0, p_0)$ where the total error $\delta p$ and $\delta v$ are given as

$$\delta p = (BA^{-1} B^*)^{-1} (e_2 + Be_1)$$

$$\delta v = e_1 - A^{-1} B^* \delta p.$$  \hspace{1cm} (6.21)

Notice that $B \delta v = -e_2 = -B v_2$. Our task is now to choose the tolerances $\tau_1$ and $\tau_2$ such that the global errors $\delta p$ and $\delta v$ do not stop the convergence of the iteration process.

To measure convergence we use

$$\epsilon = \max(\|v_2 - v\|_1, \|BA^{-1} B^* (p_2 - p)\|_0)$$  \hspace{1cm} (6.22)

In practice using the fact that $BA^{-1} B^* (p_2 - p) = Bv_1$ and assuming that $v_2$ gives a better approximation to the true $v$ than $v_0$ we will use the estimate

$$\epsilon = \max(\|v_2 - v_0\|_1, \|Bv_1\|_0)$$  \hspace{1cm} (6.23)

to estimate the progress of the iteration step after the step is completed. Note that the estimate of $\epsilon$ used in the stopping criteria 6.9. If $\chi^-$ is the convergence rate assuming exact calculations, i.e. $e_1 = 0$ and $e_2 = 0$, we are expecting to maintain $\epsilon \leq \chi^- \cdot \epsilon^-$. For the pressure increment we get:

$$\|BA^{-1} B^* (p_2 - p)\|_0 \leq \|BA^{-1} B^* (p_2 - \delta p - p)\|_0 + \|BA^{-1} B^* \delta p\|_0 \leq \chi^- \cdot \epsilon^- + \|e_2 + Be_1\|_0 \approx \chi^- \cdot \epsilon^- + \|e_2\|_0 \leq \chi^- \cdot \epsilon^- + M \tau_2 \|Bv_1\|_0.$$  \hspace{1cm} (6.24)

So we choose the value for $\tau_2$ from

$$M \tau_2 \|Bv_1\|_0 \leq (\chi^-)^2 \epsilon^-$$  \hspace{1cm} (6.25)
in order to make the perturbation for the termination of the pressure iteration a second order effect. We use a similar argument for the velocity:

$$\|v_2 - v\|_1 \leq \|v_2 - \delta v - v\|_1 + \|\delta v\|_1 \leq \chi^- \cdot \epsilon^- + \|e_1 - A^{-1} B^* \delta p\|_1 \approx \chi^- \cdot \epsilon^- + \|e_1\|_1 \leq \chi^- \cdot \epsilon^- + K \tau_1 \|dv_1 - e_1\|_1 \leq (1 + K \tau_1) \chi^- \cdot \epsilon^-$$  \hspace{1cm} (6.26)

So we choose the value for $\tau_1$ from

$$K \tau_1 \leq \chi^-$$  \hspace{1cm} (6.27)

Assuming we have estimates for $M$ and $K$ (if none is available, we use the value 1.) we can use 6.27 and 6.25 to get appropriate values for the tolerances. After the step has been completed we can calculate a new convergence rate $\chi \equiv \frac{\chi^-}{\epsilon^-}$. For partial reasons we restrict $\chi$ to be less or equal a given maximum value $\chi_{max} \leq 1$. If we see $\chi \leq \chi^-(1 + \chi^-)$ our choices for the tolerances was suitable. Otherwise, we need to adjust the values for $K$ and $M$. From the estimates 6.24 and 6.26 we establish

$$\chi \leq (1 + \max(M \tau_2 \|Bv_1\|_0, K \tau_1)) \cdot \chi^-$$  \hspace{1cm} (6.28)

If we assume that this inequality would be an equation if we would have chosen the right values $M^+$ and $K^+$ then we get

$$\chi = (1 + \max(M^+ \frac{\chi^-}{M}, K^+ \frac{\chi^-}{K})) \cdot \chi^-$$  \hspace{1cm} (6.29)
From this equation we set our choice for \( K \) was not good enough. In this case we can calculate a new value

\[
K^+ = \frac{\chi - \chi^-}{(\chi^-)^2} K
\]  

(6.30)

In practice we will use

\[
K^+ = \max\left(\frac{\chi - \chi^-}{(\chi^-)^2}, \frac{1}{2} K, 1\right)
\]  

(6.31)

where the second term is used to reduce a potential overestimate of \( K \). The same identity is used for to update \( M \). The updated \( M^+ \) and \( K^+ \) are then use in the next iteration step to control the tolerances.

In some cases one can observe that there is a significant change in the velocity but the new velocity \( v_1 \) has still a small divergence, i.e. we have \( \|Bv_1\|_0 \ll \|v_1 - v_0\|_1 \). In this case we will get a small pressure increment and consequently only very small changes to the velocity as a result of the second update step which therefore can be skipped and we can directly repeat the first update step until the increment in velocity becomes significant relative to its divergence. In practice we will ignore the second half of the iteration step as long as

\[
\|Bv_1\|_0 \leq \theta \cdot \|v_1 - v_0\|
\]  

(6.32)

where \( 0 < \theta < 1 \) is a given factor. In this case we will also check the stopping criteria with \( v_1 \rightarrow v_2 \) but we will not correct \( M \) in this case.

Starting from an initial guess \( v_0 \) and \( p_0 \) for velocity and pressure the solution procedure is implemented as follows.

1. calculate viscosity \( \eta(v_0, p_0) \) and assemble operator \( A \) from \( \eta \).
2. calculate the tolerance \( \tau_1 \) from 6.27.
3. Solve 6.6 for \( dv_1 \) with tolerance \( \tau_1 \).
4. update \( v_1 = v_0 + dv_1 \)
5. if \( Bv_1 \) is large (see 6.32):
   a. calculate the tolerance \( \tau_2 \) from 6.25.
   b. Solve 6.7 for \( dp_2 \) and \( v_2 \) with tolerance \( \tau_2 \).
   c. update \( p_2 \leftarrow p_0 + dp_2 \)
6. else:
   a. update \( p_2 \leftarrow p \) and \( v_2 \leftarrow v_1 \)
7. calculate convergence measure \( \epsilon \) and convergence rate \( \chi \)
8. if stopping criteria 6.9 holds:
   a. return \( v_2 \) and \( p_2 \)
9. else:
   a. update \( M \) and \( K \)
   b. goto step 1 with \( v_0 \leftarrow v_2 \) and \( p_0 \leftarrow p_2 \).

6.1.2 Functions

class StokesProblemCartesian(domain)

draws the Stokes problem on the Domain domain. The domain needs to support LBB compliant elements for the Stokes problem, see \([20]\) for details. For instance one can use second order polynomials for velocity and first order polynomials for the pressure on the same element. Alternatively, one can use macro elements using linear polynomial for both pressure and velocity with a subdivided element for the velocity. Typically, the macro element is more cost effective. The fact that pressure and velocity are represented in different way is expressed by
velocity=Vector(0.0, Solution(mesh))
p
pressure=Scalar(0.0, ReducedSolution(mesh))

initialize( [f=Data(), fixed_u_mask=Data(), eta=1, surface_stress=Data(), stress=Data(), restoration_factor=0] )

assigns values to the model parameters. In any call all values must be set. f defines the external force \( f \), \( \eta \) the viscosity \( \eta \), \( \sigma \) the initial stress \( \sigma \). The locations and components where the velocity is fixed are set by the values of \( \text{fixed}\_\text{u}\_\text{mask} \). \( \text{restoration}\_\text{factor} \) defines the restoring force factor \( \alpha \). The method will try to cast the given values to appropriate Data class objects.

solve(v,p, max_iter=100, verbose=False, usePCG=True)

solves the problem and return approximations for velocity and pressure. The arguments \( v \) and \( p \) define initial guess. \( v \) must have function space Solution(domain) and \( p \) must have function space ReducedSolution(domain). The values of \( v \) marked by \( \text{fixed}\_\text{u}\_\text{mask} \) 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. \( \text{max}\_\text{iter} \) defines the maximum number of iteration steps.

If verbose is set to True informations on the progress 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.

getTolerance ()

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.

getAbsoluteTolerance ()

returns the current absolute tolerance.

getSolverOptionsVelocity ()

returns the solver options used solve the equations (6.6) and (6.15) for velocity.

getSolverOptionsPressure ()

returns the solver options used solve the preconditioner equation (6.14) for pressure.

getSolverOptionsDiv ()

set the solver options for solving the equation to project the divergence of the velocity onto the function space of pressure.

6.1.3 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()
scl=StokesProblemCartesian(dom)
mask= (whereZero(x[0])*[1.,0]+whereZero(x[0]-1))*[1.,0] + 
( whereZero(x[1])*[0.,1]+whereZero(x[1]-1))*[1.,1]
scl.initialize(eta=1, fixed_u_mask= mask)
v=Vector(0.,Solution(dom))
v[0]+=whereZero(x[1]-1.)
p=Scalar(0.,ReducedSolution(dom))
v,p=scl.solve(v,p, verbose=True)
saveVTK("u.xml",velocity=v,pressure=p)
```
6.2 Darcy Flux

We want to calculate the velocity $u$ and pressure $p$ on a domain $\Omega$ solving the Darcy flux problem

$$u_i + \kappa_{ij} p,j = g_i$$

$$u_{k,k} = f$$

with the boundary conditions

$$u_i n_i = u^N_i n_i \quad \text{on} \quad \Gamma_N$$

$$p = p^D \quad \text{on} \quad \Gamma_D$$

(6.34)

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 $\Omega$, $u^N$ and $p^D$ are given functions on $\Omega$, $g_i$ and $f$ are given source terms and $\kappa_{ij}$ is the given permeability. 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

$$\alpha_0 x_i x_i \leq \kappa_{ij} x_i x_j \leq \alpha_1 x_i x_i$$

for all $x_i$.

6.2.1 Solution Method

It is useful to write equation 6.33 in operator form. For any pressure $p$ we set

$$(Gp)_i = p,j$$

(6.36)

and a velocity $v$ we set

$$Dv = v_{k,k}$$

(6.37)

We $K = (\kappa_{ij})$ we can write the Darcy problem 6.33 as

$$u + KGp = g$$

$$Du = f$$

(6.38)

We solve this equation by minimising the functional

$$J(u, p) := \|K^{-\frac{1}{2}}(u + KGp - g)\|^2 + \|\lambda(Du - f)\|^2$$

(6.39)

over all suitable $u$ and $p$. In this equation we set $\|p\|_0^2 = (p, p)_0$ with

$$(p, q)_0 = \int_\Omega p \cdot q \, dx$$

(6.40)

The factor $\lambda > 0$ is a weighting factor. A simple calculation shows that one has to solve

$$(K^{-1}(v + KGq), u + KGp - g)_0 + (\lambda Du, \lambda(Du - f))_0 = 0$$

(6.41)

for all velocities $v$ and pressure $q$ which fulfill the homogeneous boundary conditions 6.34. This so-called variational equation can be translated back into operator notation

$$(K^{-1} + D^* \lambda^2 D)u + Gp = D^* \lambda f + K^{-1} g$$

$$G^* u + G^* KGp = G^* g$$

(6.42)

where $D^*$ and $Q^*$ denote the adjoint operators with respect to $(\cdot, \cdot)_0$. In [28] it has been shown that this problem is continuous and coercive and therefore has a unique solution. Also standard FEM methods can be used for discretization. It is also possible to solve the problem in 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 $v$ from the problem. Assuming that $p$ is known we have

$$v = (K^{-1} + D^* \lambda^2 D)^{-1}(D^* \lambda f + K^{-1} g - Gp)$$

(6.43)
To derive a preconditioner we use the identity
\[ \| \mathcal{J} \| = \| \mathcal{C} \| \| p \| \]
with \( p \) for the current pressure approximation. We use the PCG method to solve this. The residual \( r \) is given as
\[ r = G^* (g - K G p - v) \]
for the current pressure approximation \( p \) and current velocity \( v \) defined by equation 6.43. In a particular implementation we use \( r = g - K G p - v \) to represent the residual. The evaluation of the iteration operator for a given \( p \) is then returning \( Q p + v \) where \( v \) is the solution of
\[ (K^{-1} + D^* \lambda^2 D)v = G p \]

To derive a preconditioner we use the identity
\[ G^*(K - (K^{-1} + D^* \lambda^2 D)^{-1})G = G^*(I - (I + K D^* \lambda^2 D)^{-1})K G \approx G^*(K D^* \lambda^2 D)K G \approx G^* \frac{\lambda}{d x^2} K^2 G \]
where \( d x \) is the local mesh size and we use the approximation
\[ D^* \lambda^2 D \approx \frac{\lambda^2}{d x^2} \]
Therefore we use \( G^* \frac{\lambda^2}{d x^2} K^2 G \) as a preconditioner. To evaluate the preconditioner we need to solve the equation
\[ G^* \frac{\lambda^2}{d x^2} K^2 G p = G^* \hat{r} \]

It remains to answer the question how to choose \( \lambda \). We need to balance the first and second term in \( J(u, p) \) in equation 6.39. We inspect \( J \) for \( (\hat{u}, \hat{p}) \) which is a perturbed exact solution \( (u, p) \). Assuming \( \hat{u} = u + u_0 e^{i k \cdot x} \) and \( \hat{p} = p + p_0 e^{i k \cdot x} \) and constant \( K \) we get
\[ J(\hat{u}, \hat{p}) = C \left[ \| v \|^2 + \| l \|^2 \| \lambda \| \| u_0 \|^2 \right] \]
with some constant \( C > 0 \). The first two terms and the third term correspond to the first term and second term in the definition of \( J(u, p) \) in equation 6.39. For small \( \| k \|_2 \) (i.e. for a smooth perturbation) \( J(\hat{u}, \hat{p}) \) is dominated by \( \| K^{-1} \|_2 |u_0|^2 \). To scale the second term which corresponds to the incompressibility condition for the velocity we need to meet the condition \( \| K^{-1} \|_2 = \lambda \| k \|_2^2 \). Taking the boundary conditions into consideration the smallest possible value for \( \| k \|_2 = \frac{\pi}{l} \) where \( l \) is the longest edge of the domain. This leads to the
\[ \lambda = \frac{\| K^{-1} \|_2}{\frac{\pi}{l}} \]
Notice that with this setting the preconditioner \( G^* \frac{\lambda^2}{d x^2} K^2 G \) becomes equivalent to \( G^* K G \) if \( K \) is a diagonal matrix and the mesh has a constant mesh size.

The residual norm used in the PCG is given as
\[ ||r||_{PCG}^2 = \int r^*(G^* \frac{\lambda^2}{d x^2} K^2 G)^{-1} r \, d x = \int r^* G(G^* \frac{\lambda^2}{d x^2} K^2 G)^{-1} r \, d x \approx \int \hat{r} K^{-1} \hat{r} \, d x \]
The iteration is terminated if
\[ ||r||_{PCG} \leq ATOL \]
where we set
\[ ATOL = atol + rtol \left( \frac{1}{||K^{-1} v||_0} + \frac{1}{||K^2 G p||_0} \right)^{-1} \]
where rtol is a given relative tolerance and atol is a given absolute tolerance (typically \( = 0 \)). Notice that if \( G p \) 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 6.43. In practice 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 6.54 is still fulfilled the calculation is terminated and \((v, p)\) is returned. Otherwise PCG is restarted with a new ATOL.

6.2. Darcy Flux
6.2.2 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 6.33. 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 6.55.

setAbsoluteTolerance (atol=0)
   sets the absolute tolerance atol in 6.55.

getsolveroptionsflux()
   Returns the solver options used to solve the flux problems (6.43) and (6.47). Use the returned
SolverOptions object to control the solution algorithms.

getsolveroptionspressure()
   Returns the solver options used to solve the pressure problems (6.50) as a preconditioner. Use the returned
SolverOptions object to control the solution algorithms.

solve (u0,p0, max_iter=100, verbose=False)
   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. max_iter sets the maximum number of iterations
   steps allowed for solving the coupled problem.

6.2.3 Example: Gravity Flow

later

6.3 Isotropic Kelvin Material

As proposed by Kelvin [24] material strain $D_{ij} = \frac{1}{2}(v_{i,j} + v_{j,i})$ can be decomposed into an elastic part $D_{ij}^{el}$ and
visco-plastic part $D_{ij}^{vp}$:

$$D_{ij} = D_{ij}^{el} + D_{ij}^{vp}$$  (6.56)

with the elastic strain given as

$$D_{ij}^{el} = \frac{1}{2\mu} \bar{\sigma}_{ij}$$  (6.57)

where $\bar{\sigma}_{ij}$ is the deviatoric stress (Notice that $\bar{\sigma}_{ii} = 0$). If the material is composed by materials $q$ the visco-plastic
strain can be decomposed as

$$D_{ij}^{vp} = \sum_{q} D_{ij}^{q}$$  (6.58)

where $D_{ij}^{q}$ is the strain in material $q$ given as

$$D_{ij}^{q} = \frac{1}{2\eta^q} \bar{\sigma}_{ij}$$  (6.59)
where \( \eta^q \) is the viscosity of material \( q \). We assume the following betwee the the strain in material \( q \)

\[
\eta^q = \eta^q_N \left( \frac{\tau}{\tau^q} \right)^{1-n^q} \quad \text{with} \quad \tau = \sqrt{\frac{1}{2} \sigma^q_{ij} \sigma^q_{ij}}
\]  

(6.60)

for a given power law coefficients \( n^q \geq 1 \) and transition stresses \( \tau^q \), see [24]. Notice that \( n^q = 1 \) gives a constant viscosity. After inserting equation 6.59 into equation 6.58 one gets:

\[
D_{ij}^{vp} = \frac{1}{2\eta^vp} \sigma^q_{ij} \quad \text{with} \quad \frac{1}{\eta^vp} = \sum_q \frac{1}{\eta^q}.
\]

(6.61)

and finally with 6.56

\[
D_{ij}^{'} = \frac{1}{2\eta^vp} \sigma^q_{ij} + \frac{1}{2\mu} \dot{\gamma}_{ij}
\]

(6.62)

The total stress \( \tau \) needs to fullfill the yield condition

\[
\tau \leq \tau_Y + \beta p
\]

(6.63)

with the Drucker-Prager cohesion factor \( \tau_Y \), Drucker-Prager friction \( \beta \) and total pressure \( p \). The deviatoric stress needs to fullfill the equilibrion equation

\[
-\sigma^i_{ij,j} + p_{,i} = F_i
\]

(6.64)

where \( F_j \) is a given external fource. We assume an incompressible media:

\[
-v_i,i = 0
\]

(6.65)

Natural boundary conditions are taken in the form

\[
\sigma^i_{ij} n_j - n_i p = f
\]

(6.66)

which can be overwritten by a constraint

\[
v_i(x) = 0
\]

(6.67)

where the index \( i \) may depend on the location \( x \) on the bondary.

### 6.3.1 Solution Method

By using a first order finite difference approximation wit step size \( dt > 0 \) 6.57 get the form

\[
\dot{\sigma}_{ij} = \frac{1}{dt} (\sigma_{ij} - \sigma^-_{ij})
\]

(6.68)

and

\[
D_{ij}^{'} = \left( \frac{1}{2\eta^vp} + \frac{1}{2\mu dt} \right) \sigma^q_{ij} - \frac{1}{2\mu dt} \sigma^-_{ij}
\]

(6.69)

where \( \sigma^-_{ij} \) is the stress at the precious time step. With

\[
\dot{\gamma} = \sqrt{2 \left( D_{ij}^{'} + \frac{1}{2\mu dt} \sigma^-_{ij} \right)^2}
\]

(6.70)

we have

\[
\tau = \eta_{eff} \cdot \dot{\gamma}
\]

(6.71)

where

\[
\eta_{eff} = \min \left( \frac{1}{\mu dt} + \frac{1}{\eta^vp} \right)^{-1}, \eta_{max} \right) \quad \text{with} \quad \eta_{max} = \begin{cases} \frac{\tau_Y + \beta p}{\dot{\gamma}} & \text{if} \quad \dot{\gamma} > 0 \\ \infty & \text{otherwise} \end{cases}
\]

(6.72)

The upper bound \( \eta_{max} \) makes sure that yield condition 6.63 holds. With this setting the equation 6.69 takes the form

\[
\sigma^q_{ij} = 2\eta_{eff} \left( D_{ij}^{'} + \frac{1}{2\mu dt} \sigma^-_{ij} \right)
\]

(6.73)
After inserting 6.73 into 6.64 we get
\[- (\eta_{eff}(v_{i,j} + v_{i,j}))_j + p_{,i} = F_i + \left( \frac{\eta_{eff} \sigma_{i,j}}{\mu dt} \right)_j \] (6.74)

Combining this with the incompressibility condition 6.56 we need to solve a Stokes problem as discussed in section 6.1.1 in each time step.

If we set
\[\eta(\tau) = 1 + \eta_{\tau} \cdot \tau \] (6.75)

we need to solve the nonlinear problem
\[\eta_{eff} - \min(\eta(\dot{\gamma} \cdot \eta_{eff}), \eta_{max}) = 0 \] (6.76)

We use the Newton-Raphson Scheme to solve this problem
\[\eta_{eff}^{(n+1)} = \min(\eta_{max}, \eta_{eff}^{(n)} - \frac{\eta_{eff}^{(n)} - \eta(\tau^{(n)})}{1 - \dot{\gamma} \cdot \eta'(\tau^{(n)})}) = \min(\eta_{max}, \frac{\eta(\tau^{(n)}) - \eta(\tau^{(n)})}{1 - \dot{\gamma} \cdot \eta'(\tau^{(n)})}) \] (6.77)

where \(\eta'\) denotes the derivative of \(\eta\) with respect of \(\tau\) and \(\tau^{(n)} = \dot{\gamma} \cdot \eta^{(n)}\).

Looking at the evaluation of \(\eta\) in 6.75 it makes sense formulate the iteration 6.77 using \(\Theta = \eta^{-1}\). In fact we have
\[\eta' = - \frac{\Theta'}{\Theta^2} \text{ with } \Theta' = \sum q \left( \frac{1}{\eta^q} \right) \] (6.78)

As
\[\left( \frac{1}{\eta^q} \right)' = \frac{\eta^q - 1}{\eta^q N} \cdot \tau^{n^q-2} = \frac{\eta^q - 1}{\eta^q} \cdot \frac{1}{\tau} \] (6.79)

we have
\[\Theta' = \frac{1}{\tau} \omega \text{ with } \omega = \sum q \frac{n^q - 1}{\eta^q} \] (6.80)

which leads to
\[\eta_{eff}^{(n+1)} = \min(\eta_{max}, \eta_{eff}^{(n)} \eta^{(n)}_\Theta + \omega^{(n)}_\omega) \] (6.81)

6.3.2 Functions

class IncompressibleIsotropicFlowCartesian ( domain [, stress=0 [, v=0 [, p=0 [, t=0 [, numMaterials=1 [, verbose=True [, adaptSubTolerance=True ]]]]]]]) opens an incompressible, isotropic flow problem in Cartesian coordinates on the domain domain. stress, v, p, and t set the initial deviatoric stress, velocity, pressure and time. numMaterials specifies the number of materials used in the power law model. Some progress information are printed if verbose is set to True. If adaptSubTolerance is equal to True the tolerances for subproblems are set automatically.

The domain needs to support LBB compliant elements for the Stokes problem, see [20] for details. For instance one can use second order polynomials for velocity and first order polynomials for the pressure on the same element. Alternatively, one can use macro elements using linear polynomial for both pressure and velocity but with a subdivided element for the velocity. Typically, the macro element is more cost effective. The fact that pressure and velocity are represented in different way is expressed by

velocity=Vector(0.0, Solution(mesh))
pressure=Scalar(0.0, ReducedSolution(mesh))

gGetDomain() returns the domain.

gGetTime() Returns current time.
getStress()
    Returns current stress.

getDeviatoricStress()
    Returns current deviatoric stress.

getPressure()
    Returns current pressure.

getVelocity()
    Returns current velocity.

getDeviatoricStrain()
    Returns deviatoric strain of current velocity

getTau()
    Returns current second invariant of deviatoric stress

gammaDot()
    Returns current second invariant of deviatoric strain

setTolerance(tol=1.e-4)
    Sets the tolerance used to terminate the iteration on a time step.

setFlowTolerance(tol=1.e-4)
    Sets the relative tolerance for the incompressible solver, see StokesProblemCartesian for details.

setElasticShearModulus(mu=None)
    Sets the elastic shear modulus $\mu$. If $\mu$ is set to None (default) elasticity is not applied.

setEtaTolerance(rtol=1.e-8)
    Sets the relative tolerance for the effective viscosity. Iteration on a time step is completed if the realtive of the effective viscosity is less than $\operatorname{rtol}$.

setDruckerPragerLaw([tau_Y=None, friction=None])
    Sets the parameters $\tau_Y$ and $\beta$ for the Drucker-Prager model in condition 6.63. If $\tau_Y$ is set to None (default) Drucker-Prager condition is not applied.

setElasticShearModulus(mu=None)
    Sets the elastic shear modulus $\mu$. If $\mu$ is set to None (default) elasticity is not applied.

setPowerLaws(eta_N, tau_t, power)
    Sets the parameters of the power-law for all materials as defined in equation 6.60. $\eta_N$ is the list of viscosities $\eta^q_N$, $\tau_t$ is the list of reference stresses $\tau^q_t$, and $power$ is the list of power law coefficients $n^q$.

update(dt, iter_max=100, inner_iter_max=20)
    Updates stress, velocity and pressure for time increment $dt$. where $iter\_max$ is the maximum number of iteration steps on a time step to update the effective viscosity and $inner\_iter\_max$ is the maximum number of itertion steps in the incompressible solver.

6.3.3 Example

later

6.4 Fault System

The FaultSystem is an easy to use interface to handle 2D and 3D fault systems as used for instance in simulating fault ruptures. The main purpose of the class is to provide a parameterization of an individual fault in the system of fault. In case of a 2D fault the fault is parameterized by a single value $w_0$ and in the case of a 3D fault two parameters $w_0$ and $w_1$ are used. This parameterization can be used to impose data (e.g. a slip distribution) onto the fault. It can also be a useful tool to visualize or analyze the results on the fault if the fault is not straight.

A fault $t$ in the fault system is represented by a staring point $V^{t0}$ and series of direction, called strikes, and the
FIGURE 6.1: Two dimensional fault system with one fault named 't' in the $(x_0, x_1)$ and its parameterization in the $w_0$ space. The fault has three segments.

lengths ($l^{ti}$). The strike of segment $i$ is defined by the angle $\sigma^{ti}$ between the $x_0$-axis and the direction of the fault, see Figure 6.1. The length and strike defines the polyline ($V^{ti}$) of the fault by

$$V^{ti} = V^{t(i-1)} + l^{ti} \cdot S^{ti} \text{ with } S^{ti} = \begin{bmatrix} \cos(\sigma^{ti}) \\ \sin(\sigma^{ti}) \\ 0 \end{bmatrix}$$ (6.82)

In the 3D case each fault segment $i$ has an additional dip $\theta^{ti}$ and at each vertex $i$ a depth $\delta^{ti}$ is given. The fault segment normal $n^{ti}$ is given by

$$n^{ti} = \begin{bmatrix} -\sin(\theta^{ti}) \cdot S_1^{ti} \\ \sin(\theta^{ti}) \cdot S_0^{ti} \\ \cos(\theta^{ti}) \end{bmatrix}$$ (6.83)

At each vertex we define a depth vector $\tilde{d}^{ti}$ defined as the intersect of the fault planes of segment $(i-1)$ and $i$ where for the first segment and last segment the vector orthogonal to strike vector $S^{ti}$ and the segment normal $n^{ti}$ is used. The direction $d^{ti}$ of the depth vector is given as

$$\tilde{d}^{ti} = n^{ti} \times n^{t(i-1)}$$ (6.84)

If $\tilde{d}^{ti}$ is zero the strike vectors $L^{t(i-1)}$ and $L^{ti}$ are collinear and we can set $\tilde{d}^{ti} = l^{ti} \times n^{ti}$. If the two fault segments are almost orthogonal $\tilde{d}^{ti}$ is pointing in the direction of $L^{t(i-1)}$ and $L^{ti}$. In this case no depth can be defined. So we will reject a fault system if

$$\min(\|\tilde{d}^{ti} \times L^{t(i-1)}\|, \|\tilde{d}^{ti} \times L^{ti}\|) \leq 0.1 \cdot \|\tilde{d}^{ti}\|$$ (6.85)

which corresponds to an angle of less then $10^\circ$ between the depth vector and the strike. We then set

$$d^{ti} = \delta^{ti} \cdot \frac{\tilde{d}^{ti}}{\|\tilde{d}^{ti}\|}$$ (6.86)

We can then define the polyline ($v^{ti}$) for the bottom of the fault as

$$v^{ti} = V^{ti} + d^{ti}$$ (6.87)
In order to simplify working on a fault \( t \) in a fault system a parameterization \( P^t : (w_0, w_1) \rightarrow (x_0, x_1, x_2) \) over a rectangular domain is introduced such that
\[
0 \leq w_0 \leq w_{0\text{max}}^t \text{ and } -w_{1\text{max}}^t \leq w_1 \leq 0
\]  
(6.88)

with positive numbers \( w_{0\text{max}}^t \) and \( w_{1\text{max}}^t \). Typically one chooses \( w_{0\text{max}}^t \) to be the unrolled length of the fault \( w_{1\text{max}}^t \) to be the mean value of segment depth. Moreover we have
\[
P^t(W'^t) = V'^t \text{ and } P^t(w'^t) = v'^t
\]  
(6.89)

where
\[
W'^t = (\Omega'^t, 0) \text{ and } w'^t = (\Omega'^t, -w_{1\text{max}}^t)
\]  
(6.90)

and \( \Omega'^t \) is the unrolled distance of \( W'^t \) from \( W^t \), i.e. \( l'^t = \Omega'^{(i+1)} - \Omega'^i \). In the 2D case \( w_{1\text{max}}^t \) is set to zero and therefore the second component is dropped, see Figure 6.1.

In the case of 2D the parameterization \( P^t \) is constructed as follows: The line connecting \( V'^{(i-1)} \) and \( V'^{ti} \) is given by
\[
x = V'^t + s \cdot (V'^{(i+1)} - V'^{ti})
\]  
(6.91)

where \( s \) is between 0 and 1. The point \( x \) is on \( i \)-th fault segment if and only if such an \( s \) exists. If assume \( x \) is on the fault one can calculate \( s \) as
\[
s = \frac{(x - V'^{ti}) \cdot (V'^{(i+1)} - V'^{ti})}{\|V'^{(i+1)} - V'^{ti}\|^2}
\]  
(6.92)

We then can set
\[
w_0 = \Omega'^i + s \cdot (\Omega'^i - \Omega'^{(i-1)})
\]  
(6.93)

to get \( P^t(w_0) = x \). It remains the question if the given \( x \) is actual on the segment \( i \) of fault \( t \). To test this \( s \) is restricted between 0 and 1 (so if \( s < 0 \) \( s \) is set to 0 and if \( s > 1 \) \( s \) is set to 1) and we check the residual of equation 6.91, i.e. \( x \) is been accepted to be in the segment if
\[
\|x - V'^{ti} - s \cdot (V'^{(i+1)} - V'^{ti})\| \leq tol \cdot \text{max}(l'^i, \|x - V'^{ti}\|)
\]  
(6.94)

where \( tol \) is a given tolerance.

In the 3D case the situation is a bit more complicated: we split the fault segment across the diagonal \( V'^{ti} - v'^{(i+1)} \) to produce two triangles. In the upper triangle we use the parameterization
\[
x = V'^{ti} + s \cdot (V'^{(i+1)} - V'^{ti}) + r \cdot (v'^i - V'^{ti}) \text{ with } r \leq s;
\]  
(6.95)

while in the lower triangle we use
\[
x = V'^{ti} + s \cdot (v'^{(i+1)} - v'^{ti}) + r \cdot (v'^i - V'^{ti}) \text{ with } s \leq r;
\]  
(6.96)

where \( 0 \leq s, r \leq 1 \). Both equations are solved in the least-squares sense e.g. using the Moore-Penrose pseudoinverse for the coefficient matrices. The resulting \( s \) and \( r \) are then restricted to the unit square. Similar to the 2D case (see equation 6.94) we identify \( x \) to be in the upper triangle of the segment if
\[
\|x - V'^{ti} - s \cdot (V'^{(i+1)} - V'^{ti}) - r \cdot (v'^i - V'^{ti})\| \leq tol \cdot \text{max}(\|x - V'^{ti}\|, \|v'^{(i+1)} - V'^{ti}\|)
\]  
(6.97)

and in the lower part
\[
\|x - V'^{ti} - s \cdot (v'^{(i+1)} - v'^{ti}) - r \cdot (v'^i - V'^{ti})\| \leq tol \cdot \text{max}(\|x - V'^{ti}\|, \|v'^{(i+1)} - V'^{ti}\|)
\]  
(6.98)

after the restriction of \((s, t)\) to the unit square. Note that \( \|v'^{(i+1)} - V'^{(i)}\| \) is length of the diagonal of the fault segment. For those \( x \) which have been located in the \( i \)-th segment we then set
\[
w_0 = \Omega'^i + s \cdot (\Omega'^i - \Omega'^{(i-1)}) \text{ and } w_1 = w_{1\text{max}}^i (r - 1)
\]  
(6.99)
6.4.1 Functions

class FaultSystem(dim = 3)
  Creates a fault system in the dim dimensional space.

getMediumDepth(tag)
  returns the medium depth of fault tag.

getTags()
  returns a list of the tags used by the fault system

getStart(tag)
  returns the starting point of fault tag as a numpy.ndarray object.

getDim()
  returns the spatial dimension

getDepths(tag)
  returns the list of the depths of the segments in fault tag.

getTopPolyline(tag)
  returns the polyline used to describe fault tagged by tag

getStrikes(tag)
  returns the list of strikes $\sigma^{ti}$ of the segments in fault $t = tag$

getStrikeVectors(tag)
  returns the strike vectors $S^{ti}$ of fault $t = tag$

getLengths(tag)
  returns the lengths $l^{ti}$ of segments in fault $t = tag$

getTotalLength(tag)
  returns the total unrolled length of fault tag

getDips(tag)
  returns the list of the dips of the segments in fault tag.

getBottomPolyline(tag)
  returns the list of the vertices defining the bottom of the fault tag.

getSegmentNormals(tag)
  returns the list of the normals of the segments in fault tag.

getDepthVectors(tag)
  returns the list of the depth vectors $d^{ti}$ for fault $t = tag$.

getDepths(tag)
  returns the list of the depths of the segments in fault tag.

getW0Range(tag)
  returns the range of the parameterization in $w_0$. For tag $t$ this is the pair $(\Omega^{t0}, \Omega^{tn})$ where $n$ is the number of segments in the fault. In most cases one has $(\Omega^{t0}, \Omega^{tn}) = (0, w_{t_{\text{max}}}^{t})$.

getW1Range(tag)
  returns the range of the parameterization in $w_1$. For tag $t$ this is the pair $(-w_{t_{\text{max}}}^{t}, 0)$.

getW0Offsets(tag)
  returns the offsets for the parameterization of fault tag. For tag tag=$t$ this is the list $[\Omega^{ti}]$.

getCenterOnSurface()
  returns the center point of the fault system at the surfaces. In 3D the calculation of the center is considering the top edge of the faults and projects the edge to the surface (the $x_2$ component is assumed to be 0). An numpy.ndarray object is returned.

getOrientationOnSurface()
  returns the orientation of the fault system in RAD on the surface ($x_2 = 0$ plane) around the fault system center.
transform([\text{rot}=0, \text{shift}=\text{numpy.zeros((3,))}])

applies a shift shift and a consecutive rotation in the \( x_2 = 0 \) plane. \text{rot} is a float number and shift an \text{numpy.ndarray} object.

getMaxValue(f[, tol=1.e-8])

returns the tag of the fault of where \( f \) takes the maximum value and a \text{Locator} object which can be used to collect values from \text{Data} objects at the location where the maximum is taken, e.g.

\begin{verbatim}
fs=FaultSystem()
f=Scalar(..)
t, loc=fs.getMaxValue(f)
print 'maximum value of f on the fault %s is %s at location.'%(t,loc(f),loc.getX())
\end{verbatim}

\( f \) must be a scalar \text{Data} object. When the maximum is calculated only data sample points are considered which are on a fault in the fault system in the sense of condition 6.94 or 6.98, respectively. In the case no data sample points is found the returned tag is \text{None} and the maximum value as well as the location of the maximum value are undefined.

getMinValue(f[, tol=1.e-8])

returns the tag of the fault of where \( f \) takes the minimum value and a \text{Locator} object which can be used to collect values from \text{Data} objects at the location where the minimum is taken, e.g.

\begin{verbatim}
fs=FaultSystem()
f=Scalar(..)
t, loc=fs.getMinValue(f)
print 'minimum value of f on the fault %s is %s at location.'%(t,loc(f),loc.getX())
\end{verbatim}

\( f \) must be a scalar \text{Data} object. When the minimum is calculated only data sample points are considered which are on a fault in the fault system in the sense of condition 6.94 or 6.98, respectively. In the case no data sample points is found the returned tag is \text{None} and the minimum value as well as the location of the minimum value are undefined.

getParametrization(x[, tag=\text{None}][, \text{tol}=1.e-8], outsider=\text{None})

returns the argument \( w \) of the parameterization \( P^t \) for \( tag=t \) to provide \( x \) together with a mask indicating where the given location if on a fault in the fault system by the value 1 (otherwise the value is set to 0), \( x \) needs to be vector \text{Data} object or \text{numpy.ndarray} object. \text{tol} defines the tolerance to decide if a given data sample points is on fault \( tag \). The value \text{outside} is the value used as a replacement value for \( w \) where the corresponding value in \( x \) is not on a fault. If not \text{outside} is not present an appropriate value is used.

getSideAndDistance(x, tag)

returns the side and the distance at locations \( x \) from the fault \( tag \). \( x \) needs to be vector \text{Data} object or \text{numpy.ndarray} object. Positive values for side means that the corresponding location is to the right of the fault, a negative value means that the corresponding location is to the left of the fault. The value zero means that the side is undefined.

getFaultSegments(tag)

returns the polylines used to describe fault \( tag \). For \( tag=t \) this is the list of the vertices \([V]^t\) for the 2D and the pair of lists of the top vertices \([V]^t\) and the bottom vertices \([v]^t\) in 3D. Note that the coordinates are represented as \text{numpy.ndarray} objects.

addFault(strikes, l[, V0=[0,0,0], tag=\text{None}, dips=\text{None}, depths=\text{None}, w0_offsets=\text{None}, w1_max=\text{None}])

adds the fault \( tag \) to the fault system. \( V0 \) defines the start point of fault names \( t=tag \). The polyline defining the fault segments on the surface are set by the strike angles \( \text{strikes} = (\sigma^t, \text{north} = \pi/2, \text{the orientation is counter clockwise}.) \) and the length \( l \) and the length \( l \) and the dip angles \( \text{dips} = (\phi^t, \text{vertical}=0, \text{right-hand rule applies}.) \) and the depth \( \text{depths} \) for each segment. \( w1_{\text{max}} \) defines the range of the \( w1 \). If not present the mean value over the depth of all segment edges in the fault is used. \( w0_{\text{offsets}} \) sets the offsets \( \Omega^t \). If not present it is chosen such that \( \Omega = (\Omega(i-1))^t \) is the length of the \( i \)-th segment. In some cases, e.g. when kinks in the fault are relevant, it can be useful to explicitly specify the offsets in order to simplify the assignment of values.
6.4.2 Example

See section 1.7
The Module \texttt{esys.finley}

\texttt{finley} is a library of C functions solving linear, steady partial differential equations (PDEs) or systems of PDEs using isoparametrical finite elements. It supports unstructured, 1D, 2D and 3D meshes. The module \texttt{esys.finley} provides an access to the library through the \texttt{LinearPDE} class of \texttt{esys.escript} supporting its full functionality. \texttt{finley} is parallelized using the OpenMP paradigm.

7.1 Formulation

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

\begin{equation}
\int_\Omega A_{jl} \cdot v_j u_l + B_j \cdot v_j u + C_l \cdot v u_l + D \cdot v u \; d\Omega \\
+ \int_\Gamma d \cdot vu \; d\Gamma + \int_{\Gamma_{\text{contact}}} \left[ v \right] \left[ u \right] \; d\Gamma \\
= \int_\Omega X_j \cdot v_j + Y \cdot v \; d\Omega \\
+ \int_\Gamma y \cdot v \; d\Gamma + \int_{\Gamma_{\text{contact}}} y_{\text{contact}} \cdot \left[ v \right] \; d\Gamma
\end{equation}

(7.1)

7.2 Meshes

To understand the usage of \texttt{esys.finley} one needs to have an understanding of how the finite element meshes are defined. Figure 7.1 shows an example of the subdivision of an ellipse into so called elements. In this case, triangles have been used but other forms of subdivisions can be constructed, e.g., into quadrilaterals or, in the three dimensional case, into tetrahedrons and hexahedrons. The idea of the finite element method is to approximate the solution by a function which is a polynomial of a certain order and is continuous across it boundary to neighbor elements. In the example of Figure 7.1 a linear polynomial is used on each triangle. As one can see, the triangulation is quite a poor approximation of the ellipse. It can be improved by introducing a midpoint on each element edge then positioning those nodes located on an edge expected to describe the boundary, onto the boundary. In this case the triangle gets a curved edge which requires a parameterization of the triangle using a quadratic polynomial. For this case, the solution is also approximated by a piecewise quadratic polynomial (which explains the name isoparametrical elements), see Reference [38, 13] for more details. \texttt{esys.finley} supports macro elements. For these elements a piecewise linear approximation is used on an element which is further subdivided (in the case \texttt{esys.finley} halved). As such these elements do not provide more than a further mesh refinement but should be used in the case of incompressible flows, see \texttt{StokesProblemCartesian}. For these problems a linear approximation of the pressure across the element is used (use the reduced solution \texttt{FunctionSpace}) while the refined element is used to approximate velocity. So a macro element provides a continuous pressure approximation together with a velocity approximation on a refined mesh. This approach is necessary to make sure that the incompressible flow has a unique solution.

The union of all elements defines the domain of the PDE. Each element is defined by the nodes used to describe its shape. In Figure 7.1 the element, which has type \texttt{Tri3}, with element reference number 19 is defined by the nodes with reference numbers 9, 11 and 0. Notice that the order is counterclockwise. The coefficients of the PDE are evaluated at integration nodes with each individual element. For quadrilateral elements a Gauss quadrature
scheme is used. In the case of triangular elements a modified form is applied. The boundary of the domain is also subdivided into elements. In Figure 7.1 line elements with two nodes are used. The elements are also defined by their describing nodes, e.g. the face element reference number 20 which has type Line2 is defined by the nodes with the reference numbers 11 and 0. Again the order is crucial, if moving from the first to second node the domain has to lie on the left hand side (in the case of a two dimension surface element the domain has to lie on the left hand side when moving counterclockwise). If the gradient on the surface of the domain is to be calculated rich face elements face to be used. Rich elements on a face are identical to interior elements but with a modified order of nodes such that the 'first' face of the element aligns with the surface of the domain. In Figure 7.1 elements of the type Tri3Face are used. The face element reference number 20 as a rich face element is defined by the nodes with reference numbers 11, 0 and 9. Notice that the face element 20 is identical to the interior element 19 except that, in this case, the order of the node is different to align the first edge of the triangle (which is the edge starting with the first node) with the boundary of the domain.

Be aware that face elements and elements in the interior of the domain must match, i.e. a face element must be the face of an interior element or, in case of a rich face element, it must be identical to an interior element. If no face elements are specified esys.finley implicitly assumes homogeneous natural boundary conditions, i.e. \( d = 0 \) and \( y = 0 \), on the entire boundary of the domain. For inhomogeneous natural boundary conditions, the boundary must be described by face elements.
Table 7.1: Finley elements and corresponding elements to be used on domain faces and contacts. The rich types have to be used if the gradient of function is to be calculated on faces and contacts, respectively.

If discontinuities of the PDE solution are considered contact elements are introduced to describe the contact region $\Gamma_{\text{contact}}$ even if $d_{\text{contact}}$ and $y_{\text{contact}}$ are zero. Figure 7.2 shows a simple example of a mesh of rectangular elements around a contact region $\Gamma_{\text{contact}}$. The contact region is described by the elements 4, 3 and 6. Their element type is $\text{Line2}_\text{Contact}$. The nodes 9, 12, 6, 5 define contact element 4, where the coordinates of nodes 12 and 5 and nodes 4 and 6 are identical with the idea that nodes 12 and 9 are located above and nodes 5 and 6 below the contact region. Again, the order of the nodes within an element is crucial. There is also the option of using rich elements if the gradient is to be calculated on the contact region. Similarly to the rich face elements these are constructed from two interior elements by reordering the nodes such that the ‘first’ face of the element above and the ‘first’ face of the element below the contact regions line up. The rich version of element 4 is of type $\text{Rec4Face}_\text{Contact}$ and is defined by the nodes 9, 12, 16, 18, 6, 5, 0 and 2.

Table 7.1 shows the interior element types and the corresponding element types to be used on the face and contacts. Figure 7.3, Figure 7.4 and Figure 7.5 show the ordering of the nodes within an element.

The native $\text{esys.finley}$ file format is defined as follows. Each node $i$ has $\text{dim}$ spatial coordinates $\text{Node}[i]$, a reference number $\text{Node}\_\text{ref}[i]$, a degree of freedom $\text{Node}\_\text{DOF}[i]$ and tag $\text{Node}\_\text{tag}[i]$. In most cases $\text{Node}\_\text{DOF}[i]=\text{Node}\_\text{ref}[i]$ however, for periodic boundary conditions, $\text{Node}\_\text{DOF}[i]$ is chosen differently, see example below. The tag can be used to mark nodes sharing the same properties. Element $i$ is defined by the $\text{Element}\_\text{numNodes}$ nodes $\text{Element}\_\text{Nodes}[i]$ which is a list of node reference numbers. The order is crucial. It has a reference number $\text{Element}\_\text{ref}[i]$ and a tag $\text{Element}\_\text{tag}[i]$. The tag can be used to mark elements sharing the same properties. For instance elements above a contact region are marked with 2 and elements below a contact region are marked with 1. $\text{Element}\_\text{Type}$ and $\text{Element}\_\text{Num}$ give the element type and the number of elements in the mesh. Analogue notations are used for face and contact elements. The following Python script prints the mesh definition in the $\text{esys.finley}$ file format:

```python
print "%%s\\n"#mesh_name
# node coordinates:
print "%%dD-nodes %d\\n"%(dim,numNodes)
for i in range(numNodes):
    print " %d %d %d"%(Node_ref[i],Node_DOF[i],Node_tag[i])
    for j in range(dim): print " %e"%Node[i][j]
print "\n"
# interior elements
print "%%s\\n"%(Element_Type,Element_Num)
for i in range(Element_Num):
    print " %d %d"%(Element_ref[i],Element_tag[i])
    for j in range(Element_numNodes): print " %d"%Element_Nodes[i][j]
print "\n"
# face elements
```

7.2. Meshes
The following example of a mesh file defines the mesh shown in Figure 7.2:

```
Example 1
2D Nodes 16
0  0  0  0.  0.
2  2  0  0.33 0.
3  3  0  0.66 0.
7  4  0  1.  0.
5  5  0  0.  0.5
6  6  0  0.33 0.5
8  8  0  0.66 0.5
10 10 0  1.  0.5
12 12 0  0.  0.5
9  9  0  0.33 0.5
13 13 0  0.66 0.5
15 15 0  1.  0.5
16 16 0  0.  1.0
18 18 0  0.33 1.0
19 19 0  0.66 1.0
20 20 0  1.0  1.0
Rec4 6
0  1  0  2  6  5
1  2  3  8  6
2  1  3  7 10 8
5  2 12  9 18 16
7  2 13  9 18  9
10 2 20  19 13 15
Line2 0
Line2_Contact 3
4  0  9 12  6 5
3  0 13  9  8 6
6  0 15  13 10 8
Point1 0
```

Notice that the order in which the nodes and elements are given is arbitrary. In the case that rich contact elements are used the contact element section gets the form

```
Rec4Face_Contact 3
4  0  9 12  6 18 16 5 0 2
3  0 13  9 18 19 8 6 2 3
6  0 15  13 19 20 10 8 3 7
```

Periodic boundary condition can be introduced by altering `Node_DOF`. It allows identification of nodes even if they have different physical locations. For instance, to enforce periodic boundary conditions at the face $x_0 = 0$ and $x_0 = 1$ one identifies the degrees of freedom for nodes 0, 5, 12 and 16 with the degrees of freedom for 7, 10, 15 and 20, respectively. The node section of the esys.finley mesh gets now the form:
<table>
<thead>
<tr>
<th>Node</th>
<th>X</th>
<th>Y</th>
<th>X Scale</th>
<th>Y Scale</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
</tr>
<tr>
<td>2</td>
<td>0.2</td>
<td>0.0</td>
<td>0.33</td>
<td>0.0</td>
</tr>
<tr>
<td>3</td>
<td>0.3</td>
<td>0.0</td>
<td>0.66</td>
<td>0.0</td>
</tr>
<tr>
<td>7</td>
<td>0.0</td>
<td>0.6</td>
<td>1.0</td>
<td>0.0</td>
</tr>
<tr>
<td>5</td>
<td>0.5</td>
<td>0.0</td>
<td>0.0</td>
<td>0.5</td>
</tr>
<tr>
<td>6</td>
<td>0.6</td>
<td>0.0</td>
<td>0.33</td>
<td>0.5</td>
</tr>
<tr>
<td>8</td>
<td>0.8</td>
<td>0.0</td>
<td>0.66</td>
<td>0.5</td>
</tr>
<tr>
<td>10</td>
<td>1.0</td>
<td>0.0</td>
<td>1.0</td>
<td>0.5</td>
</tr>
<tr>
<td>12</td>
<td>1.2</td>
<td>0.0</td>
<td>0.0</td>
<td>0.5</td>
</tr>
<tr>
<td>9</td>
<td>0.9</td>
<td>0.0</td>
<td>0.33</td>
<td>0.5</td>
</tr>
<tr>
<td>13</td>
<td>1.3</td>
<td>0.0</td>
<td>0.66</td>
<td>0.5</td>
</tr>
<tr>
<td>15</td>
<td>1.5</td>
<td>0.0</td>
<td>1.0</td>
<td>0.5</td>
</tr>
<tr>
<td>14</td>
<td>1.4</td>
<td>0.0</td>
<td>0.0</td>
<td>1.0</td>
</tr>
<tr>
<td>18</td>
<td>1.8</td>
<td>0.0</td>
<td>0.33</td>
<td>1.0</td>
</tr>
<tr>
<td>19</td>
<td>1.9</td>
<td>0.0</td>
<td>0.66</td>
<td>1.0</td>
</tr>
<tr>
<td>20</td>
<td>2.0</td>
<td>0.0</td>
<td>1.0</td>
<td>1.0</td>
</tr>
</tbody>
</table>
Figure 7.3: Elements of order 1
Figure 7.4: Elements of order 2 and macro elements
Rec9 and Rec9Macro

Figure 7.5: Additional shape functions
7.3 Macro Elements

esys.finley supports the usage of macro elements which can be used to achieve LBB compliance when solving incompressible fluid flow problems. LBB compliance is required to get a problem which has a unique solution for pressure and velocity. For macro elements the pressure and velocity are approximated by a polynomial of order 1 but the velocity approximation bases on a refinement of the element. The nodes of a triangle and quadrilateral element is shown in Figures 7.6(a) and 7.6(b), respectively. In essence, the velocity uses the same nodes like a quadratic polynomial approximation but replaces the quadratic polynomial by piecewise linear polynomials. In fact, this is the way esys.finley is defining the macro elements. In particular esys.finley uses the same local ordering of the nodes for the macro element as for the corresponding quadratic element. Another interpretation is that one uses a linear approximation of the velocity together with a linear approximation of the pressure but on elements created by combining elements to macro elements. Notice that the macro elements still use quadratic interpolation to represent the element and domain boundary. However, if elements have linear boundary a macro element approximation for the velocity is equivalent to using a linear approximation on a mesh which is created through a one step, global refinement. Typically macro elements are only required to use when an incompressible fluid flow problem is solved, e.g the Stokes problem in Section 6.1. Please see Section 7.2 for more details on the supported macro elements.

7.4 Linear Solvers in SolverOptions

Table 7.2 and Table 7.3 show the solvers and preconditioners supported by esys.finley through the PASO library. Currently direct solvers are not supported under MPI. By default, esys.finley is using the iterative solvers SolverOptions.PCG for symmetric and SolverOptions.BICGSTAB for non-symmetric problems. If the direct solver is selected which can be useful when solving very ill-posed equations.
uses the MKL \(^1\) solver package. If MKL is not available UMFPACK is used. If UMFPACK is not available a suitable iterative solver from the PASO is used.

### 7.5 Functions

**ReadMesh** (fileName \([\text{integrationOrder}=-1, \text{optimize}=True]\))

creates a Domain object form the FEM mesh defined in file fileName. The file must be given the esys.finley file format. If integrationOrder is positive, a numerical integration scheme chosen which is accurate on each element up to a polynomial of degree \(\text{integrationOrder}\). Otherwise an appropriate integration order is chosen independently. By default the labeling of mesh nodes and element distribution is optimized. Set optimize=False to switch off relabeling and redistribution.

**ReadGmsh** (fileName \([\text{integrationOrder}=-1, \text{optimize}=True, \text{useMacroElements}=False]\))

creates a Domain object form the FEM mesh defined in file fileName. The file must be given the Gmsh\(^{[19]}\) file format. If integrationOrder is positive, a numerical integration scheme chosen which is accurate on each element up to a polynomial of degree \(\text{integrationOrder}\). Otherwise an appropriate integration order is chosen independently. By default the labeling of mesh nodes and element distribution is optimized. Set optimize=False to switch off relabeling and redistribution. If useMacroElements is set, second order elements are interpreted as macro elements. Currently ReadGmsh does not support MPI.

**MakeDomain** (design\([\text{integrationOrder}=-1, \text{optimizeLabeling}=True, \text{useMacroElements}=False]\))

Creates a Finley Domain from a Design object from esys.pycad using Gmsh\(^{[19]}\). The Design design defines the geometry. If integrationOrder is positive, a numerical integration scheme chosen which is accurate on each element up to a polynomial of degree \(\text{integrationOrder}\). Otherwise an appropriate integration order is chosen independently. Set optimizeLabeling=False to switch off relabeling and redistribution (not recommended). If useMacroElements is set, macro elements are used. Currently MakeDomain does not support MPI.

**load** (fileName)

recovery a Domain object from a dump file created by the dump method of a Domain object defined in file fileName.

**Rectangle** (n0,n1.order=1,l0=1.,l1=1., \(\text{integrationOrder}=-1, \text{periodic0}=False, \text{periodic1}=False, \text{useElementsOnFace}=False, \text{useMacroElements}=False, \text{optimize}=False\))

Generates a Domain object representing a two dimensional rectangle between \((0,0)\) and \((l0,l1)\) with orthogonal edges. The rectangle is filled with \(n0\) elements along the \(x0\)-axis and \(n1\) elements along the \(x1\)-axis. For \(\text{order}=1\) and \(\text{order}=2\) Rec4 and Rec8 are used, respectively. In the case of useElementsOnFace=False, Line2 and Line3 are used to subdivide the edges of the rectangle, respectively. If \(\text{order}=1\), Rec8Macro and Line3Macro. This option should be used when solving incompressible fluid flow problem, e.g. StokesProblemCartesian. In the case of useElementsOnFace=True (this option should be used if gradients are calculated on domain faces), Rec4Face and Rec8Face are used on the edges, respectively.

\(^1\) If the stiffness matrix is non-regular MKL may return without returning a proper error code. If you observe suspicious solutions when using MKL, this may cause by a non-invertible operator.
If integrationOrder is positive, a numerical integration scheme chosen which is accurate on each element up to a polynomial of degree integrationOrder. Otherwise an appropriate integration order is chosen independently. If periodic0=True, periodic boundary conditions along the x0-directions are enforced. That means when for any solution of a PDE solved by esys.finley the value on the line x0 = 0 will be identical to the values on x0 = l0. Correspondingly, periodic1=False sets periodic boundary conditions in x1-direction. If optimize=True mesh node relabeling will be attempted to reduce the computation and also ParMETIS will be used to improve the mesh partition if running on multiple CPUs with MPI.

**Brick** (n0,n1,n2,order=1,l0=1.,l1=1.,l2=1., integrationOrder=-1,
periodic0=False,periodic1=False,periodic2=False,useElementsOnFace=False, useMacroElements=False, optimize=False)

Generates a Domain object representing a three dimensional brick between (0,0,0) and (l0,l1,l2) with orthogonal faces. The brick is filled with n0 elements along the x0-axis, n1 elements along the x1-axis and n2 elements along the x2-axis. For order=1 and order=2 Hex8 and Hex20 are used, respectively. In the case of useElementsOnFace=False, Rec4 and Rec8 are used to subdivide the faces of the brick, respectively. In the case of useElementsOnFace=True (this option should be used if gradients are calculated on domain faces), Hex8Face and Hex20Face are used on the brick faces, respectively. If order=-1, Hex20Macro and Rec8Macro. This option should be used when solving incompressible fluid flow problem, e.g. StokesProblemCartesian. If integrationOrder is positive, a numerical integration scheme chosen which is accurate on each element up to a polynomial of degree integrationOrder. Otherwise an appropriate integration order is chosen independently. If periodic0=True, periodic boundary conditions along the x0-directions are enforced. That means when for any solution of a PDE solved by esys.finley the value on the plane x0 = 0 will be identical to the values on x0 = l0. Correspondingly, periodic1=False and periodic2=False sets periodic boundary conditions in x1-direction and x2-direction, respectively. If optimize=True mesh node relabeling will be attempted to reduce the computation and also ParMETIS will be used to improve the mesh partition if running on multiple CPUs with MPI.

**GlueFaces** (meshList,safetyFactor=0.2,tolerance=1.e-13)

Generates a new Domain object from the list meshList of esys.finley meshes. Nodes in face elements whose difference of coordinates is less then tolerance times the diameter of the domain are merged. The corresponding face elements are removed from the mesh.

TODO: explain safetyFactor and show an example.

**JoinFaces** (meshList,safetyFactor=0.2,tolerance=1.e-13)

Generates a new Domain object from the list meshList of esys.finley meshes. Face elements whose nodes coordinates have difference is less then tolerance times the diameter of the domain are combined to form a contact element. The corresponding face elements are removed from the mesh.

TODO: explain safetyFactor and show an example.
A.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 repeated subscripted variables, they are assumed to be summed over all possible values, from 0 to \( n \). For example, for the following expression:

\[
y = a_0 b_0 + a_1 b_1 + \ldots + a_n b_n
\]  

(A.1)

can be represented as:

\[
y = \sum_{i=0}^{n} a_i b_i
\]  

(A.2)

then in Einstein notation:

\[
y = a_i b_i
\]  

(A.3)

Another example:

\[
\nabla p = \frac{\partial p}{\partial x_0} \mathbf{i} + \frac{\partial p}{\partial x_1} \mathbf{j} + \frac{\partial p}{\partial x_2} \mathbf{k}
\]  

(A.4)

can be expressed in Einstein notation as:

\[
\nabla p = p_{,i}
\]  

(A.5)

where the comma ‘,’ indicates the partial derivative.

For a tensor:

\[
\sigma_{ij} = \begin{bmatrix}
\sigma_{00} & \sigma_{01} & \sigma_{02} \\
\sigma_{10} & \sigma_{11} & \sigma_{12} \\
\sigma_{20} & \sigma_{21} & \sigma_{22}
\end{bmatrix}
\]  

(A.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 \( (i \neq j) \).
\[ \delta_{ij} = \begin{cases} 1, & \text{if } i = j \\ 0, & \text{if } i \neq j \end{cases} \] (A.7)

A.2 Changes from previous releases

3.0 to 3.1

- The escript launcher has been renamed to run-escript the old-name will still work in this version but will be removed in future.
- Lazy evaluation features have been improved and documented (See Section 2.5).
- Escript documentation includes a new Cookbook which demonstrates how to solve sample problems using escript.
- Macro elements have been introduced.
- The saveDataCSV method allows one or more Data objects to be exported in CSV format.
- Data objects can be populated by interpolating from a values in a table.
- The new getInfLocator and getSupLocator functions in esys.escript.pdetools return Locators to a minimal/-maximal point over the data.
- There is a new class to model fault systems (esys.escript.faultsystems.FaultSystem).
- A beta version of an Algebraic Multigrid (AMG) solver is included.
- Inverting square matrices larger than 3x3 is now permitted if escript is compiled with Lapack support.
- If escript is compiled with a modern compiler, then inf/sup/Lsup will now report NaN, +/-inf as appropriate if those values appear in the data.
- Data.setTags will take tag names as well as tag numbers.
- The Scalar, Vector, Tensor, Tensor3, Tensor4 factory methods can now take array/nested sequence like objects as their initial values.
- escript.util.mkDir can now take a list of directories to create.
- Behind the scenes, python docstrings have been rewritten from epydoc to restructured text.
- Various other bug fixes and performance tweaks.

2.0 to 3.0

- The major change here was replacing numarray with numpy. For general instructions on converting scripts to use numpy see [http://www.stsci.edu/resources/software_hardware/numarray/numarray2numpy.pdf](http://www.stsci.edu/resources/software_hardware/numarray/numarray2numpy.pdf).
  The specific changes to esys.escript are:
  - getValueOfDataPoint() which returned a numarray.array has been replaced by getTupleForDataPoint() which returns a python tuple containing the components of the data point. In the case of matrices or higher ranked data, the tuples will be nested. Use numpy.array(data.getTupleForDataPoint()) if a numpy.ndarray object is required.
  - getValueOfGlobalDataPoint has similarly been replaced by getTupleForGlobalDataPoint().
  - integrate(data) now returns a numpy.ndarray instead of a numarray.array.
  Any python methods which previously accepted numarray objects will accept numpy objects instead.
- The way solver options are defined for LinearPDE objects has been changed. There is now a SolverOptions object attached to the LinearPDE object which is handling the options of solvers used to solve the PDE. The following changes apply:
- The `setTolerance` and `setAbsoluteTolerance` methods have been removed. Use now `getSolverOptions().setTolerance` and `getSolverOptions().setAbsoluteTolerance`.

- The `setSolverPackage` and `setSolverMethod` methods have been removed. Use now `getSolverOptions().setPackage`, `getSolverOptions().setSolverMethod` and `getSolverOptions().setPreconditioner`.

- The static class variables defining packages, solvers and preconditioners have been removed and are now accessed via the corresponding static class variables in `SolverOptions`. For instance use `SolverOptions.PCG` instead of `LinearPDE.PCG` to select the preconditioned conjugate gradient method.

- The `getSolution` takes now no argument. Use the corresponding methods of the `SolverOptions` object returned by `getSolverOptions()` to set values, e.g. use `getSolverOptions().setVerbosityOn()` instead of argument `verbose=True` and `getSolverOptions().setIterMax(1000)` instead of argument `iter_max=1000`.

- The `esys.pyvisi` module from previous releases has been deprecated and will no longer be supported. It is still present in the source code and can still be used if you compile `esys.escript` from source. It will not be available in binary releases. Its use is discouraged. The documentation for `esys.pyvisi` can be found in Appendix B.
A.3 escript references

If you use escript in your research we would appreciate a citation (of course we do not require this). Possible references include:

```latex
@InProceedings{lazyauspdc,  
  author = {Joel Fenwick and Lutz Gross},  
  title = {Lazy Evaluation of PDE Coefficients in the EScript System},  
  booktitle = {Parallel and Distributed Computing 2010 (AusPDC2010)},  
  pages = {71--76},  
  year = {2010},  
  editor = {Jinjun Chen and Rajiv Ranjan},  
  volume = {107},  
  series = {Conferences in Research and Practice in Information Technology},  
  month = {January},  
  issn = {1445-1336}
}

@article{GROSS2006,  
  author = {L. Gross and L. Bourgouin and A. J. Hale and H.-B Muhlhaus},  
  title = {Interface Modeling in Incompressible Media using Level Sets in Escript},  
  journal = {Physics of the Earth and Planetary Interiors},  
  year = {2007},  
  volume = {163},  
  pages = {23--34},  
  month = {Aug.},  
  doi = {doi:10.1016/j.pepi.2007.04.004},
}

@article{GROSS2007,  
  author = {L. Gross and B. Cumming and K. Steube and D. Weatherley},  
  title = {A Python Module for PDE-Based Numerical Modelling},  
  journal = {PARA},  
  year = {2007},  
  volume = {4699},  
  pages = {270--279},  
  doi = {doi:10.1007/978-3-540-75755-9},
}
```
The Module esys.pyvisi

Warning: The Module esys.pyvisi is no longer supported and will be removed from future releases.
Warning: The Module esys.pyvisi is not supported under MPI.

B.1 Introduction

esys.pyvisi is a Python module that is used to generate 2D and 3D visualizations for escript and its PDE solver finley. The module provides an easy to use interface to the 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 Scene instance - a window in which objects will be rendered on.
2. Create a data input instance (i.e. DataCollector or ImageReader) - reads the source data for visualization.
3. Create a data visualization object (i.e. Map, Velocity, Ellipsoid, Contour, Carpet, StreamLine, etc.) - creates a visual representation of the source data.
4. Create a Camera or 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)

\[ \text{scene} \rightarrow \text{data input} \rightarrow \text{data visualization} \rightarrow \text{camera/light} \rightarrow \text{render} \rightarrow \text{movie} \]

B.2 esys.pyvisi Classes

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

B.2.1 Scene Classes

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

```python
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:

- `setBackground(color)`
  Set the background color of the scene.

- `render(image_name = None)`
  Render the object using either the Online, Offline or Display mode.

Camera class

```python
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

```python
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.

B.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.
B.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 B.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
Map(scene, data_collector, viewport = Viewport.SOUTH_WEST, lut = Lut.COLOR, cell_to_point = False, outline = True)
```

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 and DataSetMapper.

A typical usage of Map is shown below.

```python
# Import the necessary modules.
from esys.pyvisi import Scene, DataCollector, Map, Camera
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 = "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.
dc1 = DataCollector(source = Source.XML)
dc1.setFileName(file_name = os.path.join(PYVISI_EXAMPLE_MESHES_PATH, FILE_3D))
dc1.setActiveScalar(scalar = SCALAR_FIELD_POINT_DATA)

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

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

# 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_collector, 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.TWO_D, lut = Lut.COLOR, cell_to_point = False, outline = True)
vopc1.setScaleFactor(scale_factor = 0.5)
vopc1.setPlaneToXY(offset = 0.5)
vopc1.setPlaneToXY_ratio(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, 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
# 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.
dcl = DataCollector(source = Source.XML)
dcl.setFileName(file_name = os.path.join(PYVISI_EXAMPLE_MESHES_PATH, FILE_3D))
dcl.setActiveTensor(tensor = TENSOR_FIELD_CELL_DATA)

# Create an EllipsoidOnPlaneClip.
eopc1 = EllipsoidOnPlaneClip(scene = s, data_collector = dcl,
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))

Contour class

```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.

```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 = 400

SCALAR_FIELD_POINT_DATA = "temperature"
FILE_3D = "interior_3D.xml"
IMAGE_NAME = "contour.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 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_POINT_DATA)

# Create three contours.
ctr1 = Contour(scene = s, data_collector = dc1, 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

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

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

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_sample_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

122 Appendix B. The Module esys.pyvisi
# 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.
dcl = DataCollector(source = Source.XML)
dcl.setFileName(file_name = os.path.join(PYVISI_EXAMPLE_MESHES_PATH, FILE_3D))
dcl.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.

```python
# Import the necessary modules.
from esys.pyvisi import Scene, DataCollector, Map, ImageReader, Image, Camera
from esys.pyvisi import GlobalPosition
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"
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 'ppmtopeg' 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")

# 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 + "000949.jpg")
```

B.2. esys.pyvisi Classes
B.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.

B.2.5 Supporting Classes

This subsection details the supporting classes and their corresponding methods inherited by the input (see Section B.2.2) and data visualization classes (see Section B.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** *(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** *(resolution)*
Set the resolution of the sweep for the rotation, which controls the number of intermediate points.

**setAngle** *(angle)*
Set the angle of rotation.

## B.3 More Examples

This section provides examples for some common tasks.

### B.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.
dcl = DataCollector(source = Source.XML)
dcl.setActiveScalar(scalar = SCALAR_FIELD_POINT_DATA_1)

# Create a Contour.
mosc1 = Contour(scene = s, data_collector = dcl,
viewport = 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
```

---

B.3. More Examples 131
# 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)

B.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
# 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)
```
B.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
import os

PYVISI_EXAMPLE_IMAGES_PATH = "data_sample_images"
X_SIZE = 400
Y_SIZE = 400
JPG_RENDERER = Renderer.ONLINE_JPG

#... set some parameters ...
x_c = [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 ...
mpde = LinearPDE(mydomain)
mpde.setSymmetryOn()
mpde.setValue(A=kappa*kronecker(mydomain), D=rhocp/h, d=eta, y=eta*Tref)
# set heat source: ...
x = mydomain.getX()
qH = qc*whereNegative(length(x-x_c)-r)

# set initial temperature ....
T=Tref

# Create a Scene.
s = Scene(renderer = JPG_RENDERER, x_size = X_SIZE, y_size = Y_SIZE)
```
# Create a DataCollector reading directly from escript objects.
dc = DataCollector(source = Source.EScript)

# 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)
B.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 B.1: Useful keys in Online render mode
B.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
* 42
** 42
+ 42
- 42
/ 42

___eq___() (Domain method), 40
___eq___() (FunctionSpace method), 41
___ne___() (Domain method), 40
___ne___() (FunctionSpace method), 41
___str___() (Data method), 44
___str___() (Domain method), 40, 41

A (data in ), 52
acceptConvergenceFailure() (SolverOptions method), 66
acos() (in module esys.escript), 48
acosh() (in module esys.escript), 49
adaptInnerTolerance() (SolverOptions method), 66
addFault() (FaultSystem method), 95
addItem() (PropertySet method), 78
addItem() (Design method), 80
AGGREGATION_COARSENING (SolverOptions attribute), 68
algebraic Multi-grid, 59, 64, 65, 68
AMG, 59, 64, 65, 68
AMG (SolverOptions attribute), 67
Arc (class in esys.pycad), 76
asin() (in module esys.escript), 48
asinh() (in module esys.escript), 49
atan() (in module esys.escript), 48
atanh() (in module esys.escript), 49
atm (data in ), 53
atmosphere, 52
Atto (data in ), 54
azimuth() (Camera method), 114

backView() (Camera method), 114
backward Euler, 9
BezierCurve (class in esys.pycad), 76
BICGSTAB (SolverOptions attribute), 67
BiCGStab, 67, 105
boldOn() (Text2D method), 115
bottomView() (Camera method), 114
boundary condition
natural, 10, 20, 57
boundary conditions
periodic, 100
boundary value problem, 2
BVP, 2, 4
Brick() (in module ), 107
BSpline (class in esys.pycad), 76
C (data in ), 53
Camera (class in ), 114
Carpet (class in ), 122
Celsius, 52
Celsius (data in ), 53
Centi (data in ), 54
CGNS (Design attribute), 79
CGS (SolverOptions attribute), 67
characteristic function, 4, 5, 10, 57
checkSymmetry() (LinearPDE method), 60
CHOLESKY (SolverOptions attribute), 67
clearItems() (Design method), 80
clearItems() (PropertySet method), 78
clip() (in module esys.escript), 46
close() (FileWriter method), 54
closed (FileWriter attribute), 54
cm (data in ), 52
cohesion factor, 89
costat, 10, 20, 57
contact conditions, 99
ContinuousFunction() (in module esys.escript), 41
Contour (class in ), 120
ContourOnPlaneClip (class in ), 121
ContourOnPlaneCut (class in ), 121
cos() (in module esys.escript), 48
cosh() (in module esys.escript), 49
Courant condition, 17
CSV, 50
CurveLoop (class in esys.pycad), 75
Darcy flow, 86
Darcy flux, 86, 88
DarcyFlow (class in ), 88
data (class in esys.escript), 43
data sample
points, 36, 37, 40–46, 95
dataCollector (class in ), 115
day (data in ), 52
Deca (data in ), 53
Deci (data in ), 53
DEFAULT (SolverOptions attribute), 66
DEFAULT_REORDERING (SolverOptions attribute), 68
DEG (data in esys.pycad), 78
DELAUNAY (Design attribute), 79
Design (class in esys.pycad.gmsh), 79
DIFFPACK (Design attribute), 79
diffusion equation, 8
Dilation (class in esys.pycad), 78
dip, 92
DIRECT (SolverOptions attribute), 67
Dirichlet boundary condition, 4
  homogeneous, 2, 5
discontinuity, 36, 57, 58
displacement, 24
div() (in module esys.escript), 48
dolly() (Camera method), 114
Domain (class in esys.escript), 39
Druck-Prager, 89
dump() (Data method), 44
dump() (Domain method), 40
eigenvalues() (in module esys.escript), 46
eigenvalues_and_eigenvectors() (in module esys.escript), 46
element, 97
  contact, 99, 107
  face, 98
  reference number, 97
elevation() (Camera method), 114
Ellipsoid (class in ), 119
EllipsoidOnPlaneClip (class in ), 119
EllipsoidOnPlaneCut (class in ), 119
empty Data, 43, 44
Environment
  ESCRIPT_HOSTFILE, 32
  ESCRIPT_NUM_NODES, 32
  ESCRIPT_NUM_PROCS, 32
  ESCRIPT_NUM_THREADS, 32
  ESCRIPT_STDFILES, 32, 33
  MPL_COMM_WORLD, 55
  OMP_NUM_THREADS, 3
  PATH, 31, 32
esys.escript (extension module), 39
esys.escript.linearPDEs (extension module), 59
esys.escript.pdetools (extension module), 61
esys.pycad (extension module), 75
esys.pycad.gmsh (extension module), 78
esys.pyvisi (extension module), 113
Exa (data in ), 53
exp() (in module esys.escript), 49
explicit scheme, 17
  Courant condition, 17
F (data in ), 53
Fahrenheit (data in ), 53
fault, 24
faults, 91
FaultSystem (class in ), 94
FEM
  elements, 97
  isoparametrical, 97
  mesh, 97
Femto (data in ), 54
FileWriter (class in ), 54
finite element method, 2, 3
  element, 3
  FEM, 2
  mesh, 3
  nodes, 2
finley
  Hex20, 99, 107
  Hex20Face, 99, 107
  Hex20Face_contact, 99
  Hex20Macro, 107
  Hex27, 99
  Hex27Macro, 99
  Hex8, 99, 107
  Hex8Face, 99, 107
  Hex8Face_contact, 99
  Line2, 98, 99, 106
  Line2_contact, 99
  Line2Face, 99
  Line2Face_contact, 99
  Line3, 99, 106
  Line3_contact, 99
  Line3Face, 99
  Line3Face_contact, 99
  Line3Macro, 99, 103, 106
  Point1, 99
  Point1_contact, 99
  Rec4, 98, 99, 106, 107
  Rec4_contact, 99
  Rec4Face, 99, 106
  Rec4Face_contact, 99
  Rec8, 99, 106, 107
  Rec8_contact, 99
  Rec8Face, 99, 106
  Rec8Face_contact, 99
  Rec8Macro, 106, 107
  Rec9, 99
  Rec9Face, 99
  Rec9Face_contact, 99
  Rec9Macro, 99, 104
  Tet10, 99
  Tet10Face, 99
  Tet10Face_contact, 99
  Tet10Macro, 99, 103
  Tet4, 99
  Tet4Face, 99
  Tet4Face_contact, 99
  Tri3, 97–99
  Tri3Face, 98, 99
  Tri3Face_contact, 99
Index 141

Tri6, 99
Tri6_CoOntact, 99
Tri6Face, 99
Tri6Face_CoOntact, 99
Tri6Macro, 99
Tri9, 99
Tri9_CoOntact, 99

finley (extension module), 97
flush() (FileWriter method), 54
flux, 60
force, internal, 81
Function (in module esys.escript), 41
FunctionOnBoundary (in module esys.escript), 41
FunctionOnContactOne (in module esys.escript), 41
FunctionOnContactZero (in module esys.escript), 41
FunctionSpace (class in esys.escript), 40

Gauss-Seidel Scheme, 65
GAUSS_SEIDEL (SolverOptions attribute), 67
generalized minimal residual method
GMRES, 85
generate() (Design method), 79
generateContours() (ContourModule method), 127
getAbsoluteTolerance() (SolverOptions method), 65
getAbsoluteTolerance() (StokesProblemCartesian method), 85
getBottomPolyline() (FaultSystem method), 94
centerOnSurface() (FaultSystem method), 94
getCoarsening() (SolverOptions method), 64
getCoarseningThreshold() (SolverOptions method), 65
getcCoefficient() (LinearPDE method), 59
getCommandString() (Design method), 80
getDepths() (FaultSystem method), 94
getDeviatoricStrain() (IncompressibleIsotropicFlowCartesian method), 91
getDeviatoricStress() (IncompressibleIsotropicFlowCartesian method), 91
getDiagnostics() (SolverOptions method), 63
getDim() (Design method), 79
getDim() (Domain method), 40
getDim() (FaultSystem method), 94
getDim() (FunctionSpace method), 41
getDim() (LinearPDE method), 60
getDim() (PropertySet method), 78
getDips() (FaultSystem method), 94
getDomain() (Data method), 43
getDomain() (FunctionSpace method), 41
getDomain() (IncompressibleIsotropicFlowCartesian method), 90
getDomain() (LinearPDE method), 60

getDropStorage() (SolverOptions method), 66
getDropTolerance() (SolverOptions method), 65
getElemenofDistribution() (Arc method), 76
getElemenofDistribution() (BSpline method), 76
getElemenofDistribution() (BezierCurve method), 76
getElemenofDistribution() (Line method), 75
getElemenofDistribution() (Spline method), 75
ggetElementOrder() (Design method), 79
ggetElementSize() (Design method), 79
gescriptParamInt() (in module ), 55
getFileFormat() (Design method), 80
getFlux() (LinearPDE method), 60
getFunctionSpace() (Data method), 43
getFunctionSpaceForCoefficient() (LinearPDE method), 59
getGammaDot() (IncompressibleIsotropicFlowCartesian method), 91
getInnerIterMax() (SolverOptions method), 66
getInnerTolerance() (SolverOptions method), 65
getItems() (Design method), 80
getItems() (PropertySet method), 78
getIte() (SolverOptions method), 78
getLengths() (FaultSystem method), 94
getLevelMax() (SolverOptions method), 64
getMaxValue() (FaultSystem method), 95
getMediumDepth() (FaultSystem method), 94
getMeshHandler() (Design method), 80
getMinCoarseMatrixSize() (SolverOptions method), 65
getMinValue() (FaultSystem method), 95
getMPIRank() (Domain method), 40
getMPIRankWorld() (in module ), 55
getMPISize() (Domain method), 40
getMPISizeWorld() (in module ), 55
getMPIWorldMax() (in module ), 55
getName() (PropertySet method), 78
getName() (SolverOptions method), 63
getNormal() (Domain method), 40
getNormal() (FunctionSpace method), 41
getNumEquations() (LinearPDE method), 60
getNumSweeps() (SolverOptions method), 65
getNumSweeps() (SolverOptions method), 65
getNumSweeps() (SolverOptions method), 65
getOrientationOnSurface() (FaultSystem method), 94
getPackage() (SolverOptions method), 63
getParametrization() (FaultSystem method), 95
getPreconditioner() (SolverOptions method), 63
getPressure() (IncompressibleIsotropicFlowCartesian method), 91
getRank() (Data method), 44
getRelaxationFactor() (SolverOptions method), 66
getReordering() (SolverOptions method), 64
getRestart() (SolverOptions method), 64
getRightHandSide() (LinearPDE method), 60
getScriptFileName() (Design method), 80
getScriptString() (Design method), 80
getSegmentNormals() (FaultSystem method), 94
getShape() (Data method), 43
getShapeOfCoefficient() (LinearPDE method), 59
getSideAndDistance() (FaultSystem method), 95
getSize() (Domain method), 40
getSize() (FunctionSpace method), 41
getSmoothen() (SolverOptions method), 65
getSolution() (LinearPDE method), 60
g getSolverMethod() (SolverOptions method), 63
g getSolverOptions() (LinearPDE method), 59
g getSolverOptions() (Projector method), 62
g getSolverOptionsDiv() (StokesProblemCartesian method), 85
g getSolverOptionsFlux() (DarcyFlow method), 88
g getSolverOptionsPressure() (DarcyFlow method), 88
g getSolverOptionsPressure() (StokesProblemCartesian method), 85
g getSolverOptionsVelocity() (StokesProblemCartesian method), 85
g Gets() (FaultSystem method), 94
getStress() (IncompressibleIsotropicFlowCartesian method), 91
getStress() (IncompressibleIsotropicFlowCartesian method), 91
getStress() (IncompressibleIsotropicFlowCartesian method), 91
getStress() (IncompressibleIsotropicFlowCartesian method), 91
getStress() (IncompressibleIsotropicFlowCartesian method), 91
getStress() (IncompressibleIsotropicFlowCartesian method), 91
getStress() (IncompressibleIsotropicFlowCartesian method), 91
getStress() (IncompressibleIsotropicFlowCartesian method), 91
getStress() (IncompressibleIsotropicFlowCartesian method), 91
getStress() (IncompressibleIsotropicFlowCartesian method), 91
getStress() (IncompressibleIsotropicFlowCartesian method), 91
getStress() (IncompressibleIsotropicFlowCartesian method), 91
getStress() (IncompressibleIsotropicFlowCartesian method), 91
getStress() (IncompressibleIsotropicFlowCartesian method), 91
getStress() (IncompressibleIsotropicFlowCartesian method), 91
getStress() (IncompressibleIsotropicFlowCartesian method), 91
getStress() (IncompressibleIsotropicFlowCartesian method), 91
getStress() (IncompressibleIsotropicFlowCartesian method), 91
getStress() (IncompressibleIsotropicFlowCartesian method), 91
getStress() (IncompressibleIsotropicFlowCartesian method), 91
getStress() (IncompressibleIsotropicFlowCartesian method), 91
getStress() (IncompressibleIsotropicFlowCartesian method), 91
getStress() (IncompressibleIsotropicFlowCartesian method), 91
getStress() (IncompressibleIsotropicFlowCartesian method), 91
getStress() (IncompressibleIsotropicFlowCartesian method), 91
getStress() (IncompressibleIsotropicFlowCartesian method), 91
getStress() (IncompressibleIsotropicFlowCartesian method), 91
getStress() (IncompressibleIsotropicFlowCartesian method), 91
getStress() (IncompressibleIsotropicFlowCartesian method), 91
getStress() (IncompressibleIsotropicFlowCartesian method), 91
getStress() (IncompressibleIsotropicFlowCartesian method), 91
getStress() (IncompressibleIsotropicFlowCartesian method), 91
getStress() (IncompressibleIsotropicFlowCartesian method), 91
getStress() (IncompressibleIsotropicFlowCartesian method), 91
getStress() (IncompressibleIsotropicFlowCartesian method), 91
getStress() (IncompressibleIsotropicFlowCartesian method), 91
getStress() (IncompressibleIsotropicFlowCartesian method), 91
getStress() (IncompressibleIsotropicFlowCartesian method), 91
getStress() (IncompressibleIsotropicFlowCartesian method), 91
getStress() (IncompressibleIsotropicFlowCartesian method), 91
getStress() (IncompressibleIsotropicFlowCartesian method), 91
getStress() (IncompressibleIsotropicFlowCartesian method), 91
getStress() (IncompressibleIsotropicFlowCartesian method), 91
getStress() (IncompressibleIsotropicFlowCartesian method), 91
getStress() (IncompressibleIsotropicFlowCartesian method), 91
getStress() (IncompressibleIsotropicFlowCartesian method), 91
getStress() (IncompressibleIsotropicFlowCartesian method), 91
getStress() (IncompressibleIsotropicFlowCartesian method), 91
getStress() (IncompressibleIsotropicFlowCartesian method), 91
getStress() (IncompressibleIsotropicFlowCartesian method), 91
getStress() (IncompressibleIsotropicFlowCartesian method), 91
getStress() (IncompressibleIsotropicFlowCartesian method), 91
getStress() (IncompressibleIsotropicFlowCartesian method), 91
getStress() (IncompressibleIsotropicFlowCartesian method), 91
getStress() (IncompressibleIsotropicFlowCartesian method), 91
getStress() (IncompressibleIsotropicFlowCartesian method), 91
getStress() (IncompressibleIsotropicFlowCartesian method), 91
getStress() (IncompressibleIsotropicFlowCartesian method), 91
getStress() (IncompressibleIsotropicFlowCartesian method), 91
getStress() (IncompressibleIsotropicFlowCartesian method), 91
getStress() (IncompressibleIsotropicFlowCartesian method), 91
getStress() (IncompressibleIsotropicFlowCartesian method), 91
getStress() (IncompressibleIsotropicFlowCartesian method), 91
getStress() (IncompressibleIsotropicFlowCartesian method), 91
getStress() (IncompressibleIsotropicFlowCartesian method), 91
getStress() (IncompressibleIsotropicFlowCartesian method), 91
getStress() (IncompressibleIsotropicFlowCartesian method), 91
getStress() (IncompressibleIsotropicFlowCartesian method), 91
getStress() (IncompressibleIsotropicFlowCartesian method), 91
getStress() (IncompressibleIsotropicFlowCartesian method), 91
getStress() (IncompressibleIsotropicFlowCartesian method), 91
getStress() (IncompressibleIsotropicFlowCartesian method), 91
getStress() (IncompressibleIsotropicFlowCartesian method), 91
getStress() (IncompressibleIsotropicFlowCartesian method), 91
getVelocity() (IncompressibleIsotropicFlowCartesian method), 91
getW0Offsets() (FaultSystem method), 94
g e t W 0 R a n g e () ( F a u l t S y s t e m m e t h o d ), 94
g e t W 1 R a n g e () ( F a u l t S y s t e m m e t h o d ), 94
getX() (Domain method), 40
getX() (FunctionSpace method), 41
Giga (data in ), 53
GlobalPosition (class in ), 126
GlueFaces() (in module ), 107
GMRES, 64, 67
GMRES (SolverOptions attribute), 67
Gmsh (Design attribute), 79
Gmsh, 69–71, 77, 78, 106
gnuplot, 18
getValue() (Module esys.escript), 48
g getValue() (Module esys.escript), 48
g getValue() (Module esys.escript), 48
h (data in ), 52
hasConverged() (SolverOptions method), 64
Hecto (data in ), 53
Helmholtz (class in esys.escript.linearPDEs), 61
Helmholtz equation, 8, 10
Hz (data in ), 53
IDEAS (Design attribute), 79
identityTensor() (in module esys.escript), 44
identityTensor4() (in module esys.escript), 44
ILU0, 68
ILU0 (SolverOptions attribute), 67
ILUT, 65, 66
ILUT (SolverOptions attribute), 67
Image (class in ), 123
imageList() (Movie method), 125
imageRange() (Movie method), 125
ImageReader (class in ), 115
implicit scheme, 17
incompressible fluid, 81
IncompressibleIsotropicFlowCartesian (class in ), 90
inf() (in module esys.escript), 45
initialize() (StokesProblemCartesian method), 85
inner() (in module esys.escript), 46
interpolate() (in module esys.escript), 48
iteration order, 106, 107
interpolate() (in module esys.escript), 48
interpolateTable, 49
interpolation, 36, 62
inverse() (in module esys.escript), 46
isNotEmpty() (Data method), 44
isNotEmpty() (Operator method), 51
isotropic view() (Camera method), 114
isSymmetric() (LinearPDE method), 60
isSymmetric() (SolverOptions method), 66
isUsingLumping() (LinearPDE method), 60
isValidTagName() (Domain method), 40
isVerbose() (SolverOptions attribute), 67
ITERATIVE (SolverOptions attribute), 67
Index

Pa (data in ), 53
packages
MKL, 63, 66, 105, 106
PASO, 67, 105, 106
UMFPACK, 66, 105, 106
partial derivative, 2
partial differential equation, 1, 35, 41
PDE, 1, 3
partial differential equations, 97
PASO, 67, 105, 106
PASO (SolverOptions attribute), 67
PCG, 59, 67, 82, 87, 105
PCG (SolverOptions attribute), 67
periodic boundary conditions, 107
Peta (data in ), 53
Pico (data in ), 54
PlaneSurface (class in esys.pycad), 76
PLOT3D (Design attribute), 79
Point (class in esys.pycad), 75
Poisson, 59
Poisson (class in esys.escript.linearPDEs), 61
Poisson equation, 1–3
pounds, 52
preconditioned conjugate gradient method
PCG, 85
preconditioner, 82
Gauss-Seidel, 65
ILU0, 68
ILUT, 65, 66
Jacobi, 65, 67
RILU, 66
PRES20 (SolverOptions attribute), 67
projection, 36, 62
Projector (class in esys.escript.pdetools), 62
PropertySet (class in esys.pycad), 78
randomOn() (MaskPoints method), 129
rank, 44
ReadGmsh() (in module ), 106
ReadMesh() (in module ), 106
REC_ILU (SolverOptions attribute), 68
Rectangle () (in module ), 106
Rectangle (class in ), 123
ReducedSolution() (in module esys.escript), 41
Reflection (class in esys.pycad), 78
render () (Scene method), 114
resetDiagnostics() (SolverOptions method), 63
resetElementDistribution() (Arc method),
76
resetElementDistribution() (BSpline method), 76
resetElementDistribution() (BezierCurve method), 76
resetElementDistribution() (Line method),
75
resetElementDistribution() (Spline method), 75
rightView() (Camera method), 114
RILU, 66
RILU (SolverOptions attribute), 67
Rotation (class in esys.pycad), 78
rotateX() (Transform method), 128
rotateY() (Transform method), 129
rotateZ() (Transform method), 129
RUGE_STUEBEN_COARSENING (SolverOptions attribute), 68
RuledSurface (class in esys.pycad), 77
run-escript, 31
saddle point problems, 36
saveDataCSV, 50
saveDX() (in module esys.escript), 44
saveMM() (Operator method), 51
saveVTK() (in module esys.escript), 44
Scalar() (in module esys.escript), 43
Scene (class in ), 114
SciPy, 19
scripts
‘diffusion.py’, 12, 21
‘helmholtz.py’, 11, 85
‘wave.py’, 18
sec (data in ), 52
setAbsoluteTolerance() (DarcyFlow method),
88
setAbsoluteTolerance() (SolverOptions method), 65
setAbsoluteTolerance() (StokesProblemCartesian method), 85
setAcceptanceConvergenceFailureOff() (SolverOptions method), 66
setAcceptanceConvergenceFailureOn() (SolverOptions method), 66
setActiveScalar() (DataCollector method), 115
setActiveTensor() (DataCollector method), 115
setActiveVector() (DataCollector method), 115
setAngle() (Light method), 115
setAngle() (Rotation method), 131
setBackground() (Scene method), 114
setCenter() (CubeSource method), 130
setClipValue() (Clipper method), 127
setCoarsening() (SolverOptions method), 64
setCoarseningThreshold() (SolverOptions method), 64
setColor() (Actor3D method), 126
setColor() (Light method), 114
setColor() (Text2D method), 115
setData() (DataCollector method), 115
setDebugOff() (LinearPDE method), 59
setDebugOn() (LinearPDE method), 59
setDim() (Design method), 79
setDropStorage() (SolverOptions method), 65
setDropTolerance() (SolverOptions method), 65
druckerpragerLaw() (IncompressibleIsotropicFlowCartesian method), 91

144 Index
setElasticShearModulus() (IncompressibleIsotropicFlowCartesian method), 91
setElementDistribution() (Arc method), 76
setElementDistribution() (BSpine method), 76
setElementDistribution() (BezierCurve method), 76
setElementDistribution() (Line method), 75
setElementDistribution() (PlaneSurface method), 76
setElementDistribution() (RuledSurface method), 77
setElementDistribution() (Spline method), 75
setElementDistribution() (Volume method), 77
setElementOrder() (Design method), 79
setElementSize() (Design method), 79
setEscriptParamInt() (in module ), 55
setEtaTolerance={() (IncompressibleIsotropicFlowCartesian method), 91
setFileFormat() (Design method), 80
setName() (PropertySet method), 78
setNumPostSweeps() (SolverOptions method), 65
setNumPreSweeps() (SolverOptions method), 65
setOpacity() (Actor3D method), 126
setOptions() (Design method), 80
setOrientationToHorizontal() (ScalarBar method), 130
setOrientationToVertical() (ScalarBar method), 130
setOrigin() (PlaneSource method), 127
setPackage() (SolverOptions method), 63
setPhiResolution() (Sphere method), 128
setPlaneToXY() (Transform method), 129
setPlaneToXZ() (Transform method), 129
setPlaneToYZ() (Transform method), 129
setPoint1() (PlaneSource method), 128
setPoint2() (PlaneSource method), 128
setPointSourceCenter() (PointSource method), 128
setPointSourceNumberOfPoints() (PointSource method), 128
setPointSourceRadius() (PointSource method), 128
setPosition() (Actor2D method), 126
setPosition() (Camera method), 114
setPosition() (Light method), 115
setPosition() (ScalarBar method), 130
setPowerLaws() (IncompressibleIsotropicFlowCartesian method), 91
setPreconditioner() (SolverOptions method), 63
setRatio() (MaskPoints method), 129
setRecombination() (PlaneSurface method), 76
setRecombination() (RuledSurface method), 77
setRecombination() (Volume method), 77
setReducedOrderOff() (LinearPDE method), 60
setReducedOrderOn() (LinearPDE method), 60
setRelaxationFactor() (SolverOptions method), 66
setReordering() (SolverOptions method), 64
setScaleFactor() (Warp method), 129
setScaleFactor() (TensorGlyph method), 127
setScaleFactor() (Glyph3D method), 127
setScaleModeByScalar() (Glyph3D method), 127
setScaleModeByVector() (Glyph3D method), 127
setScriptFileName() (Design method), 80
setScriptFileName() (Data method), 44
setScriptParamInt() (in module ), 55
setSize() (ImageReslice method), 130
setSymmetryOff() (LinearPDE method), 60
setSymmetryOn() (SolverOptions method), 66
setTaggedValue() (Data method), 44
setTagMap() (Domain method), 40
setTags() (FunctionSpace method), 41
setThetaResolution() (Sphere method), 128
setTitle() (ScalarBar method), 130
setTitleColor() (ScalarBar method), 130
setThetaResolution() (IncompressibleIsotropicFlowCartesian method), 91
setTolerance() (SolverOptions method), 65
setTolerance() (StokesProblemCartesian method), 85
setTransfiniteMeshing() (PlaneSurface method), 77
setTransfiniteMeshing() (RuledSurface method), 77
setTolerance() (Volume method), 77
setTruncation() (SolverOptions method), 64
setTubeRadius() (Tube method), 129
setTubeRadiusToVaryByScalar() (Tube method), 129
setTubeRadiusToVaryByVector() (Tube method), 129
setValue() (DarcyFlow method), 88
setValue() (Helmholtz method), 61
setValue() (Lame method), 61
setValue() (LinearPDE method), 59
setValue() (Operator method), 51
setValue() (Poisson method), 61
setVerbosityOff() (SolverOptions method), 66
setVerbosityOn() (SolverOptions method), 66
setWidth() (ScalarBar method), 130
setX() (Domain method), 40
setXLength() (CubeSource method), 130
setYLength() (CubeSource method), 130
setZLength() (CubeSource method), 131
shape, 4, 36, 37, 40, 42, 43
SI units, 51
sign() (in module esys.escript), 49
sin() (in module esys.escript), 48
sinh() (in module esys.escript), 48
slicing, 42
slip, 24
Solution() (in module esys.escript), 41
solution, 17, 41, 61, 62
reduced, 41, 62, 97
solve() (DarcyFlow method), 88
solve() (StokesProblemCartesian method), 85
SolverOptions (class in ), 62
solves() (Operator method), 51
Spline (class in esys.pycad), 75
sqrt() (in module esys.escript), 49
SSOR (SolverOptions attribute), 67
STANDARD COARSENING (SolverOptions attribute), 68
STL (Design attribute), 79
Stokes problem, 81, 84
StokesProblemCartesian (class in ), 84
StreamLine (class in ), 121
stress, 14, 24
stress, initial, 81
strike, 91, 92
summation convention, 2, 9
sup() (in module esys.escript), 45
SUPER LU (SolverOptions attribute), 68
SurfaceLoop (class in esys.pycad), 75
swap_axes() (in module esys.escript), 45
symmetric() (in module esys.escript), 45
symmetric PDE, 11, 21
symmetrical, 58
tag, 78
tagging, 38, 70
tan() (in module esys.escript), 48
tanh() (in module esys.escript), 49
Tensor() (in module esys.escript), 43
Tensor3() (in module esys.escript), 43
Tensor4() (in module esys.escript), 43
tensor_mult() (module esys.escript), 47
tensor_transposed_mult() (module esys.escript), 47
Tera (data in ), 53
TETGEN (Design attribute), 79
Text2D (class in ), 115
TFQMR, 67
TFQMR (SolverOptions attribute), 67
time integration
explicit, 17
implicit, 17
ton (data in ), 52
topView() (Camera method), 114
trace() (in module esys.escript), 45
transform() (FaultSystem method), 94
translate() (Transform method), 128
Translation (class in esys.pycad), 77
transpose() (module esys.escript), 45
transposed_matrix_mult() (module esys.escript), 47
transposed_tensor_mult() (module esys.escript), 47
TRILINOS (SolverOptions attribute), 68
UMFPACK, 66, 105, 106
UMFPACK (SolverOptions attribute), 66
unitVector() (in module esys.escript), 45
update() (IncompressibleIsotropicFlowCartesian method), 91
Uzawa scheme, 82
V (data in ), 53
Vector() (in module esys.escript), 43
Velocity (class in ), 117
velocity, 81
VelocityOnPlaneClip (class in ), 119
VelocityOnPlaneCut (class in ), 118
Verlet scheme, 15
VisIt, 5, 7
visualization
  gnuplot, 18
  matplotlib, 5–7, 19
  mayavi, 5, 8, 24
  OpenDX, 44
  VisIt, 5, 7
  VTK, 5, 7, 13, 21, 44, 113
Volume (class in esys.pycad), 77
von–Mises stress, 21
VRML (Design attribute), 79
VTK, 5, 7, 13, 21, 44, 113

W (data in ), 53
wave equation, 13
whereNegative() (in module esys.escript), 49
whereNonNegative() (in module esys.escript), 49
whereNonPositive() (in module esys.escript), 49
whereNonZero() (in module esys.escript), 49
wherePositive() (in module esys.escript), 49
whereZero() (in module esys.escript), 49
write() (FileWriter method), 54
writelines() (FileWriter method), 54

YAIR_SHAPIRA_COARSENING (SolverOptions attribute), 68
yield condition, 89
Yocto (data in ), 54
Yotta (data in ), 53
yr (data in ), 52
Zepto (data in ), 54
Zetta (data in ), 53
[22] Intel’s math kernel library.


[34] VisIt. https://wci.llnl.gov/codes/visit/home.html.


