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 B.

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 D.
Researchers and Developers

Escript is the product of years of work by many people. The active researchers for the current release series (3.X) are listed here in alphabetical order. While development is collaborative, each person is listed with some of their major contributions — this list is not exhaustive. Personnel for previous release series are listed in an appendix of the user guide.

Cihan Altinay   esys.weipa visualisation package, SCons build system rework.
Artak Amirbekyan  Solvers, OSX work.
Joel Fenwick   Lazy evaluation, maintenance of escript module, release wrangler.
Lin Gao        Performance analysis.
Lutz Gross     Patriarch, technical lead, solvers, large chunks of the original code.
## Contents

1  Tutorial: Solving PDEs  
   1.1  Installation ...................................................... 9  
   1.2  The First Steps .................................................. 9  
       1.2.1  Plotting Using matplotlib .................................. 13  
       1.2.2  Visualization using export files .......................... 15  
   1.3  The Diffusion Problem ........................................... 16  
       1.3.1  Outline ..................................................... 16  
       1.3.2  Temperature Diffusion ...................................... 16  
       1.3.3  Helmholtz Problem .......................................... 17  
       1.3.4  The Transition Problem .................................... 19  
   1.4  Elastic Deformation .............................................. 28  
   1.5  Stokes Flow ..................................................... 30  
   1.6  Slip on a Fault .................................................. 32  

2  Execution of an escript Script  
   2.1  Overview ....................................................... 37  
   2.2  Options ........................................................ 38  
       2.2.1  Notes ....................................................... 38  
   2.3  Input and Output ............................................... 39  
   2.4  Hints for MPI Programming .................................... 39  
   2.5  Lazy Evaluation ................................................ 40  

3  The esys.escript Module  
   3.1  Concepts ....................................................... 41  
       3.1.1  Function spaces ............................................ 41  
       3.1.2  Data Objects .............................................. 43  
       3.1.3  Tagged, Expanded and Constant Data ....................... 44  
       3.1.4  Saving and Restoring Simulation Data ...................... 45  
   3.2  esys.escript Classes ........................................... 46  
       3.2.1  The Domain class .......................................... 46  
       3.2.2  The FunctionSpace class ................................... 47  
       3.2.3  The Data Class ............................................ 49  
       3.2.4  Generation of Data objects ................................ 50  
       3.2.5  Data methods .............................................. 51  
       3.2.6  Functions of Data objects ................................ 51  
       3.2.7  Interpolating Data ......................................... 58  
       3.2.8  The DataManager Class ..................................... 58  
       3.2.9  Saving Data as CSV ....................................... 60  
       3.2.10 The Operator Class ........................................ 61  
   3.3  Physical Units .................................................. 61  
   3.4  Utilities ........................................................ 64  

Contents 5
### 4 The `esys.escript.linearPDEs` Module

4.1 Linear Partial Differential Equations ........................................ 67
  4.1.1 Classes ................................................................. 67
  4.1.2 `LinearPDE` class .................................................... 69
  4.1.3 The `Poisson` Class ................................................ 71
  4.1.4 The `Helmholtz` Class ............................................ 71
  4.1.5 The `Lame` Class .................................................. 72

4.2 Projection ................................................................. 72

4.3 Solver Options ........................................................... 73

4.4 Some Remarks on Lumping .................................................... 79
  4.4.1 Scalar wave equation .............................................. 80
  4.4.2 Advection equation .............................................. 81
  4.4.3 Summary ............................................................... 83

### 5 The `esys.pycad` Module

5.1 Introduction ............................................................... 85

5.2 The Unit Square ............................................................ 85

5.3 Holes ................................................................. 87

5.4 A 3D example .............................................................. 88

5.5 Alternative File Formats .................................................. 89

5.6 Element Sizes ............................................................ 90

5.7 `esys.pycad` Classes ...................................................... 90
  5.7.1 Primitives ............................................................. 90
  5.7.2 Transformations .................................................... 94
  5.7.3 Properties ............................................................ 94

5.8 Interface to the mesh generation software ................................. 95

### 6 Models

6.1 The Stokes Problem ....................................................... 99
  6.1.1 Solution Method ..................................................... 99
  6.1.2 Functions ............................................................ 103
  6.1.3 Example: Lid-driven Cavity ..................................... 104

6.2 Darcy Flux ................................................................. 104
  6.2.1 Solution Method ..................................................... 104
  6.2.2 Functions ............................................................ 106

6.3 Isotropic Kelvin Material ................................................. 107
  6.3.1 Solution Method ..................................................... 108
  6.3.2 Functions ............................................................ 109

6.4 Fault System ............................................................. 110
  6.4.1 Functions ............................................................ 112
  6.4.2 Example .............................................................. 114

### 7 The `esys.finley` Module

7.1 Formulation ............................................................... 115

7.2 Meshes ................................................................. 115

7.3 Macro Elements ............................................................ 122

7.4 Linear Solvers in `SolverOptions` .................................... 122

7.5 Functions ................................................................. 122

### 8 The `esys.weipa` Module and Data Visualization

8.1 The `EscriptDataset` class ............................................... 125

8.2 Functions ................................................................. 126

8.3 Visualizing `escript` Data ............................................. 127
  8.3.1 Using the `VisIt` GUI ............................................. 127
  8.3.2 Using the `VisIt` CLI ............................................. 128

### A Einstein Notation

A Einstein Notation .......................................................... 129
1.1 Installation

To download *escript* and friends, please visit https://launchpad.net/escript-finley. The web site offers binary distributions for some platforms and provides 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 in 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

\[ -\Delta u = f \]  

(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

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

(1.2)

The domain is shown in Figure 1.1.

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

\[ \Delta u = (u, _0)_0 + (u, _1)_1 \]  

(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 compact 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, i.e. \( u \) shall fulfill the homogeneous Neumann boundary condition

\[n_i u_{,i} = 0\]  

(1.11)

\( n = (n_i) \) denotes the outer normal field of the domain, see Figure 1.1. Remember that we are applying the Einstein summation convention, i.e. \( n_i u_{,i} = n_0 u_{,0} + n_1 u_{,1} \). The Neumann boundary condition of Equation (1.11) should be fulfilled on the set \( \Gamma^N \) which is the top and right edge of the domain:

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

(1.12)

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

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

(1.13)

the solution shall be identical to 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 \( u \).

\footnote{You 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

\[-\nabla^2 u = f\]}

\footnote{Some readers may familiar with the notation \( \frac{\partial n}{\partial x} = n_i u_{,i} \) for the normal derivative.}
In general the BVP cannot be solved analytically and numerical methods have to be used to 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 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 their 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 [39, 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 it is more efficient to use scripts which you can edit with your favorite editor. To enter the escript environment, use the \texttt{run-escript} command\footnote{\texttt{run-escript} is not available under Windows yet. If you run under Windows you can just use the \texttt{python} command and the \texttt{OMP_NUM_THREADS} environment variable to control the number of threads.}:

\begin{verbatim}
run-escript
\end{verbatim}

which will pass you on to the python prompt

\begin{verbatim}
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.
>>>
\end{verbatim}

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

\begin{verbatim}
>>> x=2+3
>>> print "2+3=",x
2+3= 5
\end{verbatim}

We refer to the python user’s guide if you not familiar with python.

esys.escript provides the class \texttt{Poisson} to define a Poisson equation. (We will discuss a more general form of a PDE that can be defined through the \texttt{LinearPDE} class later.) The instantiation of a \texttt{Poisson} class object requires the specification of the domain $\Omega$. In esys.escript\texttt{} 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 \texttt{esys.finley}. The following statements create the Domain object \texttt{mydomain} from the \texttt{esys.finley} method \texttt{Rectangle}:

\begin{verbatim}
from esys.finley import Rectangle
mydomain = Rectangle(l0=1.,l1=1.,n0=40, n1=20)
\end{verbatim}

In this case the domain is a rectangle with the lower, left corner at point $(0,0)$ and the right, upper corner at $(10,11)=(1,1)$. The arguments $n0$ and $n1$ define the number of elements in $x_0$ and $x_1$-direction respectively. For more details on \texttt{Rectangle} and other Domain generators within the \texttt{esys.finley} module, see Chapter 7.
The following statements define the \texttt{Poisson} class object \texttt{mypde} with domain \texttt{mydomain} and the right hand side $f$ of the PDE to constant 1:

\begin{verbatim}
from esys.escript.linearPDEs import Poisson
mypde = Poisson(mydomain)
mypde.setValue(f=1)
\end{verbatim}

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

\begin{verbatim}
x=mydomain.getX()
\end{verbatim}

The method \texttt{getX} of the Domain \texttt{mydomain} gives access to locations in the domain defined by \texttt{mydomain}. The object $x$ is actually a \texttt{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 \texttt{getRank} and \texttt{getShape} methods:

\begin{verbatim}
print "rank ",x.getRank(),", shape ",x.getShape()
\end{verbatim}

This will print something like

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

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

\begin{verbatim}
print x.getFunctionSpace()
\end{verbatim}

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 Finley mesh. To get the $x_0$ coordinates of the locations we use the statement

\begin{verbatim}
x0=x[0]
\end{verbatim}

Object $x0$ is again a \texttt{Data} object now with rank 0 and shape (). It inherits the \texttt{FunctionSpace} from $x$:

\begin{verbatim}
print x0.getRank(), x0.getShape(), x0.getFunctionSpace()
\end{verbatim}

will print

\begin{verbatim}
0 () Function space type: Finley_Nodes on FinleyMesh
\end{verbatim}

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

\begin{verbatim}
from esys.escript import whereZero
gammaD=whereZero(x[0])+whereZero(x[1])
\end{verbatim}

whereZero($x[0]$) creates a function which equals 1 where $x[0]$ is (almost) equal to zero and 0 elsewhere. Similarly, whereZero($x[1]$) creates a 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 \texttt{mydomain} which is strictly positive where $x_0$ or $x_1$ is equal to zero. Note that $\gamma_D$ has the same rank, shape and \texttt{FunctionSpace} like $x0$ used to define it. So from

\begin{verbatim}
print gammaD.getRank(), gammaD.getShape(), gammaD.getFunctionSpace()
\end{verbatim}

one gets

\begin{verbatim}
0 () Function space type: Finley_Nodes on FinleyMesh
\end{verbatim}

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

```
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, e.g. calculating the gradient or the average value, which will be discussed later, plotting the solution is one of the things 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 converted to hand over the solution. The first option is using the matplotlib module which allows plotting 2D results relatively quickly from within the Python script, see [23]. However, there are limitations when using this tool, especially for large problems and when solving 3-dimensional problems. Therefore, esys.escript provides functionality to export data as files which can subsequently be read by third-party software packages such as mayavi [10] or VisIt [35].

### 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 mapped onto a rectangular grid. We will make use of the numpy module.

First we need to create a rectangular grid which is accomplished by the following statements:

```
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. Spatial coordinates are easily extracted as a list by

```
x=mydomain.getX()[0].toListOfTuples()
y=mydomain.getX()[1].toListOfTuples()
```

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 used when extracting \( x \) and \( y \). We apply the interpolation to \( u \) before extraction to achieve this:

```
z=interpolate(u, mydomain.getX().getFunctionSpace())
```

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

---

4 Users 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.
Now \( z_{\text{grid}} \) gives the values of the PDE solution \( u \) at the grid which can be plotted using `contourf`:

```python
import matplotlib
z_grid = matplotlib.mlab.griddata(x, y, z, xi=x_grid, yi=y_grid)
```

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
import matplotlib
matplotlib.pyplot.contourf(x_grid, y_grid, z_grid, 5)
matplotlib.pyplot.savefig("u.png")
```

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)
```

**Figure 1.3**: Visualization of the Poisson Equation Solution for \( f = 1 \) using matplotlib

![Poisson Equation Solution](image.png)
# 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 [23].

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 export files

As an alternative to matplotlib, escript supports exporting data to VTK and SILO files which can be read by visualization tools such as mayavi[10] and VisIt [35]. This method is MPI safe and works with large 2D and 3D problems.

To write the solution $u$ of the Poisson problem in the VTK file format to the file $u.vtu$ one needs to add:

```python
from esys.weipa import saveVTK
saveVTK("u.vtu", sol=u)
```

This file can then be opened in a VTK compatible visualization tool where the solution is accessible by the name $sol$. Similarly,

```python
from esys.weipa import saveSilo
saveSilo("u.silo", sol=u)
```

will write $u$ to a SILO file if escript was compiled with SILO support.

The Poisson problem script is now

```python
from esys.escript import *
from esys.escript.linearPDEs import Poisson
from esys.finley import Rectangle
from esys.weipa import saveVTK
```

$^5$The phrase ‘safe under MPI’ means that a program will produce correct results when run on more than one processor under MPI.
# 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.vtu",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.vtu -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 this solver 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

$$\rho c_p T_i, t - (\kappa T_i), x = q_H$$

(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

---

### Figure 1.5: Temperature Diffusion Problem with Circular Heat Source
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:
\[
q_H(x, t) = \begin{cases} 
q^c & \text{if } \|x - x^c\| \leq r \\
0 & \text{else}
\end{cases}
\]  
(1.16)
for all $x$ in the domain and time $t > 0$.

On the surface of the domain we specify a radiation condition which prescribes the normal component of the flux $\kappa T$ to be proportional to the difference of the current temperature to the surrounding temperature $T_{ref}$:
\[
\kappa T_{i,n} = \eta(T_{ref} - T)
\]  
(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:
\[
T(x, 0) = T_{ref}
\]  
(1.18)
for all $x$ in the domain. Note 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 $t^{(n)}$:
\[
T^{(n)} \approx T^{(n-1)} + T^{(n)}_{,i}(t^{(n)} - t^{(n-1)}) = T^{(n-1)} + h \cdot T^{(n)}_{,i}
\]  
(1.19)
This is inserted into Equation (1.15). By separating the terms at $t^{(n)}$ and $t^{(n-1)}$ one gets for $n = 1, 2, 3 \ldots$
\[
\frac{\rho c_p}{h} T^{(n)} - (\kappa T_{,i}^{(n)})_{,i} = q_H + \frac{\rho c_p}{h} T^{(n-1)}
\]  
(1.20)
where $T^{(0)} = T_{ref}$ is taken form the initial condition given by Equation (1.18). Together with the natural boundary condition
\[
\kappa T_{,i}^{(n)} n_i = \eta(T_{ref} - T^{(n)})
\]  
(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 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
\[
\omega T^{(n)} - (\kappa T_{,i}^{(n)})_{,i} = f
\]  
(1.22)
and we set
\[
\omega = \frac{\rho c_p}{h} \text{ and } f = q_H + \frac{\rho c_p}{h} T^{(n-1)}.
\]  
(1.23)
With $g = \eta T_{ref}$ the radiation condition defined by Equation (1.21) takes the form
\[
\kappa T_{,i}^{(n)} n_i = g - \eta T^{(n)} \text{ on } \Gamma
\]  
(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:
\[
-\langle A_{ij} u_{,j} \rangle_{,j} + Du = Y
\]  
(1.25)
where we show only the coefficients relevant for the problem discussed here. For the general form of a single PDE see Equation (4.1). The coefficients $A$ and $Y$ have to be specified through Data objects in the general FunctionSpace on the PDE or objects that can be converted into such Data objects. $A$ is a rank-2 Data object and $D$ and $Y$ are scalar. The following natural boundary conditions are considered on $\Gamma$:

$$n_j A_{jl} u, l + d u = y.$$  \tag{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 prescribe the value of the solution at certain locations in the domain. They have the form

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

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

$$A_{ij} = \kappa \delta_{ij} \quad D = \omega \quad Y = f$$

$$d = \eta \quad y = g$$ \tag{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.\footnote{There is a difference in esys.escript for a coefficient to be not present and set to zero. Since in the former case the coefficient is not processed, it is more efficient to leave it undefined instead of assigning zero to it.}

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 the following way:\footnote{Note that this is not a complete code. The full source code can be found in “helmholtz.py”}:

```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 an esys.escript function returning the Kronecker symbol.

The coefficients can set by several calls to 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()
```

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

$$A_{jl} = A_{lj}.$$ \tag{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:

```python
mypde = LinearPDE(mydomain)
mypde.setValue(A=kappa*kronecker(mydomain), d=eta)
print mypde.checkSymmetry() # returns True
```

1.3. The Diffusion Problem
print mypde.checkSymmetry() # returns False
mypde.setValue(C=kronecker(mydomain)[0])
print mypde.checkSymmetry() # returns True

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

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

Now we want to see how we actually solve the Helmholtz equation on a rectangular domain of length $l_0 = 5$ and height $l_1 = 1$. We choose a simple test solution such that we can verify the returned solution against the exact answer. Actually, we take $T = x_0$ (here $qH = 0$) 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 $\kappa n_i u_i = \kappa n_0$. Thus we have to set $g = \kappa n_0 + \eta x_0$. The following script `helmholtz.py` which is available in the example directory implements this test problem using the `esys.finley` PDE solver:

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

To visualize the solution `x0.vtu` you can use the command

```
mayavi -d x0.vtu -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` is 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` to approximate the solution, and our solution is a linear function of the spatial coordinates, one might expect that the error would be zero or in the order of machine precision (typically $\approx 10^{-15}$). However most PDE packages use an iterative solver which is terminated when a given tolerance has been reached. The default tolerance is $10^{-8}$. This value can be altered by using the `setTolerance` of the `LinearPDE` class.

### 1.3.4 The Transition Problem

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

```python
t=0
T=Tref
mypde=LinearPDE(mydomain)
```
mypde.setValue(A=kappa*kronecker(mydomain), D=rhocp/h, d=eta, y=eta*Tref)
while t<t_end:
    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 (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 is equal to T_{ref}. The LinearPDE object mypde and the coefficients that do not change over time are set up before the loop 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 \( q_c \) in an area defined as a circle of radius \( r \) and center \( x_c \), and zero outside this circle. \( q_0 \) 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 object of the esys.escript module defining locations in the Domain mydomain. The length() function imported from the esys.escript module returns the Euclidean norm:

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

(1.30)

So length(x-xc) calculates the distances of the location \( x \) to the center of the circle \( x_c \) where the heat source is acting. Note that the coordinates of \( x_c \) are defined as a list of floating point numbers. It is automatically converted into a Data class object before being subtracted from \( x \). The function whereNegative applied to length(x-xc)-r returns a Data object which is equal to one where the object is negative (inside the circle) and zero elsewhere. After multiplication with \( q_c \) we get a function with the desired property of having value \( q_c \) 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
from esys.weipa import saveVTK
#... set some parameters ...
xc=[0.02, 0.002]
r=0.001
qc=50.e6
Tref=0.
rhocp=2.6e6
eta=75.
kappa=240.
tend=5.
# ... time, time step size and counter ...
t=0
h=0.1
i=0
#... generate domain ...
mydomain = Rectangle(l0=0.05, l1=0.01, n0=250, n1=50)
#... open PDE ...
mypde=LinearPDE(mydomain)
mypde.setSymmetryOn()
mypde.setValue(A=kappa*kronecker(mydomain), D=rhocp/h, d=eta, y=eta*Tref)
#... set heat source: ....
x=mydomain.getX()
```

1.3. The Diffusion Problem
The script will create the files T.1.vtu, T.2.vtu, ..., T.50.vtu in the directory where the script has been started. The files contain the temperature distributions at time steps 1, 2, i, ..., 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.vtu -m SurfaceMap
```

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

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

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

$$\sigma_{ij} = \lambda u_{k,k} \delta_{ij} + \mu (u_{i,j} + u_{j,i})$$  \hspace{1cm} (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

$$\sigma_{ij} n_j = 0$$  \hspace{1cm} (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

$$u_0(x_c,t) = U_0 \sqrt{\frac{2}{\alpha}} \frac{(t-t_0)}{\alpha} e^{-\frac{(t-t_0)^2}{\alpha^2}}$$  \hspace{1cm} (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 apply the source as a constraint in a sphere of small radius around the point \( x_C \).

We use an explicit time integration scheme to calculate the displacement field \( u \) at certain time marks \( t^{(n)} \), where \( t^{(n)} = t^{(n-1)} + h \) with time step size \( h > 0 \). In the following the upper index \( (n) \) refers to values at time \( t^{(n)} \). We use the Verlet scheme with constant time step size \( h \) which is defined by

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

(1.35)

(1.36)

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

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

(1.37)

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

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

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

(1.38)

and boundary conditions

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

(1.39)

derived from Equation (1.33) where

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

(1.40)

We also need to apply the constraint

\[
 a^{(n)}_0(x_C, t) = U_0 \sqrt{\frac{2}{\alpha}} \left( \frac{4(t - t_0)^3}{\alpha^3} - 6 \frac{t - t_0}{\alpha} \right) e^2 - \frac{(t-t_0)^2}{\alpha^2}
\]

(1.41)
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, \( u^{(n)} \), into a problem for acceleration, \( a^{(n)} \), which depends on the solution
at the previous two time steps \( u^{(n-1)} \) and \( u^{(n-2)} \).

In each time step we have to solve this problem to get the acceleration \( a^{(n)} \), and we will use the LinearPDE
class of the esys.escript.linearPDEs package 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

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

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 \( X \):

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

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 3% of the width of the domain):

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

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

```python
def wavePropagation(domain, h, t, 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 = [], [], [], []
```

Figure 1.8: Input Acceleration at Source Point (\( \alpha = 0.7, t_0 = 3, U_0 = 1 \)).
Notice that all coefficients of the PDE which are independent of time $t$ are set outside the while loop. This is for efficiency reasons since it allows the LinearPDE class to reuse information as much as possible when iterating over time.

The statement
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, see also Section 4.4.

There are a few new \texttt{esys.escript} functions in this example: \texttt{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 \texttt{grad} function, for instance the statement \( \text{grad} (\text{grad}(u)) \) will raise an exception. \texttt{trace(g)} returns the sum of the main diagonal elements \( g[k,k] \) of \( g \) and \texttt{transpose(g)} returns the matrix transpose of \( g \) (i.e. \texttt{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 \texttt{FunctionSpace} as they have to be seen as solutions of PDEs from previous time steps. In fact, the \texttt{grad} does not accept arguments with a certain \texttt{FunctionSpace}, for more details see Section 3.2.3.

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

\begin{verbatim}
L=Locator(domain, xc)
u=...
u_pc=L.getValue(u)
\end{verbatim}

The return value \( u_{\text{pc}} \) is the value of \( u \) at the location \( x_C \). The values are collected in the lists \( u_{\text{pc}0}, u_{\text{pc}1} \) and \( u_{\text{pc}2} \) together with the corresponding time marker in \( t_s \). These values are handed back to the caller. Later we will show 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 than the stress which is actually present at the current time step. Schemes using this approach are called 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 than values from previous time steps. This will lead to a problem which is more expensive to solve, in particular for non-linear cases. Although explicit time integration schemes are cheap to finalize a single time step, they need significantly smaller time steps than implicit schemes and can suffer from stability problems. Therefore they require a very careful selection of the time step size \( h \).

An easy, heuristic way of choosing an appropriate time step size is the CourantFriedrichsLewy condition (CFL condition) which says that within a time step 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 \( \sqrt{\lambda + 2\mu \over \rho} \) so one should choose \( h \) from

\begin{equation}
  h = \frac{1}{5} \sqrt{\frac{\rho}{\lambda + 2\mu}} \Delta x
\end{equation}

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

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

\begin{verbatim}
from esys.finley import Brick
ne = 32  # number of cells in x_0 and x_1 directions
width = 10000.  # length in x_0 and x_1 directions
lam = 3.462e9
mu = 3.462e9
rho = 1154.
tend = 60
U0 = 1.  # amplitude of point source

mypde.getSolverOptions().setSolverMethod(mypde.getSolverOptions().HRZ_LUMPING)
\end{verbatim}

In fact, it is the finite element node which is closest to the given position. The usage of \texttt{Locator} is MPI safe.
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.

# spherical source at middle of bottom face
xc=[width/2.,width/2.,0.]
# define small radius around point xc
src_radius = 0.03*width
print("src_radius =",src_radius)
mydomain=Brick(ne, ne, 10, 10-width, 11-width, 12=10.*width/32.)
h=(1./5.)*inf(sqrt(rho/(lam+2*mu)))*inf(domain.getSize())
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() function returns the local element size $\Delta x$. Using $\inf$ ensures that the CFL condition \ref{1.46} holds 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 -n SurfaceMap

You can rotate this figure by clicking on it with the mouse and moving it around. Again use Configure Data to move backward and forward in time, and also to choose the results (acceleration, displacement or $u_x$) 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\cite{37}, Excel or OpenOffice.org Calc to read the data for further analysis. The following code shows one possible way to write the data to the file ./data/U_pc.csv:

```python
u_pc_data=FileWriter('./data/U_pc.csv')
for i in xrange(len(ts)):
    u_pc_data.write("%f %f %f\n"%(ts[i],u_pc0[i],u_pc1[i],u_pc2[i]))
u_pc_data.close()
```

1.3. The Diffusion Problem
The file U_pc.csv 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[37] the command:

```
plot 'U_pc.csv' u 1:2 title 'U_x' w l lw 2, 'U_pc.csv' u 1:3 title 'U_y' w l lw 2, 'U_pc.csv' 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). It is pointed out that we are not using the standard python open to write to the file U_pc.csv 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 plt.show() to create an interactive browser window. Notice that by checking the condition getMPIRankWorld()==0 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 included in the example file wave.py. There are other options available to process these data in particular through the SciPy[15] package, e.g. Fourier transformations. It
is beyond the scope of this user’s guide to document the usage of SciPy [15] for time series analysis but it is highly recommended that users use SciPy [15] for any further data processing.

1.4 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 ?? 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 through 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. 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 a rank-4 Data object and \( X \) is a rank-2 Data object. We show here the coefficients relevant for the problem we are 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)

while constraints take the form

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

(1.55)

\( r \) and \( q \) are each a 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)

(1.58)

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

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

(1.59)
Under the assumption that $\lambda$, $\mu$, $\beta$ and $T_{\text{ref}}$ 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

$$A_{ijkl} = A_{klji}$$  \hspace{1cm} (1.60)

$$A_{ijkl} = A_{jikl}$$  \hspace{1cm} (1.61)

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

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

$$\sigma_{\text{mises}} = \sqrt{\frac{1}{2}((\sigma_{00} - \sigma_{11})^2 + (\sigma_{11} - \sigma_{22})^2 + (\sigma_{22} - \sigma_{00})^2) + 3(\sigma_{01}^2 + \sigma_{12}^2 + \sigma_{20}^2)}$$  \hspace{1cm} (1.62)

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

The following script, which is available in `heatedblock.py` in the example directory, solves the Lame equation and writes the displacements and the von-Mises stress into a file `deform.vtu` in the VTK file format:

```python
from esys.escript import *
from esys.escript.linearPDEs import LinearPDE
from esys.finley import Brick
from esys.weipa import saveVTK

#... 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.,0.]
    +whereZero(x[1])*[0.,1.,0.,0.]
    +whereZero(x[2])*[0.,0.,1.,0.]
sigma0=(lam+2./3.*mu)*alpha*(T-T_ref)*kronecker(mydomain)
mypde.setValue(A=C, X=sigma0, q=msk)

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

#... calculate von-Mises stress
sigma=mu*(g+transpose(g))+lam*trace(g)*kronecker(mydomain)-sigma0
sigma_mises=sqrt(((sigma[0,0]-sigma[1,1])**2+(sigma[1,1]-sigma[2,2])**2+(sigma[2,2]-sigma[0,0])**2)/2. + 3*(sigma[0,1]**2 + sigma[1,2]**2 + sigma[2,0]**2))

saveVTK('deform.vtu', u, sigma, sigma_mises)
```

Chapter 1. Tutorial: Solving PDEs
Finally, the results can be visualized by calling

```
mayavi2 -d deform.vtu -f CellToPointData -m VelocityVector -m SurfaceMap
```

Note that the filter CellToPointData is applied to create a smoother representation of the von-Mises stress. Figure 1.11 shows the results where the colour of the vertical planes represent the von-Mises stress and a horizontal plane of arrows shows the displacements vectors.

### 1.5 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 is low; that is, the inertial forces are small compared with the viscous forces, therefore the inertial terms in the Navier-Stokes equations can be ignored. For a body force \( f \), the governing equations are given by:

\[
\nabla \cdot (\eta (\nabla \vec{v} + (\nabla \vec{v})^T)) - \nabla p = -f, \tag{1.63}
\]

with the incompressibility condition

\[
\nabla \cdot \vec{v} = 0. \tag{1.64}
\]

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):

\[
-(\eta(v_{i,j} + v_{j,i}))_{,j} - p_{,i} = f_{i}, \tag{1.65}
\]

with the incompressibility condition

\[
-v_{i,i} = 0. \tag{1.66}
\]

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 \( f = -\rho g \delta_{33} \). The Stokes equation 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. A more detailed discussion of the class can be found in Chapter 6. In order to keep numerical stability and satisfy the Courant-Friedrichs-Lewy condition (CFL condition), the time-step size needs to be kept below a certain value. The Courant number is defined as:

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

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 x 20 elements with piecewise linear elements for the pressure and for velocity but the elements are subdivided for the velocity. This approach is called macro elements and needs to be applied to make sure that the discretized problem has a unique solution, see [21] for details. The fact that pressure and velocity are represented in different ways is expressed by

```python
velocity=Vector(0., Solution(mesh))
presure=Scalar(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 similarly 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 class 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-Friedrichs-Lewy condition (CFL 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 a file for visualization.

```python
from esys.escript import *
import esys.finley
from esys.escript.linearPDEs import LinearPDE
from esys.escript.models import StokesProblemCartesian
from esys.weipa import import saveVTK

# physical constants
rho=100.
g=10.

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

# define mesh
H=2.
L=1.
W=1.

mesh = esys.finley.Rectangle(l0=L, l1=H, order=-1, n0=20, n1=20)
coordinates = mesh.getX()
```

Alternatively, one can use second order elements for the velocity and first order elements for pressure on the same element. You can set order=2 in esys.finley Rectangle.
# gravitational force
Y=Vector(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., Solution(mesh))
pressure=Scalar(0., ReducedSolution(mesh))

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

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

    # CFL 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)

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

mayavi2 -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. Figure 1.12 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.

1.6 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 as illustrated in Figure 1.13. 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. Further, we assume an isotropic, linear elastic material model of the form

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

(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

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

(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 1.4. 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 $s_i$

$$u_i = v_i + \frac{1}{2} s_i \theta$$

(1.72)
where \( s^\pm = s \) when right of the fault and \( s^\pm = -s \) when left of the fault. We assume that \( s^\pm = 0 \) when sufficiently away from the fault.

We insert this into the stress definition in Equation (1.68)

\[
\sigma_{ij} = \sigma^c_{ij} + \frac{1}{2}\sigma^s_{ij}
\]  
(1.73)

with

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

and

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

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

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

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

\[
-\sigma^c_{ij,j} = \frac{1}{2}\sigma^s_{ij,j}
\]  
(1.77)

Together with the definition (1.74) we have a differential equation for the continuous function \( v_i \). Notice that the boundary condition (1.70) and (1.69) transfer to \( v_i \) and \( \sigma^c_{ij} \) as \( s \) is zero away from the fault. In Section 1.4 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 1 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)
```

The fault added starts at point \((0.5, 0.25)\) has length 0.5 and points north. The main purpose of the 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.
p0, p1 = fs.getW0Range(tag=1)

Typically p0 is equal to zero while p1 is equal to the length of the fault. The parameterization is given as a mapping from a set of local coordinates onto a parameter range (in our case the range p0 to p1). For instance, to map the entire domain mydomain onto the fault one can use

```python
x = mydomain.getX()
p, m = fs.getParametrization(x, tag=1)
```

Of course there is the problem that not all locations 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 define 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 of p is undefined. Now p can be used to define a slip distribution on the fault via

```python
s = m*(p-p0)*(p1-p)/((p1-p0)/2)**2*slip_max*[0.,1.]
```

Notice the factor m which ensures 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
#... set some parameters ...  
lam=1.  
mu=1  
slip_max=1.  
mydomain = Rectangle(l0=1.,l1=1.,n0=16, n1=16)  # n1 needs to be a multiple of 4! 
# .. create the fault system 
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 ...  
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  
# ... fix displacement in normal direction  
x=mydomain.getX()  
msk=whereZero(x[0])*[1.,0.] + whereZero(x[0]-1.)*[1.,0.] 
    + whereZero(x[1])*[0.,1.] + whereZero(x[1]-1.)*[0.,1.]  
mypde.setValue(A=C, X=-0.5*sigma_s, q=msk)  
#... solve pde ...  
mypde.getSolverOptions().setVerbosityOn()  
v=mypde.getSolution()  
#.. write the displacement to file:  
D=symmetric(grad(v))
```

Chapter 1. Tutorial: Solving PDEs 35
\[
\sigma = (\mu \mathbf{D} + \lambda \text{trace} \mathbf{D}) \text{kronecker} \text{(mydomain)}) + 0.5 \cdot \sigma_s
\]

The script creates the file \texttt{slip.vtu} which contains the total displacements and stress. These values are stored as cell-centered data. See Figure 1.14 for a visualization of 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\(^1\). 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\(^2\) of esys.escript allow either run-escript or escript to be used but the latter name will be removed in future releases. To run your script, issue\(^3\)

```
run-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 (Control-z on MS Windows).

To run the script using four threads (e.g. if you have a multi-core processor) you can use

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

This requires escript to be compiled with OpenMP\(^4\) support. To run the script using MPI\(^5\) with 8 processes use

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

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

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

In the case of a supercomputer or a cluster, you may wish to distribute the workload over a number of nodes\(^4\). 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:

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

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

---

\(^1\)The run-escript launcher is not supported under MS Windows yet.
\(^2\)i.e. not MS Windows or Ubuntu 9.10
\(^3\)For this discussion, it is assumed that run-escript is included in your PATH environment. See the installation guide for details.
\(^4\)For simplicity, we will use the term node to refer to either a node in a supercomputer or an individual machine in a cluster.
2.2 Options

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

```
run-escript [-n nn][-p np][-t nt][-f hostfile][-x][-V][-e][-h][-v][-o][-c][-i][-b][file][ARGS]
```

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

The options have the following meaning:

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

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

- `-t nt` the number of threads used per 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` processes should be spawned. These addresses or names of the compute nodes are 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 no host file is used.

- `-c` prints 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 · np > 1`) is used.

- `-b` do not invoke python. This is used to run non-python programs within an environment set for `escript`.

- `-e` shows additional environment variables and commands used to set up the `escript` environment. This option is useful if users wish to execute scripts without using the `run-escript` command.

- `-o` enables the redirection of messages printed by 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` if applicable.

- For MPICH and INTELMPI and for the case a hostfile is present `run-escript` will start the `mpd` daemon 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 \((rank = 0)\) will also not be available. Error output from any processor will be redirected to the node where run-escript has been invoked. If the -o Option or ESCRIPT_STDFILES is set, 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 is to avoid problems if processes are using a shared file system. Files which collect data that are global for all MPI processors are created by the process with MPI rank 0 only. Users should keep in mind that if the file system is not shared among the processes, then a file containing global information which is read by all processors needs to be copied to the local file system(s) 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 (Data excluded) should have the same values on all processors, e.g. 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 rank 0 only:

```python
from esys.escript import FileWriter
f = FileWriter('test.txt')
f.write('test message')
f.close()
```

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

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

\(^7\)getTupleForDataPoint
We strongly recommend using this class rather than Python's built-in `open` function as it will guarantee a script which will run in single processor mode as well as under MPI.

If the situation occurs that one of the processors throws an exception, for instance when 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 still 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 [19]. Lazy evaluation is when expressions are not evaluated until they are 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.

```
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 time steps, 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 give a greater benefit. If you are uncertain, try running a test on a smaller domain first.
The esys.escript Module

3.1 Concepts

esys.escript is a python module that allows you to represent the values of a function at points in a Domain in such a way that the function will be useful for the Finite Element Method (FEM) simulation. It also provides what we call a function space that describes how the data is used in the simulation. Stored along with the data is information about the elements and nodes which will be used by esys.finley.

3.1.1 Function spaces

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

A function space is described by a FunctionSpace object. The following statement generates the object solution_space which is a FunctionSpace object and provides access to the function space of PDE solutions on the Domain mydomain:

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

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

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

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

- **ReducedFunction (mydomain)**
- **ReducedFunctionOnBoundary (mydomain)**
The reduced smoothness for a PDE solution is often used to fulfill the Ladyzhenskaya-Babuska-Brezzi condition [21] 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 a 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).

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, for example 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 nodes 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.

**Figure 3.1**: Dependency of function spaces in esys.finley. An arrow indicates that a function in the FunctionSpace at the starting point can be interpolated to the FunctionSpace of the arrow target. All function spaces above the dotted line can be interpolated to any of the function spaces below the line. See also Section 4.2.

- ReducedFunctionOnContact0(mydomain)
- ReducedFunctionOnContact1(mydomain)

In comparison to the corresponding full version they use a reduced number of integration nodes (typically one only) to represent values.
3.1.2 Data Objects

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

The values of the function have a rank which gives the number of indices, and a shape defining the range of each index. The rank in *esys.escript* is limited to the range 0 through 4 and it is assumed that the rank and shape is the same for all data sample points. The shape of a *Data* object is a tuple (list) $s$ of integers. The length of $s$ is the rank of the *Data* object and the $i$-th index ranges between 0 and $s[i] - 1$. For instance, a stress field has rank 2 and shape $(d, d)$ where $d$ is the spatial dimension. The following statement creates the *Data* object `mydat` representing a continuous function with values of shape $(2, 3)$ and rank 2:

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

The initial value is the constant 1 for all data sample points and all components.

*Data* objects can also be created from any *numpy* array or any object, such as a list of floating point numbers, that can be converted into a *numpy.ndarray* [14]. The following two statements create objects which are equivalent to `mydat`:

```
mydat1=Data(value=numpy.ones((2,3)), what=ContinuousFunction(myDomain))
mydat2=Data(value=[[1,1], [1,1], [1,1]], what=ContinuousFunction(myDomain))
```

In the first case the initial value is `numpy.ones((2,3))` which generates a $2 \times 3$ matrix as an instance of *numpy.ndarray* filled with ones. The shape of the created *Data* object is taken from the shape of the array. In the second case, the creator converts the initial value, which is a list of lists, into a *numpy.ndarray* before creating the actual *Data* object.

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

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

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

*Data* objects can be manipulated by applying unary operations (e.g. cos, sin, log), and they can be combined point-wise by applying arithmetic operations (e.g. +, -, , , , , , , , , , )

We emphasize that *esys.escript* itself does not handle any spatial dependencies as it does not know how values are interpreted by the processing PDE solver library. However *esys.escript* invokes interpolation if this is needed during data manipulations. Typically, this occurs in binary operations when both arguments belong to different function spaces or when data are handed over to a PDE solver library which requires functions to be represented in a particular way.

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

\[
\sigma_{ij} = \lambda u_{kk} \delta_{ij} + \mu (u_{ij} + u_{ji})
\]

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

```
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
```

Chapter 3. The *esys.escript* Module 43
The variable $d$ gives the spatial dimension of the domain on which the displacements are defined. \texttt{kronecker} returns the Kronecker symbol with indexes $i$ and $j$ running from 0 to $d$-1. The call \texttt{grad(u)} requires the displacement field $u$ to be in the Solution or continuous FunctionSpace. The result $g$ as well as the returned stress will be in the general FunctionSpace. If, for example, $u$ is the solution of a PDE then \texttt{getStress} might be called in the following way:

\begin{verbatim}
s=getStress(u, 1., 2.)
\end{verbatim}

However \texttt{getStress} can also be called with Data objects as values for \texttt{lam} and \texttt{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:

\begin{verbatim}
lam=Scalar(1., ContinuousFunction(mydomain))
mu=Scalar(2., Function(mydomain))
s=getStress(u, lam, mu)
\end{verbatim}

The function \texttt{lam} belongs to the continuous FunctionSpace but with $g$ the function \texttt{trace(g)} is in the general FunctionSpace. 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. \texttt{esys.escript} converts the arguments into an appropriate function space according to Figure 3.1. In this example that means \texttt{esys.escript} sees lam as a function of the general FunctionSpace. 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.

### 3.1.3 Tagged, Expanded and Constant Data

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 \textit{tagging}, which uses storage efficiently. Figure 3.2 shows an example. In this case two rock types \texttt{white} and \texttt{gray} can be found in the domain. The domain is subdivided into triangular shaped cells. Each cell has a tag indicating the rock type predominantly found in this cell. Here 1 is used to indicate rock type \texttt{white} and 2 for rock type \texttt{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 to use tagged values for \texttt{lam} as shown in Figure 3.2 for the stress calculation discussed above:

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

In this example lam is set to 30 for those cells with tag white (=1) and to 5000 for 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 for models 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 while 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 specifying expanded=True in the Data object constructor, while tagged data requires calling 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.

3.1.4 Saving and Restoring Simulation Data

Data objects can be written to disk files with the dump method and read back using the load method, both of which use the netCDF [27] file format. Use these to save data for 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 a continuous FunctionSpace to the file x.nc use

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

To recover the object x, and you know that mydomain was an esys.finley mesh, use

```python
from esys.finley import LoadMesh
mydomain=LoadMesh("dom.nc")
x=load("x.nc", mydomain)
```

Obviously, it is possible to execute the same steps that were originally used to generate mydomain to recreate it. However, in most cases using dump and load is faster, particularly if optimization has been applied. If esys.escript is running on more than one MPI process dump will create an individual file for each process containing the local data. In order to avoid conflicts the file names are extended by the MPI processor rank, that is instead of one file dom.nc you would get dom.nc.0000, dom.nc.0001, etc. You still call LoadMesh("dom.nc") to load the domain but you have to make sure that the appropriate file is accessible from the corresponding rank, and loading will only succeed if you run with as many processes as were used when calling dump.

The function space of the Data is stored in x.nc. 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 reordered so be aware and confirm that you get what you wanted.

A newer, more flexible way of saving and restoring esys.escript simulation data is through an instance of the DataManager class. It has the advantage of allowing to save and load not only a Domain and Data objects but also other values you compute in your simulation script. Further, DataManager objects can simultaneously create files for visualization so no extra calls to saveVTK etc. are needed.

The following example shows how the DataManager class can be used. For an explanation of all member functions and options see the class reference section 3.2.8.

---

1The python pickle module is used for other types.
In the constructor we specify that we want RESTART (i.e. dump) files and VTK files to be saved. By default, the constructor will look for previously saved RESTART files under the current directory and load them. We can then enquire if such files were found by calling the hasData method. If it returns True we retrieve the domain and values into local variables. Otherwise the same variables are initialized with appropriate values to start a new simulation. Note, that $t$ and $t_{max}$ are regular floating point values and not Data objects. Yet they are treated the same way by the DataManager.

After this initialization step the script enters the main simulation loop where calculations are performed. When these are finalized for a time step we call the addData method to let the manager know which variables to store on disk. This does not actually save the data yet and it is allowed to call addData more than once to add information incrementally, e.g. from separate functions that have access to the DataManager instance. Once all variables have been added the export method has to be called to flush all data to disk and clear the manager. In this example, this call dumps mydomain and val to files in a restart directory and also stores $t$ and $t_{max}$ on disk. Additionally, it generates a VTK file for visualization of the data. If the script would stop running before its completion for some reason (e.g. because its runtime limit was exceeded in a multiuser environment), you could simply run it again and it would resume at the point it stopped before.

### 3.2 esys.escript Classes

#### 3.2.1 The 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)  
  writes the Domain to the file filename using the netCDF file format.

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 new locations 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 a Data object.

getSize()

returns the local sample size, i.e. the element diameter, as a 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)

returns True if tag_name is a valid tag name.

_eq_(arg)

\((\text{python } == \text{ operator})\) returns True if the Domain arg describes the same domain, False otherwise.

_ne_(arg)

\((\text{python } != \text{ operator})\) returns True if the Domain arg does not describe the same domain, False otherwise.

_str_(())

\((\text{python str()} \text{ function})\) returns a 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.

### 3.2.2 The FunctionSpace class

class FunctionSpace()

FunctionSpace objects, which are instantiated by generator functions, are used to define properties of Data objects such as continuity. 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 object 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 samples where mask is positive for a least one data point. mask must be defined on this FunctionSpace. Use the setTagMap to assign a tag name to new_tag.

_eq_(arg)
(python == operator) returns True if the FunctionSpace arg describes the same function space, False otherwise.

_ne_(arg)
(python != operator) returns True if the FunctionSpace arg does not describe the same function space, False otherwise.

_str_
(python str() function) returns a string representation of the FunctionSpace.

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

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

FunctionOnBoundary(domain)
returns the boundary 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 the 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 the geometric region defined by domain and are solutions of partial differential equations with a reduced smoothness for the solution approximation.

3.2.3 The 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>+arg</td>
<td>identical to arg</td>
</tr>
<tr>
<td>-arg</td>
<td>negation of arg</td>
</tr>
<tr>
<td>arg0+arg1</td>
<td>adds arg0 and arg1</td>
</tr>
<tr>
<td>arg0*arg1</td>
<td>multiplies arg0 and arg1</td>
</tr>
<tr>
<td>arg0-arg1</td>
<td>subtracts arg1 from arg0</td>
</tr>
<tr>
<td>arg0/arg1</td>
<td>divides arg0 by arg1</td>
</tr>
<tr>
<td>arg0**arg1</td>
<td>raises arg0 to the power of arg1</td>
</tr>
</tbody>
</table>

At least one of the arguments arg0 or arg1 must be a Data object. Either of the arguments may be a Data object, a python number or a 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+=arg1</td>
<td>adds arg1 to arg0</td>
</tr>
<tr>
<td>arg0*=arg1</td>
<td>multiplies arg0 by arg1</td>
</tr>
<tr>
<td>arg0-=arg1</td>
<td>subtracts arg1 from arg0</td>
</tr>
<tr>
<td>arg0/=arg1</td>
<td>divides arg0 by arg1</td>
</tr>
<tr>
<td>arg0**=arg1</td>
<td>raises arg0 to the power of arg1</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 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 arg</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>N/A</td>
</tr>
<tr>
<td>1</td>
<td>arg[10:u0]</td>
<td>(u0-10,)</td>
</tr>
<tr>
<td>2</td>
<td>arg[10:u0,11:u1]</td>
<td>(u0-10,u1-11)</td>
</tr>
<tr>
<td>3</td>
<td>arg[10:u0,11:u1,12:u2]</td>
<td>(u0-10,u1-11,u2-12)</td>
</tr>
<tr>
<td>4</td>
<td>arg[10:u0,11:u1,12:u2,13:u3]</td>
<td>(u0-10,u1-11,u2-12,u3-13)</td>
</tr>
</tbody>
</table>

Let s be the shape of arg, then

\[
0 \leq 10 \leq u0 \leq s[0], \\
0 \leq 11 \leq u1 \leq s[1], \\
0 \leq 12 \leq u2 \leq s[2], \\
0 \leq 13 \leq u3 \leq s[3].
\]
Any of the lower indexes \(l_0, l_1, l_2\) and \(l_3\) may not be present in which case 0 is assumed. Any of the upper indexes \(u_0, u_1, u_2\) and \(u_3\) 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 \(\text{arg}\). The following examples show slicing in action:

```
\[
\begin{align*}
t &= \text{Data}(1., (4,4,6,6), \text{Function(mydomain)}) \\
t[1,1,1,0] &= 9. \\
s &= t[:,2,:,:2;6,5] \quad \text{# \(s\) has rank 3} \\
s[:,:,1] &= 1. \\
t[:,2,:2,5,5] &= s[2:4,1,:2]
\end{align*}
\]
```

### 3.2.4 Generation of Data objects

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

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

**class Data(value, what=FunctionSpace(), expanded=False)**

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

**class Data()**

creates an empty Data object. The empty Data object is used to indicate that an argument is not present where a Data object is required.

**Scalar(value=0., what=FunctionSpace(), expanded=False)**

returns a Data object of rank 0 (a constant) in the FunctionSpace what. Values are initialized with value, a double precision quantity. If expanded is true the Data object is represented in expanded form.

**Vector(value=0., what=FunctionSpace(), expanded=False)**

returns a Data object of shape \((d,\ldots)\) in the FunctionSpace what, where \(d\) is the spatial dimension of the Domain of what. Values are initialized with value, a double precision quantity. If expanded is true the Data object is represented in expanded form.

**Tensor(value=0., what=FunctionSpace(), expanded=False)**

returns a Data object of shape \((d,d)\) in the FunctionSpace what, where \(d\) is the spatial dimension of the Domain of what. Values are initialized with value, a double precision quantity. If expanded is true the Data object is represented in expanded form.

**Tensor3(value=0., what=FunctionSpace(), expanded=False)**

returns a Data object of shape \((d,d,d)\) in the FunctionSpace what, where \(d\) is the spatial dimension of the Domain of what. Values are initialized with value, a double precision quantity. If expanded is true the Data object is represented in expanded form.

**Tensor4(value=0., what=FunctionSpace(), expanded=False)**

returns a Data object of shape \((d,d,d,d)\) in the FunctionSpace what, where \(d\) is the spatial dimension of the Domain of what. Values are initialized with value, a double precision quantity. If expanded is true the Data object is represented in expanded form.
recovered a Data object on Domain domain from the file filename, which was created by dump.

### 3.2.5 Data methods

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

- `getFunctionSpace()`: returns the FunctionSpace of the object.
- `getDomain()`: returns the Domain of the object.
- `getShape()`: returns the shape of the object as a tuple of integers.
- `getRank()`: returns the rank of the data on each data point.
- `isEmpty()`: returns True if the Data object is the empty Data object, False otherwise. 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 as 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 value is assigned the 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 the responsibility of the user to save the Domain in order to be able to recover the Data object.
- `__str__()`: returns a string representation of the object.

### 3.2.6 Functions of Data objects

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

- `saveVTK(filename, **kwdata)`: writes Data defined by keywords to the file filename using the VTK file format. The keyword is used as an identifier. The statement

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

writes the scalar T as temperature and the vector v as velocity into the file out.vtu. Restrictions on the allowed combinations of FunctionSpace apply. This method is deprecated and will be removed in a future version of escript. Use the weipa module instead!

- `saveDX(filename, **kwdata)`: writes Data defined by keywords to the file filename using the OpenDX[28] file format. The keyword is used as an identifier. The statement

```python
saveDX("out.dx", u=U, v=V)
```
**saveDX**("out.dx", temperature=T, velocity=v)

writes 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 in FunctionSpace d such that

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

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 in FunctionSpace d such that

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

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

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

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

**Lsup(a)**

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

**sup(a)**

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

**inf(a)**

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

**minval(a)**

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

**maxval(a)**

returns at each data sample point the maximum value over all components.

**length(a)**

returns the Euclidean norm at each data sample point. For a rank-4 Data object a this is

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

**trace(a[:axis_offset=0 :])**

---

52 3.2. esys.escript Classes
returns the trace of \( a \). This is the sum over components \( \text{axis} \_\text{offset} \) and \( \text{axis} \_\text{offset}+1 \) with the same index. For instance, in the case of a rank-2 \( \text{Data} \) object this is

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

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

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

\text{transpose}(a[,\text{axis} \_\text{offset}=None])

returns the transpose of \( a \). This swaps the first \( \text{axis} \_\text{offset} \) components of \( a \) with the rest. If \( \text{axis} \_\text{offset} \) is not present \( \text{int}(r/2) \) is used where \( r \) is the rank of \( a \). For instance, in the case of a rank-2 \( \text{Data} \) object this is

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

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

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

\text{swap_axes}(a[,\text{axis}0=0[,\text{axis}1=1]])

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

\[
\text{swap_axes}(a,1,2)[i,j,k,l] = a[i,k,j,l]
\]

\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 \( (a-\text{transpose}(a))/2 \).

\text{inverse}(a)

return the inverse of \( a \) so that

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

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

\text{eigenvalues}(a)

returns the eigenvalues of \( a \) so that

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

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

\text{eigenvalues_and_eigenvectors}(a)

returns the eigenvalues and eigenvectors of \( a \).

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

where \( e,V=\text{eigenvalues_and_eigenvectors}(a) \). The eigenvectors \( V \) are orthogonal and normalized, i.e.

\[
\text{matrix} \_\text{mult}(\text{transpose}(V),V) = \text{kronecker}(d)
\]
if \( a \) has shape \((d,d)\). The eigenvalues are ordered in increasing size. The argument \( a \) has to be the symmetric, i.e. \( a = \text{symmetric}(a) \). 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.

\[
\text{maximum}(a_0, a_1)[i,j] = \max(a_0[i,j], a_1[i,j])
\]

at all data sample points.

**minimum\(*a\)**

returns the minimum value over all arguments at all data sample points and for each component.

\[
\text{minimum}(a_0, a_1)[i,j] = \min(a_0[i,j], a_1[i,j])
\]

at all data sample points.

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

**inner(a0, a1)**

returns the inner product of \( a_0 \) and \( a_1 \). For instance in the case of a rank-2 Data object:

\[
\text{inner}(a) = \sum_{ij} a_0[i,j] \cdot a_1[i,j]
\]

and for a rank-4 Data object:

\[
\text{inner}(a) = \sum_{ijkl} a_0[i,j,k,l] \cdot a_1[i,j,k,l]
\]

**matrix_mult(a0, a1)**

returns the matrix product of \( a_0 \) and \( a_1 \). If \( a_1 \) is a rank-1 Data object this is

\[
\text{matrix_mult}(a) [i] = \sum_{k} a_0[i,k] \cdot a_1[k]
\]

and if \( a_1 \) is a rank-2 Data object this is

\[
\text{matrix_mult}(a) [i,j] = \sum_{k} a_0[i,k] \cdot a_1[k,j]
\]

**transposed_matrix_mult(a0,a1)**

returns the matrix product of the transposed of \( a_0 \) and \( a_1 \). The function is equivalent to \( \text{matrix_mult}(\text{transpose}(a0), a1) \). If \( a_1 \) is a rank-1 Data object this is

\[
\text{transposed_matrix_mult}(a) [i] = \sum_{k} a_0[k,i] \cdot a_1[k]
\]

and if \( a_1 \) is a rank-2 Data object this is

\[
\text{transposed_matrix_mult}(a) [i,j] = \sum_{k} a_0[k,i] \cdot a_1[k,j]
\]
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 a rank-2 Data object this is
\[
\text{matrix_transposed_mult}(a) [i,j] = \sum_k a0 \cdot [i,k] a1 [j,k] \tag{3.23}
\]

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

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

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 a rank-2 Data object this is
\[
\text{transposed_tensor_mult}(a) [i,j] = \sum_{kl} a0 [k,l,i,j] \cdot a1 [k,l] \tag{3.28}
\]
and if a1 is a rank-4 Data object this is
\[
\text{transposed_tensor_mult}(a) [i,j,k,l] = \sum_{mn} a0 [m,n,i,j] \cdot a1 [k,l,m,n] \tag{3.29}
\]
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 a rank-2 Data object this is
\[
\text{tensor_transposed_mult}(a) [i,j] = \sum_{kl} a0 [i,j,k,l] \cdot a1 [l,k] \tag{3.30}
\]
and if a1 is a rank-4 Data object this is
\[
\text{tensor_transposed_mult}(a) [i,j,k,l] = \sum_{mn} a0 [i,j,m,n] \cdot a1 [k,l,m,n] \tag{3.31}
\]

grad(a[ , where=None ])
returns the gradient of a. If where is present the gradient will be calculated in the FunctionSpace where, otherwise a default FunctionSpace is used. In case that a is a rank-2 Data object one has
\[
\text{grad}(a) [i,j,k] = \frac{\partial a [i,j]}{\partial x_k} \tag{3.32}
\]
\texttt{integrate(a[,where=None])}
returns the integral of \(a\) where the domain of integration is defined by the FunctionSpace of \(a\). If \texttt{where} is present the argument is interpolated into FunctionSpace \texttt{where} before integration. For instance in the case of a rank-2 Data object in continuous FunctionSpace it is

\[
\text{integrate}(a)[i,j] = \int_{\Omega} a[i,j] \, d\Omega \tag{3.33}
\]
where \(\Omega\) is the spatial domain and \(d\Omega\) volume integration. To integrate over the boundary of the domain one uses

\[
\text{integrate}(a,\texttt{where}=\text{FunctionOnBoundary(a.getDomain)})[i,j] = \int_{\partial\Omega} a[i,j] \, ds \tag{3.34}
\]
where \(\partial\Omega\) is the surface of the spatial domain and \(ds\) area or line integration.

\texttt{interpolate(a,where)}
interpolates argument \(a\) into the FunctionSpace \texttt{where}.

\texttt{div(a[,where=None])}
returns the divergence of \(a\):

\[
\text{div}(a) = \text{trace}(\text{grad}(a)),\text{where} \tag{3.35}
\]

\texttt{jump(a[,domain=None])}
returns the jump of \(a\) over the discontinuity in its domain or if \texttt{Domain domain} is present in \texttt{domain}.

\[
\text{jump}(a) = \text{interpolate}(a,\text{FunctionOnContactOne(domain)}) - \text{interpolate}(a,\text{FunctionOnContactZero(domain)}) \tag{3.36}
\]

\texttt{L2(a)}
returns the \(L^2\)-norm of \(a\) in its FunctionSpace. This is

\[
L^2(a) = \text{integrate}(\text{length}(a)^2). \tag{3.37}
\]

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

\texttt{sin(a)}
applies the sine function to \(a\).

\texttt{cos(a)}
applies the cosine function to \(a\).

\texttt{tan(a)}
applies the tangent function to \(a\).

\texttt{asin(a)}
applies the arc (inverse) sine function to \(a\).

\texttt{acos(a)}
applies the arc (inverse) cosine function to \(a\).

\texttt{atan(a)}
applies the arc (inverse) tangent function to \(a\).
\( \text{sinh}(a) \)

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

\( \text{cosh}(a) \)

applies the hyperbolic cosine function to \( a \).

\( \text{tanh}(a) \)

applies the hyperbolic tangent function to \( a \).

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

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

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

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

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

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

\( \exp(a) \)

applies the exponential function to \( a \).

\( \sqrt{a} \)

applies the square root function to \( a \).

\( \log(a) \)

takes the natural logarithm of \( a \).

\( \log10(a) \)

takes the base-10 logarithm of \( a \).

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

applies the sign function to \( a \). The result 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 \( a \) times \( \text{rtol} \) is used.

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

returns a function which is 1 where \( a \) is non-zero with tolerance \( \text{tol} \) and 0 otherwise. If \( \text{tol} \) is not present, the absolute maximum value of \( a \) times \( \text{rtol} \) is used.
3.2.7 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 data point in the domain. Instead, esys.escript can use an interpolation table to produce a Data object.

The following example is available as int_save.py in the example directory. We will produce a Data object which approximates a sine curve.

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

n=4
r=Rectangle(n,n)
x=r.getX()
toobig=100

First we produce an interpolation table:

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 value in the table.

num_slices=len(sine_table)-1
minval=0
maxval=1
step=sup(maxval-minval)/num_slices

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

Now we produce our new Data object:

```python
result=interpolateTable(sine_table, x[0], minval, step, toobig)
```

Any values which interpolate to larger than `toobig` will raise an exception. You can switch on boundary checking by adding `check_boundaries=True` to the argument list.

Now consider 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:

```python
st=numpy.array(sine_table)
table=[st, 0.5*st, 0*st]
```

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

```python
result=interpolateTable(table, x (minval,0), (0.55, step), toobig)
```

In the 2D case the start and step parameters are tuples \((x,y)\). By default, if a point is specified which is outside the boundary, then `interpolateTable` will operate as if the point was on the boundary. Passing `check_boundaries=True` will lead to the rejection of any points outside the boundaries by `interpolateTable`.

This method can also be called with three dimensional tables and Data objects. Tuples should be ordered \((x,y,z)\).

3.2.8 The DataManager Class

```python
class DataManager(formats=[RESTART], work_dir=".", restart_prefix="restart", do_restart=True)
```
all of those directories are deleted. The restart_prefix and do_restart parameters are ignored if
RESTART is not specified in formats.

Valid values for the formats parameter are:

**RESTART**

enables writing of checkpoint files to be able to continue simulations as explained in the class description.

**SILO**

exports simulation data in the SILO file format. esys.escript must have been compiled with SILO
support for this to work.

**VISIT**

enables the VisIt simulation interface which allows connecting to and interacting with the running simulation
from a compatible VisIt client. esys.escript must have been compiled with VisIt (version 2) support and
the version of the client has to match the version used at compile time. In order to connect to the simulation
the client needs to have access and load the file escriptsim.sim2 located under the work directory.

**VTK**

exports simulation data in the VTK file format.

The DataManager class has the following methods:

**addData(**data)**

adds Data objects and other data to the manager. Calling this method does not save or export the data yet
so it is allowed to incrementally add data at various points in the simulation script if required. Note, that
only a single domain is supported so all Data objects have to be defined on the same one or an exception
is raised.

**setDomain(domain)**

explicitly sets the domain for this manager. It is generally not required to call this method directly. Instead,
the addData method will set the domain used by the Data objects. An exception is raised if the domain
was set to a different domain before (explicitly or implicitly).

**hasData()**

returns True if the manager has loaded simulation data for a restart.

**getDomain()**

returns the domain as recovered from a restart.

**getValue(value_name)**

returns a Data object or other value with the name value_name that has been recovered after a restart.

**getCycle()**

returns the export cycle, i.e. the number of times export() has been called.

**setCheckpointFrequency(freq)**

sets the frequency with which checkpoint files are created. This is only useful if the DataManager object
was created with at least one other format next to RESTART. The frequency is 1 by default which means
that checkpoint files are created every time export() is called. Unlike visualization output, a simulation
checkpoint is usually not required at every time step. Thus, the frequency can be decreased by calling this
method with freq > 1 which would then create restart files every freq times export() is called.

**setTime(time)**

sets the simulation time stamp. This floating point number is stored in the metadata of exported data but
not used by RESTART.
setMeshLabels(x, y, z="")
sets labels for the mesh axes. These are currently only used by the SILO exporter.

setMeshUnits(x, y, z="")
sets units for the mesh axes. These are currently only used by the SILO exporter.

setMetadataSchemaString(schema, metadata="")
sets metadata namespaces and the corresponding metadata. These are currently only used by the VTK exporter. schema is a dictionary that maps prefixes to namespace names, e.g. 
{"gml": "http://www.opengis.net/gml"} and metadata is a string with the actual content which will be enclosed in <MetaData> tags.

export()
executes the actual data export. Depending on the formats parameter used in the constructor all data added by addData() is written to disk (RESTART, SILO, VTK) or made available through the VisIt simulation interface (VISIT). At least the domain must be set for something to be exported.

3.2.9 Saving Data as CSV
For simple post-processing, Data objects can be saved in comma separated value (CSV) format. If mydata1 and mydata2 are scalar data, the command

saveDataCSV('output.csv', U=mydata1, V=mydata2)

will record the values in output.csv in the following format:

U, V
1.0000000e+0, 2.0000000e-1
5.0000000e-0, 1.0000000e+1
...

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 is raised.

Output can be restricted using a scalar mask as follows:

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

This command 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.
3.2.10 The Operator Class

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

```python
class Operator:
    creates an empty Operator object.

isEmpty(fileName)
    returns True is the object is empty, False otherwise.

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 with name fileName, see http://maths.nist.gov/MatrixMarket
```

3.3 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

```python
from esys.escript.unitsSI import *
l=20*m
w=30*kg
w2=40*lb
T=100*Celsius
```

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

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

defines 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:

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

The pressure \( p \) is set to 40 Mega Pascal. Units can also be converted back from the SI system into a desired unit, e.g.

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

can be used print the pressure in units of atmosphere.

The following is an incomplete list of supported physical units:

```python
km
    unit of kilometer
```
m  unit of meter
cm  unit of centimeter
mm  unit of millimeter
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 kilogram
lb  unit of pound
metric ton  unit of pound
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 degrees Kelvin

Celsius  
unit of degrees Celsius

Fahrenheit  
unit of degrees Fahrenheit

Supported unit prefixes:

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 = $10^1$

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}

3.4 Utilities

The FileWriter class provides a mechanism to write data to a file. In essence, this class wraps the standard python 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 will run with and without MPI. It is safe to use open under MPI to read data which are global under MPI.

class FileWriter(fn[, _append=False, _createLocalFiles=False])

Opens a file with name fn for writing. If _append is set to True data are appended at the end of the file. If running under MPI, only the first processor (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 debugging purposes only.

The following methods are available:

close()

closes the file.

flush()

flushes the internal buffer to disk.

write(txt)

writes string txt to the file. Note that a newline is not added.

writelines(txts)

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

closed

this member is True if the file is closed.

mode

holds the access mode.

name

holds the file name.

newlines

holds the 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 current 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 the MPI_COMM_WORLD processor group.

getMPIWorldMax(a)
    returns the maximum value of the integer a across all processors within MPI_COMM_WORLD.
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 $\Omega$ and $n$ denotes the outer normal field on $\Gamma$.

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

$$
-(A_{jl}u_{,l})_{,j} - (B_{j}u)_{,j} + C_{j}u_{,j} + Du = -X_{j,j} + Y .
$$

(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 used in this chapter. 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 each a rank-1 Data object and $D$ and $Y$ are scalars. 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 prescribe 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 a 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

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

(4.4)

$A$ is a rank-4 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:

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

(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 a rank-1 Data object. Notice that not necessarily all components must have a constraint at all locations.
LinearPDE also supports solution discontinuities over a contact region $\Gamma^{\text{contact}}$ in the domain $\Omega$. To specify the conditions across the discontinuity we are using the generalised flux $J$ which in the case of a system of PDEs and several components of the solution, is defined as

$$J_{ij} = A_{ijkl} u_{k,l} + B_{ijk} u_k - X_{ij}$$  \hspace{1cm} (4.7)

For the case of single solution component and single PDE, $J$ is defined as

$$J_j = A_{jl} u_{l} + B_{j} u_k - X_{j}$$  \hspace{1cm} (4.8)

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

$$n_j J^0_{ij} = n_j J^1_{ij} = y^{\text{contact}} - d^{\text{contact}} [u]_k.$$  \hspace{1cm} (4.9)

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

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

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

The PDE is symmetrical if

$$A_{jl} = A_{lj} \quad \text{and} \quad B_j = C_j$$  \hspace{1cm} (4.11)

The system of PDEs is symmetrical if

$$A_{ijkl} = A_{klij}$$  \hspace{1cm} (4.12)

$$B_{ijk} = C_{kij}$$  \hspace{1cm} (4.13)

$$D_{ik} = D_{ki}$$  \hspace{1cm} (4.14)

$$d_{ik} = d_{ki}$$  \hspace{1cm} (4.15)

$$d^{\text{contact}}_{ik} = d^{\text{contact}}_{ki}$$  \hspace{1cm} (4.16)

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

The following example illustrates a typical usage of the LinearPDE class:

```python
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 holds options for the solver that may be set before solving the PDE. In the following example the getSolverOptions method is used to access the SolverOptions object attached to mypde:

```python
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.setSymmetryOn()
mypde.setValue(A=kappa*kronecker(mydomain), D=1, Y=1)
```

In 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 the diffusivity) which differs from the definition used here by the sign. This needs to be kept in mind when defining natural boundary conditions.

---

68 4.1. Linear Partial Differential Equations
In this example, the preconditioned conjugate gradient method SolverOptions.PCG is used with preconditioner SolverOptions.AMG. The relative tolerance is set to $10^{-8}$ and the maximum number of iteration steps to 1000. After a completed call to getSolution(), the attached SolverOptions object gives access to diagnostic information:

```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 variable indicates that it 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, e.g. `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).

```python
class LinearPDE(domain,numEquations=0,numSolutions=0)
    opens a linear, steady, second order PDE on the Domain domain. The parameters numEquations and numSolutions give the number of equations and the number of solution components. If numEquations and numSolutions are non-positive, then the number of equations and the number of 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)
    returns the value assigned to coefficient name. If name is not a valid name an exception is raised.

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

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

\(^2\)In fact, it is assumed they are not present by assigning the value `escript.Data()`. This can be used by the solver library to reduce computational costs.
setDebugOn()
switches on debug mode so more diagnostic messages will be printed.

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.

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 matrix lumping is set as the solver for the system of linear equations, False otherwise.

getDomain()
returns the Domain of the PDE.

getDim()
returns the spatial dimension of the PDE.

getNumEquations()
returns the number of equations.

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

checkSymmetry( verbose=False )
returns True if the PDE is symmetric, 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 is printed.

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

isSymmetric()
returns True if the PDE has been indicated to be symmetric, False otherwise.

setSymmetryOn()
indicates that the PDE is symmetric.

setSymmetryOff()
indicates that the PDE is not symmetric.

setReducedOrderOn()
enables 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()
disables 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 computationally expensive and – depending on the PDE and the discretization – can 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

$$ -u_{,ii} = f $$  \hspace{1cm} (4.17)

with homogeneous boundary conditions

$$ n_i u,_{i} = 0 $$  \hspace{1cm} (4.18)

and homogeneous constraints

$$ u = 0 \text{ where } q > 0. $$  \hspace{1cm} (4.19)

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

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

setValue(f=escript.Data(),q=escript.Data())
assigns new values to $f$ and $q$.

4.1.4 The Helmholtz Class
The Helmholtz class defines the Helmholtz problem

$$ \omega u - (k u,_{j} j = f $$  \hspace{1cm} (4.20)

with natural boundary conditions

$$ k u,_{j} n,_{j} = g - \alpha u $$  \hspace{1cm} (4.21)

and constraints

$$ u = r \text{ where } q > 0. $$  \hspace{1cm} (4.22)

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

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

setValue([omega], [k], [f], [alpha], [g], [r], [q])
assigns new values to $\omega$, $k$, $f$, $\alpha$, $g$, $r$, and $q$. By default all values are set to zero.
4.1.5 The Lame Class

The Lame class defines a Lame equation problem

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

(4.23)

with natural boundary conditions

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

(4.24)

and constraint

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

(4.25)

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

class Lame(domain)

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

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

assigns new values to \(lame\_lambda\), \(lame\_mu\), \(F\), \(sigma\), \(f\), \(r\), and \(q\). By default all values are set to zero.

4.2 Projection

Using the LinearPDE class provides an option to change the FunctionSpace attribute in addition to the standard interpolation mechanism as discussed in Chapter 3. If you consider the stripped-down version

\[u = Y\]  

(4.26)

of the general scalar PDE 4.1 (boundary conditions are irrelevant), you can see the solution \(u\) of this PDE as a projection 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 possibility of projecting a potentially discontinuous function onto a continuous function, although this may 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 often used 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 \((= 1\text{ for } x_0 \geq 0\text{ and } = -1\text{ for } x_0 < 0).\) 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 than using the standard solver techniques of PDEs. In essence it leads to using the average of neighbour element values to calculate the value at each FEM node.

The following script illustrates how to evaluate the normal stress on the boundary from a given displacement field \(u\):
from esys.escript.pdetools import Projector

u=...
proj=Projector(u.getDomain())
e=symmetric(grad(u))
stress = G*e+ (K-2./3.*G)*trace(e)*kronecker(u.getDomain())
normal_stress = inner(u.getDomain().getNormal(), proj(stress))

class Projector(domain, reduce=True, fast=True):
    This class defines a projector on the domain domain. If reduce is set to True the projection will be
    returned as a reduced solution FunctionSpace Data object. Otherwise the 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 requires significantly less computation time and memory. The class is callable.

    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 information.

    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.ROWSUM_LUMPING
        SolverOptions.HRZ_LUMPING
        SolverOptions.ITERATIVE
        SolverOptions.NONLINEAR_GMRES
        SolverOptions.TFQMR
Not all packages support all solvers. It can be assumed that a package makes a reasonable choice if it encounters an unknown solver. See Table 7.2 for the solvers supported by esys.finley.

getSolverMethod()
returns the key of the solver method to be used.

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

def getPreconditioner()
returns the key of the preconditioner to be used.

def setPackage([ package=SolverOptions.DEFAULT ])
sets the solver package to be used as a solver. The value of method must be one of the constants:
SolverOptions.DEFAULT
SolverOptions.PASO
SolverOptions.SUPER_LU
SolverOptions.PASTIX
SolverOptions.MKL
SolverOptions.UMFPACK.
Not all packages are supported on all implementations. 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 and SolverOptions.UMFPACK.

def getPackage()
returns the solver package key.

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

def getDiagnostics([ name ])
returns the diagnostic information name. The following keywords are supported:
"num_iter": number of iteration steps
"cum_num_iter": cumulative number of iteration steps
"num_level": number of levels in the multi level solver
"num_inner_iter": number of inner iteration steps
"cum_num_inner_iter": cumulative number of inner iteration steps
"time": execution time
"cum_time": cumulative execution time
"set_up_time": time to set up the solver, typically this includes factorization and reordering
"cum_set_up_time": cumulative time to set up the solver

\[1\] 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 be caused by a non-invertible operator.
"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": status of convergence
"preconditioner_size": size of preconditioner in MBytes.

`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.

`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
SolverOptions.DEFAULT_REORDERING.

`getReordering()`
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 to perform a restart. If `restart` is equal to `None` no restart is performed.

`getRestart()`
returns the number of iterations steps after which `SolverOptions.GMRES` performs a restart.

`setTruncation(truncation=20)`
sets the number of residuals in `SolverOptions.GMRES` to be stored for orthogonalization.

`getTruncation()`
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.

`getCoarseningThreshold()`
returns the threshold for coarsening in the SolverOptions.AMG solver or preconditioner.

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

```python
getMinCoarseMatrixSize()
```
returns the minimum size of the coarsest level matrix in SolverOptions.AMG.

```python
setSmöother([ smoother=SolverOptions.GAUSS_SEIDEL ])
```
sets the SolverOptions.JACOBI or SolverOptions.GAUSS_SEIDEL smoother to be used with SolverOptions.AMG.

```python
getSmöother()
```
returns the key of the smoother used in SolverOptions.AMG.

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

```python
getNumSweeps()
```
returns the number of sweeps in a SolverOptions.JACOBI or SolverOptions.GAUSS_SEIDEL preconditioner.

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

```python
getNumPreSweeps()
```
returns the number of sweeps in the pre-smoothing step of SolverOptions.AMG.

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

```python
getNumPostSweeps()
```
returns the number of sweeps in the post-smoothing step of SolverOptions.AMG.

```python
setTolerance([ rtol=1.e-8 ])
```
sets the relative tolerance for the solver. The actual 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.

```python
getTolerance()
```
returns the relative tolerance for the solver.

```python
setAbsoluteTolerance([ atol=0. ])
```
sets the absolute tolerance for the solver. The actual 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.

```python
getAbsoluteTolerance()
```
returns the absolute tolerance for the solver.
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.

**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 SolverOptions.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 the 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 [24].

UMFPACK
the UMFPACK library, Reference [34]. Note that UMFPACK is not parallelized.

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

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

DIRECT
the default direct linear solver.

CHOLEVSKY
direct solver based on Cholevsky factorization (or similar), see Reference [31]. The solver requires a symmetric PDE.

PCG
preconditioned conjugate gradient method, see Reference [36]. The solver requires a symmetric PDE.

TFQMR
transpose-free quasi-minimal residual method, see Reference [36].

GMRES
the GMRES method, see Reference [36]. Truncation and restart are controlled by the truncation and restart parameters of getSolution.

MINRES
minimal residual method

ROWSUM_LUMPING
row sum lumping of the stiffness matrix, see Section ?? for details. Lumping does not use the linear system solver library.

HRZ_LUMPING
HRZ lumping of the stiffness matrix, see Section ?? for details. 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 [36].

CGS
conjugate gradient squared method, see Reference [36].

---

4The MKL library will only be available when the Intel compilation environment was used to build esys.escript.
BICGSTAB
stabilized bi-conjugate gradients methods, see Reference [36].

SSOR
symmetric successive over-relaxation method, see Reference [36]. 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 [31].

JACOBI
the Jacobi preconditioner, see Reference [31].

AMG
the algebraic multi grid method, see Reference [32]. This method can be used as linear solver method but is more robust when used as a preconditioner.

GAUSS_SEIDEL
the symmetric Gauss-Seidel preconditioner, see Reference [31]. getNumSweeps() is the number of sweeps used.

RILU
relaxed incomplete LU factorization preconditioner, see Reference [?]. This method is similar to the one used for 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 [38]. This method is similar to the one used for SolverOptions.ILU0 but applies reordering during the factorization.

NO_REORDERING
no reordering 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.

SUPER_LU
the SuperLU library [17] is used as a solver.

PASTIX
the Pastix library [22] is used as a solver.

NO_PRECONDITIONER
no preconditioner is applied.

4.4 Some Remarks on Lumping

Explicit time integration schemes (two examples are discussed later in this section), require very small time steps in order to maintain numerical stability. Unfortunately, these small time increments often result in a prohibitive computational cost. In order to minimise these costs, a technique termed lumping can be utilised. Lumping is applied to the coefficient matrix, reducing it to a simple diagonal matrix. This can significantly improve the computational speed, because the solution updates are simplified to a simple component-by-component vector-vector product. However, some care is required when making radical approximations such as these. In this section, two commonly applied lumping techniques are discussed, namely row sum lumping and HRZ lumping.
4.4.1 Scalar wave equation

One example where lumping can be applied to a hyperbolic problem, is the scalar wave equation
\[ u_{tt} = c^2 u_{xx}. \]  
(4.27)

In this example, both of the lumping schemes are tested against the reference solution
\[ u = \sin(5\pi(x_0 - c \cdot t)) \]  
(4.28)
over the 2D unit square. Note that \( u_n_i = 0 \) on faces \( x_1 = 0 \) and \( x_1 = 1 \). Thus, on the faces \( x_0 = 0 \) and \( x_0 = 1 \) the solution is constrained.

To solve this problem the explicit Verlet scheme was used with a constant time step size \( dt \) given by
\[ u^{(n+1)} = 2u^{(n)} - u^{(n-1)} + dt^2 a^{(n)} \]  
(4.29)
for all \( n = 2, 3, \ldots \) where the upper index \( (n) \) refers to values at time \( t^{(n)} = t^{(n-1)} + h \) and \( a^{(n)} \) is the solution of
\[ a^{(n)} = c^2 u^{(n-1)}_{x x}. \]  
(4.30)
This equation can be interpreted as a PDE for the unknown value \( a^{(n)} \), which must be solved at each time-step. In the notation of equation 4.1 we thus set \( D = 1 \) and \( X = -c^2 u^{(n-1)}_{x x} \). Furthermore, in order to maintain stability, the time step size needs to fulfill the Courant-Friedrichs-Lewy condition (CFL condition). For this example, the CFL condition takes the form
\[ dt = f \cdot \frac{dx}{c}. \]  
(4.31)
where \( dx \) is the mesh size and \( f \) is a safety factor. In this example, we use \( f = \frac{1}{6} \).

Figure 4.1 depicts a temporal comparison between four alternative solution algorithms: the exact solution; using a full mass matrix; using HRZ lumping; and row sum lumping. The domain utilized rectangular order 1 elements (element size is 0.01) with observations taken at the point \( (\frac{1}{2}, \frac{1}{2}) \). All four solutions appear to be identical for this example. This is not the case for order 2 elements, as illustrated in Figure 4.2. For the order 2 elements, the row sum lumping has become unstable. Row sum lumping is unstable in this case because for order 2 elements, a row sum can result in a value of zero. HRZ lumping does not display the same problems, but rather exhibits behaviour similar to the full mass matrix solution. When using both the HRZ lumping and full mass matrix, the wave-front is slightly delayed when compared with the analytical solution.

For the full mass matrix, the acceleration 4.30 and displacement formulations 4.32 are identical.

Alternatively, one can directly solve for \( u^{(n)} \) by inserting equation 4.29 into equation 4.30:
\[ u^{(n)} = 2u^{(n-1)} - u^{(n-2)} + (dt \cdot c)^2 u^{(n-1)}_{x x}. \]  
(4.32)
This can also be interpreted as a PDE that must be solved at each time-step, but for the unknown \( u^{(n)} \). As per equation 4.1 we set the general form coefficients to: \( D = 1; Y = 2u^{(n-1)} - u^{(n-2)} \); and \( X = -(h \cdot c)^2 u^{(n-1)}_{x x} \). For the full mass matrix, the acceleration 4.30 and displacement formulations 4.32 are identical.
Figure 4.2: Amplitude at point \((\frac{1}{2}, \frac{1}{2})\) using the acceleration formulation 4.30 of the Velet scheme with order 2 elements, element size 0.01, and \(c = 1\).

Figure 4.3: Amplitude at point \((\frac{1}{2}, \frac{1}{2})\) using the displacement formulation 4.32 of the Velet scheme with order 1 elements, element size 0.01 and \(c = 1\).

The displacement solution is depicted in Figure 4.3. The domain utilised order 1 elements (for order 2, both lumping methods are unstable). The solutions for the exact and the full mass matrix approximation are almost identical while the lumping solutions, whilst identical to each other, exhibit a considerably faster wave-front propagation and a decaying amplitude.

4.4.2 Advection equation

Consider now, a second example that demonstrates the advection equation

\[
\frac{\partial u}{\partial t} = (v_i u)_i \tag{4.33}
\]

where \(v_i\) is a given velocity field. To simplify this example, set \(v_i = (1, 0)\) and

\[
 u(x, t) = \begin{cases} 
 1 & x_0 < t \\
 0 & x_0 \geq t
\end{cases} \tag{4.34}
\]

The solution scheme implemented, is the two-step Taylor-Galerkin scheme (which is in this case equivalent to SUPG): the incremental formulation is given as

\[
\frac{d u^{n+\frac{1}{2}}}{dt} = \frac{dt}{2} (v_i u^{n-1})_i \tag{4.35}
\]

\[
d u^{n} = dt(v_i (u^{n-1} + d u^{n-\frac{1}{2}}))_i \tag{4.36}
\]

\[
u^{n} = u^{n} + d u^{n} \tag{4.37}
\]
This can be reformulated to calculate \( u^{(n)} \) directly:

\[
\begin{align*}
    u^{(n-\frac{1}{2})} &= u^{(n-1)} + \frac{dt}{2} (v_i u^{(n-1)})_i \\
    u^{(n)} &= u^{(n-1)} + dt(v_i u^{(n-\frac{1}{2})})_i
\end{align*}
\] (4.38) (4.39)

In some cases it may be possible to combine the two equations to calculate \( u^{(n)} \) without the intermediate step. This approach is not discussed, because it is inflexible when a greater number of terms (e.g. a diffusion term) are added to the right hand side.

The advection problem is thus similar to the wave propagation problem, because the time step also needs to satisfy the CFL condition. For the advection problem, this takes the form

\[
dt = f \cdot \frac{dx}{\|v\|}.
\] (4.40)

where \( dx \) is the mesh size and \( f \) is a safety factor. For this example, we again use \( f = \frac{1}{6} \).

Figures 4.4 and 4.5 illustrate the four incremental formulation solutions: the true solution; the exact mass matrix; the HRZ lumping; and the row sum lumping. Observe, that for the order 1 elements case, there is little deviation from the exact solution before the wave front, whilst there is a significant degree of oscillation after the wave-front has passed. For the order 2 elements example, all of the numerical techniques fail.

\[\text{Figure 4.4: Amplitude at point } \left( \frac{1}{2}, \frac{1}{2} \right) \text{ using the incremental formulation 4.35 of the Taylor-Galerkin scheme with order 1 elements, element size } dx = 0.01, v = (1, 0).\]

\[\text{Figure 4.5: Amplitude at point } \left( \frac{1}{2}, \frac{1}{2} \right) \text{ using the incremental formulation 4.35 of the Taylor-Galerkin scheme with order 2 elements, element size } 0.01, v = (1, 0).\]

\[\text{Figure 4.6 depicts the results from the direct formulation of the advection problem for an order 1 mesh. Generally, the results have improved when compared with the incremental formulation. The full mass matrix still introduces some oscillation both before and after the arrival of the wave-front at the observation point. The two}\]
lumping solutions are identical, and have introduced additional smoothing to the solution. There are no oscillatory effects when using lumping for this example. In Figure 4.7 the mesh or element size has been reduced from 0.01 to 0.002 units. As predicted by the CFL condition, this significantly improves the results when lumping is applied. However, when utilising the full mass matrix, a smaller mesh size will result in post wave-front oscillations which are higher frequency and slower to decay.

Figure 4.8 illustrates the results when utilising elements of order 2. The full mass matrix and HRZ lumping formulations are unable to correctly model the exact solution. Only the row sum lumping was capable of producing a smooth and sensible result.

\[ \text{Figure 4.6: Amplitude at point } (\frac{1}{2}, \frac{1}{2}) \text{ using the direct formulation 4.38 of the Taylor-Galerkin scheme using order 1 elements, element size } dx = 0.01, v = (1, 0). \]

\[ \text{Figure 4.7: Amplitude at point } (\frac{1}{2}, \frac{1}{2}) \text{ using the direct formulation 4.38 of the Taylor-Galerkin scheme using order 1 elements, element size } dx = 0.002, v = (1, 0). \]

4.4.3 Summary

The examples in this section have demonstrated the capabilities and limitations of both HRZ and row sum lumping with comparisons to the exact and full mass matrix solutions. Wave propagation type problems that utilise lumping, produce results similar to the full mass matrix at significantly lower computation cost. An acceleration based formulation, with HRZ lumping should be implemented for such problems, and can be applied to both order 1 and order 2 elements.

In transport type problems, it is essential that row sum lumping is used to achieve a smooth solution. Additionally, it is not recommended that second order elements be used in advection type problems.
Figure 4.8: Amplitude at point \( \left( \frac{1}{2}, \frac{1}{2} \right) \) using the direct formulation 4.38 of the Taylor-Galerkin scheme using order 2 elements, element size 0.01, \( v = (1, 0) \).
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 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[20].

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

5.2 The Unit Square

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 needs to form a closed loop. We now use the CurveLoop to define a surface

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

Note that there is a difference between the CurveLoop, which defines the boundary of the surface, and the actual surface PlaneSurface. This difference becomes clearer in the next example with a hole. Now we are ready to define the geometry which is described by an instance of the 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

```python
d.addSurface(s)
```
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\cite{gmsh} writes messages to the file .gmsh-errors in your home directory. Putting 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)
p11=PropertySet("sides",l01,l23)
p12=PropertySet("top_and_bottom",l12,l30)
d.addItems(p11, p12)
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 is not written into the mesh file mainly to reduce the size of the data file. One needs 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\cite{gmsh} installed you can run the example and view the geometry and mesh with:

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

See Figure 5.1 for a result.

In most cases it is best practice to generate the mesh and solve the mathematical model in two separate scripts. In our example you can read the esys.finley mesh into your simulation code\footnote{Gmsh\cite{gmsh} files can be directly read using ReadGmsh, see Chapter 7} using

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

Note that the underlying mesh generation software will not accept all the geometries you can create. 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.

\footnote{Gmsh\cite{gmsh} files can be directly read using ReadGmsh, see Chapter 7}
5.3 Holes

The example included below shows how to use \texttt{esys.pycad} to create a 2-D mesh in the shape of a trapezoid with a cut-out area as in Figure 5.2.

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

# A trapezoid
p0=Point(0.0, 0.0, 0.0)
p1=Point(1.0, 0.0, 0.0)
p2=Point(1.0, 0.5, 0.0)
p3=Point(0.0, 1.0, 0.0)
l01=Line(p0, p1)
l12=Line(p1, p2)
l23=Line(p2, p3)
l30=Line(p3, p0)
c=CurveLoop(l01, l12, l23, l30)

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

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

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

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

# Create the geometry, mesh and Escript domain
d.setScriptFileName("trapezoid.geo")
d.setMeshFileName("trapezoid.msh")
domain=MakeDomain(d)
```

Chapter 5. The \texttt{esys.pycad} Module
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 define 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. As an example consider the unit cube as shown in 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))
```

Similar to the definition of a CurvedLoop the orientation of the surfaces in a SurfaceLoop is relevant. In fact, the surface normal direction defined by 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. Similarly we set

```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:
To form the front face we encounter the problem as the line 145 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 -145 to indicate that the direction is to be reversed. With this notation we can write

\[
\text{front=PlaneSurface(CurveLoop(101,115,-145,140))}
\]

Keep in mind that if you use Line(p4,p5) instead of -145 both objects are treated as different although connecting the same points with a straight line in the same direction. The resulting geometry would include an opening along the p4–p5 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:

\[
\begin{align*}
137&=\text{Line(p3,p7)} \\
162&=\text{Line(p6,p2)} \\
\text{back}&=\text{PlaneSurface(CurveLoop(132,-162,-176,-137))} \\
\text{left}&=\text{PlaneSurface(CurveLoop(-140,-164,162,120))} \\
\text{right}&=\text{PlaneSurface(CurveLoop(-115,113,137,-157))}
\end{align*}
\]

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

\[
\begin{align*}
\text{sl}&=\text{SurfaceLoop(top,bottom,front,back,left,right)} \\
v&=\text{Volume(sl)}
\end{align*}
\]

As in the 2D case, the Design class is used to define the geometry:

```python
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

esys.pycad supports other file formats including I-DEAS universal file, VRML, Nastran and STL. The following example shows how to generate the STL file brick.stl:
Figure 5.4: Local refinement at the origin by local_scale=0.01 with element_size=0.3 and number of elements on the top set to 10

```
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 a finer mesh locally. A local refinement can be defined at each `Point`:

```
p0=Point(0., 0., 0., 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 `element_size=0.3`, see Figure 5.4. The point where a refinement is defined must be a point on a 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 elements on line `l23`:

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

Setting the number of elements on a curve overwrites the global mesh size `element_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., local_scale=1. )
    creates a point at the given coordinates with local characteristic length local_scale
```
class CurveLoop(list)
creates a closed curve from a list of Line, Arc, Spline, BSpline, BezierSpline objects.

class SurfaceLoop(list)
creates a loop of PlaneSurface or RuledSurface, which defines the shell of a volume.

Lines

class Line(point1, point2)
creates a line between two points.

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.

ggetElementDistribution()
returns the element distribution as a 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[, progression=1[, createBump=False ]])
defines the number of elements on the spline. 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 spline.

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

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

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 curve. 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 curve.

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

**getElementDistribution()**
returns the element distribution as a tuple of number of elements, progression factor and bump flag. If no element distribution is set None is returned.

### Bezier Curves

class BezierCurve(point0, point1, ...)
A Bezier spline curve defined by a list of points point0, point1, ...

**setElementDistribution(n[, progression=1[, createBump=False ]])**
defines the number of elements on the curve. 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 curve.

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

**getElementDistribution()**
returns the element distribution as a tuple of number of elements, progression factor and bump flag. If no element distribution is set None is returned.

### Arcs

class Arc(centre_point, start_point, end_point)
creates 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 arc. 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 arc.

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

**getElementDistribution()**
returns the element distribution as a tuple of number of elements, progression factor and bump flag. If no element distribution is set None is returned.

### Plane surfaces

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

**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. \texttt{max\_deviation}
(in radians) defines the maximum deviation of any angle in the quadrilaterals from the right angle. Set \texttt{max\_deviation = None} to remove recombination.

\begin{verbatim}
setTransfiniteMeshing([ orientation="Left" ])
\end{verbatim}

applies 2D transfinite meshing to the surface. \texttt{orientation} defines the orientation of triangles. Allowed values are "Left", "Right" and "Alternate". The boundary of the surface must be defined by three or four lines and an element distribution must be defined on all faces where opposite faces use the same element distribution. No holes must be present.

\textbf{Ruled Surfaces}

\begin{verbatim}
class RuledSurface(list)
\end{verbatim}

creates a surface that can be interpolated using transfinite interpolation. \texttt{list} gives a list of three or four lines defining the boundary of the surface.

\begin{verbatim}
setRecombination([ max\_deviation=45 * DEG ])
\end{verbatim}

the mesh generator will try to recombine triangular elements into quadrilateral elements. \texttt{max\_deviation}
(in radians) defines the maximum deviation of any angle in the quadrilaterals from the right angle. Set \texttt{max\_deviation = None} to remove recombination.

\begin{verbatim}
setTransfiniteMeshing([ orientation="Left" ])
\end{verbatim}

applies 2D transfinite meshing to the surface. \texttt{orientation} defines the orientation of triangles. Allowed values are "Left", "Right" and "Alternate". The boundary of the surface must be defined by three or four lines and an element distribution must be defined on all faces where opposite faces use the same element distribution. No holes must be present.

\begin{verbatim}
setElementDistribution(n[, progression=1[, createBump=False ]])
\end{verbatim}

defines the number of elements on all lines.

\textbf{Volumes}

\begin{verbatim}
class Volume(loop, [ holes=[list] ])
\end{verbatim}

creates a volume given a \texttt{SurfaceLoop}, which may have one or more holes define by the list of \texttt{SurfaceLoop}.

\begin{verbatim}
setElementDistribution(n[, progression=1[, createBump=False ]])
\end{verbatim}

defines the number of elements on all lines.

\begin{verbatim}
setRecombination([ max\_deviation=45 * DEG ])
\end{verbatim}

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. \texttt{max\_deviation} (in radians) defines the maximum deviation of any angle in the quadrilaterals from the right angle. Set \texttt{max\_deviation = None} to remove recombination.

\begin{verbatim}
setTransfiniteMeshing([ orientation="Left" ])
\end{verbatim}

applies transfinite meshing to the volume and all surfaces (if \texttt{orientation} is not equal to \texttt{None}). \texttt{orientation} defines the orientation of triangles. Allowed values are "Left", "Right" and "Alternate". The boundary of the surface must be defined by three or four lines and an element distribution must be defined on all faces where opposite faces use the same element distribution. If \texttt{orientation} is equal to \texttt{None} transfinite meshing is not switched on for the surfaces but needs to be set by the user. No holes must be present. \textbf{Warning: The functionality of transfinite meshing without recombination is not entirely clear in Gmsh[20]. So please apply this method with care.}
5.7.2 Transformations

Sometimes it is convenient to create an object and then make copies at different orientations or in different sizes. This can be achieved by applying transformations which are used to move geometrical objects in the 3-dimensional space and to resize them.

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

class Rotation([ axis=[1,1,1], [ point = [0,0,0], [ angle=0*RAD ] ] ])
defines a rotation by angle\( \) around axis through point point\( \) and direction axis. axis and point\( \) can be any object that can be converted into a numpy object of shape \((3,\)\). axis does not have to be normalised but must have positive length. The right-hand rule [7] applies.

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

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

DEG

a constant to convert from degrees to an internal angle representation in radians. For instance use \( 90 \cdot \text{DEG} \) 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. 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 which share the same material properties or to mark portions of the boundary. For efficiency, the Design class assigns an integer to each of its Property Sets, a so-called tag. The appropriate tag is attached to the elements at generation time.

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

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

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

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

getName()
returns the name of the set

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

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

```python
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 `Gmsh`[20]. This interface could be adopted to `triangle` or another mesh generation package if this is deemed to be desirable in the future.

```python
describes the geometry defined by primitives to be meshed. dim specifies the spatial dimension, while 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**
  - gmsh file format [20]

- **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 Delaunay triangulator, see `Gmsh`[20] and [33]

- **MESHADAPT**
  - the gmsh triangulator, see `Gmsh`[20]
FRONTAL
the NETGEN [18] triangulator

generate()
generates the mesh file. The data 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.

ggetElementOrder()
returns the element order.

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

ggetElementSize()
returns the global element size.

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

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

keepFiles()
returns True if work files are kept, False otherwise.

setScriptFileName([ name=None ])
sets the file name for the gmsh input script. If no name is given a name with extension ”geo” is generated.

ggetScriptFileName()
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.

ggetMeshFileName()
returns the name of the mesh file.

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

getItems()
returns a list of the items.
clearItems()
    resets the items in this design.

getMeshHandler()
    returns a handle to the mesh. Calling 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 a string.

setOptions( [ optimize_quality=\texttt{True} [ , smoothing=1 [ , curvature_based_element_size=\texttt{False} [ ,
algorithm2D=\texttt{Design.MESHADAPT} [ , algorithm3D=\texttt{Design.FRONTAL} [ ,
sets options for the mesh generator. algorithm2D sets the 2D meshing algorithm to be used. The al-
gorithm needs to be \texttt{Design.DELAUNAY}, \texttt{Design.FRONTAL}, or \texttt{Design.MESHADAPT}. By default
\texttt{Design.MESHADAPT} is used. algorithm3D sets the 3D meshing algorithm to be used. The algo-
rithm needs to be \texttt{Design.DELAUNAY} or \texttt{Design.FRONTAL}. By default \texttt{Design.FRONTAL} is used.
optimize_quality = \texttt{True} invokes an optimization of the mesh quality. smoothing sets the number
of smoothing steps to be applied to the mesh. curvature_based_element_size = \texttt{True} switches
on curvature based definition of element size. generate_hexahedra = \texttt{True} switches on the usage of
quadrilateral or hexahedral elements.

getTagMap()
    returns a \texttt{TagMap} to map the \texttt{PropertySet} names to tag numbers generated by gmsh.

setFileFormat( [ \texttt{format=Design.GMSH} ]
sets the file format. \texttt{format} must be one of the values:
Design.GMSH
Design.IDEAS
Design.VRML
Design.STL
Design.NASTRAN
Design.MEDIT
Design.CGNS
Design.PLOT3D
Design.DIFFPACK.

getFileFormat()
    returns the file format.
5.8. Interface to the mesh generation software
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. 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

$$-\left(\eta(v_{i,j} + v_{j,i})\right)_j + p_i = f_i - \sigma_{ij,j}$$

(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 medium:

$$-v_{i,i} = 0$$

(6.2)

Natural boundary conditions are taken in the form

$$\left(\eta(v_{i,j} + v_{j,i})\right)n_j - n_i p = s_i - \alpha \cdot n_i n_j v_j + \sigma_{ij} n_j$$

(6.3)

which can be overwritten by constraints of the form

$$v_i(x) = v_i^D(x)$$

(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. $v^D$ 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 a coercive, self-adjoint linear operator in a suitable Hilbert space, $B$ is the $(-1)$-divergence operator and $B^*$ is the 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 operator \( A \) is applied. 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

\[
\|v\|^2_2 = \int_\Omega v_{j,k}v_{j,k} \, dx \quad \text{and} \quad \|p\|^2_2 = \int_\Omega p^2 \, dx.
\]  

(6.8)

for velocity \( v \) and pressure \( p \). The iteration is terminated if the stopping criterion

\[
\max(\|Bv_1\|_0, \|v_2 - v_0\|_1) \leq \tau \cdot \|v_2\|_1
\]  

(6.9)

for a given tolerance \( 0 < \tau < 1 \) is met. 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

\[
v_1 = e_1 + v_0 + A^{-1}(G - Av_0 - B^*p_0)
\]  

(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 \( \|\cdot\| \) used in the stopping criterion for this equation. In fact we will have a stopping criterion of the form

\[
\|Ae_1\|_s = \|G - Av_1 - B^*p_0\|_s \leq \tau_1 \|G - Av_0 - B^*p_0\|_s
\]  

(6.11)

where \( \tau_1 \) is the tolerance which we need to choose. This translates into the condition

\[
\|e_1\|_1 \leq K \tau_1 \|dv_1 - e_1\|_1
\]  

(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

\[
p_2 = p_0 + (BA^{-1}B^*)^{-1}(e_2 + Bv_1)
\]  

(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 \( \frac{1}{\eta} [16] \), i.e. the evaluation of the preconditioner \( P \) for a given pressure increment \( q \) is the solution of

\[
\frac{1}{\eta}(Pq) = q.
\]  

(6.14)

Note that in each evaluation of the iteration operator \( q = BA^{-1}B^*s \) one needs to solve the problem

\[
Aw = B^*s
\]  

(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

\[
r = B(v_1 - A^{-1}B^*dp) = B(v_1 - A^{-1}B^*dp) = B(v_1 - dv_2) = Bv_2
\]  

(6.16)
In particular we have $v_2 = Bv_2$. 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

$$\|P^{\frac{1}{2}}Bv_2\|_0 \leq \tau_2\|P^{\frac{1}{2}}Bv_1\|_0$$

(6.17) where $\tau_2$ is the given tolerance. This translates into

$$\|e_2\|_0 = \|Bv_2\|_0 \leq M\tau_2\|Bv_1\|_0$$

(6.18) where $M$ is taking care of the fact that $P^{\frac{1}{2}}$ is dropped.

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

$$v_2 = v_1 - dv_2 = v_1 - A^{-1}B^*dp$$

(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

$$v = \Phi(v, p) \text{ and } p = \Psi(u, p)$$

(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

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

$$\delta v = e_1 - A^{-1}B^*\delta p.$$  

(6.21)

Notice that $B\delta v = -e_2 = -Bv_2$. 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

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

(6.22)

In practice using the fact that $BA^{-1}B^*(p_2 - p_0) = Bv_1$ and assuming that $v_2$ gives a better approximation to the true $v$ than $v_0$ we will use the estimate

$$\epsilon = \max(\|v_2 - v_0\|_1, \|Bv_1\|_0)$$

(6.23)

to estimate the progress of the iteration step after the step is completed. Note that the estimate of $\epsilon$ is used in the stopping criterion 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:

$$\|BA^{-1}B^*(p_2 - p)\|_0 \leq \|BA^{-1}B^*(p_2 - \delta p - p)\|_0 + \|BA^{-1}B^*\delta p\|_0$$

$$= \chi^{-} \cdot \epsilon^{-} + \|e_2 + Be_1 \|_0$$

$$\approx \chi^{-} \cdot \epsilon^{-} + \|e_2\|_0$$

$$\leq \chi^{-} \cdot \epsilon^{-} + M\tau_2\|Bv_1\|_0$$

(6.24)

So we choose the value for $\tau_2$ from

$$M\tau_2\|Bv_1\|_0 \leq (\chi^{-})^2\epsilon^{-}$$

(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:

$$\|v_2 - v\|_1 \leq \|v_2 - \delta v - v\|_1 + \|\delta v\|_1$$

$$\leq \chi^{-} \cdot \epsilon^{-} + \|e_1 - A^{-1}B^*\delta p\|_1$$

$$\approx \chi^{-} \cdot \epsilon^{-} + \|e_1\|_1$$

$$\leq \chi^{-} \cdot \epsilon^{-} + K\tau_1\|dv_1 - e_1\|_1$$

(6.26)

So we choose the value for $\tau_1$ from

$$K\tau_1 \leq \chi^{-}$$

(6.27)
Assuming we have estimates for $M$ and $K^1$, 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 $\chi = \frac{\epsilon}{\epsilon - \chi}$. 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 were suitable. Otherwise, we need to adjust the values for $K$ and $M$. From the estimates 6.24 and 6.26 we establish

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

(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

$$\chi = (1 + \max(M^+ \frac{\chi^-}{M}, K^+ \frac{\chi^-}{K})) \cdot \chi^-$$

(6.29)

From this equation we see if 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

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

(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 criterion 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 Equation (6.27)
3. solve Equation (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:
   - update $p_2 \leftarrow p$ and $v_2 \leftarrow v_1$
7. calculate convergence measure $\epsilon$ and convergence rate $\chi$
8. if stopping criterion 6.9 holds:
   - return $v_2$ and $p_2$
9. else:
   (a) update $M$ and $K$
   (b) goto step 1 with $v_0 \leftarrow v_2$ and $p_0 \leftarrow p_2$.

1 if no estimates are available, we use the value 1

6.1. The Stokes Problem
6.1.2 Functions

class StokesProblemCartesian(domain)

opens the Stokes problem on the Domain domain. The domain needs to support LBB compliant elements for the Stokes problem, see [21] 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 polynomials 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 ways is expressed by

```python
velocity=Vector(0.0, Solution(mesh))
presure=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 \( \eta \), \( \text{surface stress} \) the surface stress \( s \) and \( \text{stress} \) the initial stress \( \sigma \). The locations and components where the velocity is fixed are set by the values of \( \text{fixed_u_mask} \). \( \text{restoration_factor} \) defines the restoring force factor \( \alpha \). The method will try to cast the given values to appropriate Data class objects.

```python
solve(v,p [, max_iter=100 [, verbose=False [, usePCG=True ]]])
```

solves the problem and returns approximations for velocity and pressure. The arguments \( v \) and \( p \) define initial guesses. \( v \) must have function space Solution(domain) and \( p \) must have function space ReducedSolution(domain). The values of \( v \) marked by \( \text{fixed_u_mask} \) remain unchanged. If \( \text{usePCG} \) is set to \( \text{True} \) then the preconditioned conjugate gradient method (PCG) scheme is used. Otherwise the problem is solved with the generalized minimal residual method (GMRES). In most cases the PCG scheme is more efficient. \( \text{max_iter} \) defines the maximum number of iteration steps. If \( \text{verbose} \) is set to \( \text{True} \) information on the progress of of the solver is printed.

```python
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.

```python
getTolerance()
```

returns the current relative tolerance.

```python
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.

```python
getAbsoluteTolerance()
```

returns the current absolute tolerance.

```python
getSolverOptionsVelocity()
```

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

```python
getSolverOptionsPressure()
```

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

```python
getSolverOptionsDiv()
```

sets the solver options for solving the equation to project the divergence of the velocity onto the function space of pressure.
6.1.3 Example: Lid-driven Cavity

The following script `lid_driven_cavity.py` which is available in the example directory illustrates the usage of the StokesProblemCartesian class to solve the lid-driven cavity problem:

```python
from esys.escript import *
from esys.finley import Rectangle
from esys.weipa import saveVTK
from esys.escript.models import StokesProblemCartesian

NE=25
dom = Rectangle(NE,NE,order=2)
x = dom.getX()
scc=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]
scc.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=scc.solve(v, p, verbose=True)
saveVTK("u.vtu", 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

\[
\begin{align*}
\forall i: \quad u_i + \kappa_{ij} p_j &= g_i \\
u_{k,k} &= f
\end{align*}
\]  \( (6.33) \)

with the boundary conditions

\[
\begin{align*}
\forall \Gamma_N: \quad u_i n_i &= u_i^N n_i \\
p &= p^D \quad \text{on } \Gamma_D
\end{align*}
\]  \( (6.34) \)

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 \( \Omega \), \( u_i^N \) 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

\[
\alpha_0 x_i x_i \leq \kappa_{ij} x_j x_i \leq \alpha_1 x_i x_i
\]  \( (6.35) \)

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 for velocity \( v \) we set

\[
Dv = v_{k,k}
\]  \( (6.37) \)

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

\[
\begin{align*}
u + KGp &= g \\
Du &= f
\end{align*}
\]  \( (6.38) \)

We solve this equation by minimising the functional

\[
J(u,p) := \|K^{-\frac{1}{2}}(u + KGp - g)\|^2_0 + \|\lambda(Du - f)\|^2_0
\]  \( (6.39) \)
over all suitable $u$ and $p$. In this equation we set $\|p\|_0^2 = (p, p)_0$ with

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

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

$$(K^{-1}(v + K G q), u + K G p - g)_0 + (\lambda D v, \lambda(D u - f))_0 = 0$$

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

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

$$G* u + G*K G p = G* g$$

(6.42)

where $D*$ and $Q*$ denote the adjoint operators with respect to $\langle \cdot, \cdot \rangle_0$. In [30] it has been shown that this problem is continuous and coercive and therefore has a unique solution. 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

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

(6.43)

which is inserted into the second equation

$$G*(K^{-1} + \lambda D* D)^{-1}(\lambda D* f + K^{-1} g - G p) + G*K G p = G* g$$

(6.44)

which is

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

(6.45)

We use the PCG method to solve this. The residual $r$ is given as

$$r = G*(g - K G p - v)$$

(6.46)

for the current pressure approximation $p$ and current velocity $v$ defined by Equation (6.43). So in a particular implementation we use $\tilde{r} = g - K G p - v$ 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

$$(K^{-1} + D* \lambda^2 D)v = G p$$

(6.47)

To derive a preconditioner we use the identity

$$G*(K - (K^{-1} + D* \lambda^2 D)^{-1}) G = G*(I - (I + K D* \lambda^2 D)^{-1}) K G$$

$$\approx G*(K D* \lambda^2 D) K G$$

$$= G*K(D* \lambda^2 D) K G$$

(6.48)

$$\approx G* \frac{\lambda^2}{dx^2} K^2 G$$

where $dx$ is the local mesh size and we use the approximation

$$D* \lambda^2 D \approx \frac{\lambda^2}{dx^2}$$

(6.49)

Therefore we use $G* \frac{\lambda^2}{dx^2} K^2 G$ as a preconditioner. To evaluate the preconditioner we need to solve the equation

$$G* \frac{\lambda^2}{dx^2} K^2 G p = G* \tilde{r}$$

(6.50)

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 $(\tilde{u}, \tilde{p})$ which is a perturbed exact solution $(u, p)$. Assuming $\tilde{u} = u + u_0 e^{ik' x}$ and $\tilde{p} = p + p_0 e^{ik' x}$ and constant $K$ we get

$$J(\tilde{u}, \tilde{p}) = C \left( \|K^{-1} u_0\|^2 + \|K \|_2 \|k u_0\|^2 \right) + \lambda^2 \|k u_0\|^2$$

(6.51)

\textsuperscript{2}notice that $K^{-1} + \lambda D* D$ is coercive
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 $\|K^{-1}\|_2 = \lambda^2 \|k\|_2^2$. 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

$$\lambda = \|K^{-1}\|_2 \frac{1}{\pi}$$ (6.52)

Notice that with this setting the preconditioner $G^* \frac{\lambda^2}{dx^2} K^2 G$ 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 PCG is given as

$$\|r\|_{PCG}^2 = \int r(G^* \frac{\lambda^2}{dx^2} K^2 G)^{-1} r \, dx = \int G(G^* \frac{\lambda^2}{dx^2} K^2 G)^{-1} G^* \hat{r} \, dx \approx \int K^{-1} \hat{r} \, dx$$ (6.53)

The iteration is terminated if

$$\|r\|_{PCG} \leq ATOL$$ (6.54)

where we set

$$ATOL = atol + rtol \cdot \left( \frac{1}{\|K^{-1}v\|_0} + \frac{1}{\|K^2 G p\|_0} \right)^{-1}$$ (6.55)

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 can 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.

### 6.2.2 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 need to be reset. The permeability can be defined as a 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 this SolverOptions object to control the solution algorithms.

**getSolverOptionsPressure()**
returns a SolverOptions object with the options used to solve the pressure problem (6.50) as a preconditioner. Use this 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 guesses 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 iteration steps allowed for solving the coupled problem.

### 6.3 Isotropic Kelvin Material

As proposed by Kelvin [26] material strain $D_{ij} = \frac{1}{2} (v_{i,j} + v_{j,i})$ can be decomposed into an elastic part $D_{ij}^e$ and a visco-plastic part $D_{ij}^{vp}$:

$$D_{ij} = D_{ij}^e + D_{ij}^{vp}$$

(6.56) with the elastic strain given as

$$D_{ij}^e = \frac{1}{2\mu} \sigma_{ij}'$$

(6.57) where $\sigma_{ij}'$ is the deviatoric stress (notice that $\sigma_{ii}' = 0$). If the material is composed by materials $q$ the visco-plastic strain can be decomposed as

$$D_{ij}^{vp} = \sum_q D_{ij}^q$$

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

$$D_{ij}^q = \frac{1}{2\eta^q} \sigma_{ij}'$$

(6.59) and $\eta^q$ is the viscosity of material $q$. We assume the following between the strain in material $q$

$$\eta^q = \eta_N^q \left( \frac{\tau}{\tau_t^q} \right)^{1-n^q}$$

with $\tau = \sqrt{\frac{1}{2} \sigma_{ij}' \sigma_{ij}'}$ (6.60) for given power law coefficients $n^q \geq 1$ and transition stresses $\tau_t^q$, see [26]. Notice that $n^q = 1$ gives a constant viscosity. After inserting Equation (6.59) into Equation (6.58) one gets:

$$D_{ij}^{vp} = \frac{1}{2\eta^{vp}} \sigma_{ij}'$$

(6.61) with

$$\eta_v = \frac{1}{\eta^q} \sum_q \frac{1}{\eta^q}.$$  

(6.62)

and finally with 6.56

$$D_{ij}^e = \frac{1}{2\eta^{vp}} \sigma_{ij}' + \frac{1}{2\mu} \dot{\sigma}_{ij}'$$

(6.63) The total stress $\tau$ needs to fulfill the yield condition

$$\tau \leq \tau_Y + \beta p$$

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

$$-\sigma_{ij,j} + p_i = F_i$$

(6.65) where $F_j$ is a given external force. We assume an incompressible medium:

$$-v_{i,i} = 0$$

(6.66) Natural boundary conditions are taken in the form

$$\sigma_{ij}' n_j - n_i p = f$$

(6.67) which can be overwritten by a constraint

$$v_i(x) = 0$$

where the index $i$ may depend on the location $x$ on the boundary.

---

Chapter 6. Models 107
6.3.1 Solution Method

By using a first order finite difference approximation with step size \( dt > 0 \) Equation (6.57) is transformed to

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

and

\[
D_{ij}' = \left( \frac{1}{2\eta^{\nu p}} + \frac{1}{2\mu dt} \right) \sigma_{ij}' - \frac{1}{2\mu dt} \sigma_{ij}''
\]

where \( \sigma_{ij}'' \) is the stress at the previous time step. With

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

we have

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

where

\[
\eta_{eff} = \min\left(\frac{1}{\mu dt} + \frac{1}{\eta^{\nu p}}, \eta_{max}\right)
\]

with \( \eta_{max} = \left\{ \begin{array}{ll} \frac{\tau + \beta}{\gamma} & \text{if } \dot{\gamma} > 0 \\ \infty & \text{otherwise} \end{array} \right. \)

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

\[
\sigma_{ij}' = 2\eta_{eff} \left( D_{ij}' + \frac{1}{2\mu dt} \sigma_{ij}'' \right)
\]

After inserting 6.73 into 6.64 we get

\[
-(\eta_{eff}(v_{i,j} + v_{j,i}))_j + p_i = F_i + \left( \frac{\eta_{eff}}{\mu dt} \sigma_{ij}'' \right)_j
\]

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

\[
\eta' = \frac{\Theta'}{\eta^{\nu p}} \text{ with } \Theta' = \sum_q \left( \frac{1}{\eta_q} \right)'\]

we need to solve the nonlinear problem

\[
\eta_{eff} - \min(\eta(\dot{\gamma} \cdot \eta_{eff}), \eta_{max}) = 0
\]

We use the Newton-Raphson scheme to solve this problem:

\[
\eta_{eff}^{(n+1)} = \min(\eta_{max}, \eta_{eff}^{(n)} - \frac{\eta_{eff}^{(n)} - \eta(\tau^{(n)})}{1 - \dot{\gamma} \cdot \eta'(\tau^{(n)})}) = \min(\eta_{max}, \frac{\eta(\tau^{(n)}) - \tau^{(n)} \cdot \eta'(\tau^{(n)})}{1 - \dot{\gamma} \cdot \eta'(\tau^{(n)})})
\]

where \( \eta' \) denotes the derivative of \( \eta \) with respect to \( \tau \) and \( \tau^{(n)} = \dot{\gamma} \cdot \eta_{eff}^{(n)} \). Looking at the evaluation of \( \eta \) in 6.75 it makes sense to formulate the iteration 6.77 using \( \Theta = \eta^{-1} \). In fact we have

\[
\eta' = -\frac{\Theta'}{\eta^{\nu p}} \text{ with } \Theta' = \sum_q \left( \frac{1}{\eta_q} \right)'\]

As

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

we have

\[
\Theta' = \frac{1}{\tau} \omega \text{ with } \omega = \sum_q \frac{n^q - 1}{\eta^{\nu p}}
\]

which leads to

\[
\eta_{eff}^{(n+1)} = \min(\eta_{max}, \eta_{eff}^{(n)} \frac{\Theta^{(n)} + \omega^{(n)}}{\eta_{eff}^{(n)} \Theta^{(n)^2} + \omega^{(n)}})
\]
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 is 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 [21] 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 polynomials for both pressure and velocity but with a subdivided element for the velocity. Typically, the macro element method is more cost effective. The fact that pressure and velocity are represented in different ways is expressed by

\[
\begin{align*}
\text{velocity} &= \text{Vector}(0.0, \text{Solution}(\text{mesh})) \\
\text{pressure} &= \text{Scalar}(0.0, \text{ReducedSolution}(\text{mesh}))
\end{align*}
\]

getDomain()
returns the domain.

getTime()
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.

getGammaDot()
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 relative of the effective viscosity is less than $\text{rtol}$.

```
setDruckerPragerLaw([ tau_Y=None, [ friction=None ] ])
```
sets the parameters $\tau_Y$ and $\beta$ for the Drucker-Prager model in condition 6.63. If $\text{tau}_Y$ is set to None (default) then the Drucker-Prager condition is not applied.

```
setElasticShearModulus(mu=None)
```
sets the elastic shear modulus $\mu$. If $\text{mu}$ 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 $\tau^q$, and $\text{power}$ 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 $\text{iter}_\text{max}$ is the maximum number of iteration steps on a time step to update the effective viscosity and $\text{inner}_\text{iter}_\text{max}$ is the maximum number of iteration steps in the incompressible solver.

### 6.4 Fault System

The `FaultSystem` class provides 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 faults. 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 starting point $V^{t0}$ and series of directions, called strikes, and the 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

\[
\begin{bmatrix}
-\sin(\theta^ti) \cdot S^t_1 \\
\sin(\theta^ti) \cdot S^t_0 \\
\cos(\theta^ti)
\end{bmatrix}
\]  

(6.83)

At each vertex we define a depth vector \(d^ti\) 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 \(\vec{d}^ti\) of the depth vector is given as

\[
\vec{d}^ti = n^ti \times n^{t(i-1)}
\]  

(6.84)

If \(\vec{d}^ti\) is zero the strike vectors \(L^t(i-1)\) and \(L^ti\) are collinear and we can set \(\vec{d}^ti = l^ti \times n^ti\). If the two fault segments are almost orthogonal \(\vec{d}^ti\) 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

\[
\min(\|\vec{d}^ti \times L^t(i-1)\|, \|\vec{d}^ti \times L^ti\|) \leq 0.1 \cdot \|\vec{d}^ti\|
\]  

(6.85)

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

\[
d^ti = \delta^ti \cdot \frac{\vec{d}^ti}{\|\vec{d}^ti\|}
\]  

(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

\[
0 \leq w_0 \leq w^t_{0\text{max}} \quad \text{and} \quad -w^t_{1\text{max}} \leq w_1 \leq 0
\]  

(6.88)

with positive numbers \(w^t_{0\text{max}}\) and \(w^t_{1\text{max}}\). Typically one chooses \(w^t_{0\text{max}}\) to be the unrolled length of the fault and \(w^t_{1\text{max}}\) to be the mean value of segment depth. Moreover we have

\[
P^t(W^ti) = V^ti \quad \text{and} \quad P^t(v^ti) = v^ti
\]  

(6.89)

where

\[
W^ti = (\Omega^ti, 0) \quad \text{and} \quad v^ti = (\Omega^ti, -w^t_{1\text{max}})
\]  

(6.90)

and \(\Omega^ti\) is the unrolled distance of \(W^ti\) from \(W^t0\), i.e. \(l^ti = \Omega^t(i+1) - \Omega^ti\). In the 2D case \(w^t_{1\text{max}}\) is set to zero and therefore the second component is dropped, see Figure 6.1.

In the 2D case the parameterization \(P^t\) is constructed as follows: The line connecting \(V^t(i-1)\) and \(V^ti\) is given by

\[
x = V^ti + s \cdot (V^t(i+1) - V^ti)
\]  

(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. Assuming \(x\) is on the fault it can be calculated as

\[
s = \frac{(x - V^ti) \cdot (V^t(i+1) - V^ti)}{\|V^t(i+1) - V^ti\|^2}
\]  

(6.92)

We then can set

\[
w_0 = \Omega^ti + s \cdot (\Omega^ti - \Omega^t(i-1))
\]  

(6.93)

to get \(P^t(w_0) = x\). It remains the question if the given \(x\) is actually 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 then we check the residual of Equation (6.91), i.e. \(x\) has been accepted to be in the segment if

\[
\|x - V^ti - s \cdot (V^t(i+1) - V^ti)\| \leq tol \cdot \max(l^ti, \|x - V^ti\|)
\]  

(6.94)

where \(tol\) is a given tolerance.

Chapter 6. Models

111
In the 3D case the situation is a bit more complicated: we split the fault segment across the diagonal \( V^{ti} - V^{t(i+1)} \) to produce two triangles. In the upper triangle we use the parameterization

\[
x = V^{ti} + s \cdot (V^{t(i+1)} - V^{ti}) + r \cdot (v^{t(i+1)} - V^{t(i+1)}) \quad \text{with} \quad r \leq s;
\]

while in the lower triangle we use

\[
x = V^{ti} + s \cdot (v^{t(i+1)} - v^{ti}) + r \cdot (v^{ti} - V^{ti}) \quad \text{with} \quad s \leq r;
\]

where \( 0 \leq s, r \leq 1 \). Both equations are solved in the least-squares sense e.g. using the Moore-Penrose pseudo-inverse 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

\[
\|x - V^{ti} - s \cdot (V^{t(i+1)} - V^{ti}) - r \cdot (v^{t(i+1)} - V^{t(i+1)})\| \leq tol \cdot \max(\|x - V^{ti}\|, \|v^{t(i+1)} - V^{t(i)}\|) \quad (6.97)
\]

and in the lower part

\[
\|x - V^{ti} - s \cdot (v^{t(i+1)} - v^{ti}) - r \cdot (v^{ti} - V^{ti})\| \leq tol \cdot \max(\|x - V^{ti}\|, \|v^{t(i+1)} - V^{t(i)}\|) \quad (6.98)
\]

after the restriction of \((s, t)\) to the unit square. Note that \( \|v^{t(i+1)} - V^{t(i)}\| \) is the length of the diagonal of the fault segment. For those \( x \) which have been located in the \( i \)-th segment we then set

\[
w_0 = \Omega^{ti} + s \cdot (\Omega^{ti} - \Omega^{t(i-1)}) \quad \text{and} \quad w_1 = w_{1\max}^i (r - 1) \quad (6.99)
\]

### 6.4.1 Functions

```python
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 the 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 the segments in fault \( t = tag \).

getTotalLength(tag)
```

112 6.4. Fault System
returns the total unrolled length of fault \( t \).

\textbf{getDips(\( t \))}
returns the list of the dips of the segments in fault \( t \).

\textbf{getBottomPolyline(\( t \))}
returns the list of the vertices defining the bottom of the fault \( t \).

\textbf{getSegmentNormals(\( t \))}
returns the list of the normals of the segments in fault \( t \).

\textbf{getDepthVectors(\( t \))}
returns the list of the depth vectors \( d_i \) for fault \( t = t \).

\textbf{getDepths(\( t \))}
returns the list of the depths of the segments in fault \( t \).

\textbf{getW0Range(\( t \))}
returns the range of the parameterization in \( w_0 \). For tag \( t \) this is the pair \( (\Omega^0_t, \Omega^n_t) \) where \( n \) is the number of segments in the fault. In most cases one has \( (\Omega^0_t, \Omega^n_t) = (0, w^t_{\text{max}}) \).

\textbf{getW1Range(\( t \))}
returns the range of the parameterization in \( w_1 \). For tag \( t \) this is the pair \( (-w^t_{\text{max}}, 0) \).

\textbf{getW0Offsets(\( t \))}
returns the offsets for the parameterization of fault \( t \). For tag \( t = t \) this is the list \([\Omega^i_t]\).

\textbf{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). A \texttt{numpy.ndarray} object is returned.

\textbf{getOrientationOnSurface()}
returns the orientation of the fault system in RAD on the surface \((x_2 = 0 \text{ plane})\) around the fault system center.

\textbf{transform([ \ rot=0, \ [ \ shift=numpy.zeros((3,) \] \])}
applies a shift \texttt{shift} and a consecutive rotation in the \( x_2 = 0 \) plane. \texttt{rot} is a float number and \texttt{shift} an \texttt{numpy.ndarray} object.

\textbf{getMaxValue(\( f \) , tol=1.e-8 )}
returns the tag of the fault where \( f \) takes the maximum value and a \texttt{Locator} object which can be used to collect values from \texttt{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 %s.\nloc(f), loc.getX())")
\end{verbatim}

\( f \) must be a scalar \texttt{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 are found the returned tag is \texttt{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 where \( f \) takes the minimum value and a \texttt{Locator} object which can be used to collect values from \texttt{Data} objects at the location where the minimum is taken, e.g.

```python
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())
```

\( f \) must be a scalar \texttt{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 are found the returned tag is \texttt{None} and the minimum value as well as the location of the minimum value are undefined.

getParametrization(x,tag [ , tol=1.e-8 ], outsider=None )
returns the argument \( w \) of the parameterization \( P^t \) for \( \texttt{tag}=t \) to provide \( x \) together with a mask indicating where the given location on a fault in the fault system by the value 1 (otherwise the value is set to 0).

\( x \) needs to be a \texttt{vector Data} object or \texttt{numpy.ndarray} object. \texttt{tol} defines the tolerance to decide if given data sample points are on fault \texttt{tag}. The value \texttt{outside} is the value used as a replacement value for \( w \) where the corresponding value in \( x \) is not on a fault. If \texttt{outside} is not present an appropriate value is used.

getSideAndDistance(x,tag)
returns the side and the distance at locations \( x \) from the fault \texttt{tag}. \( x \) needs to be a \texttt{vector Data} object or \texttt{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 \texttt{tag}. For \texttt{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 \texttt{numpy.ndarray} objects.

addFault( strikes[ , ls[ , V0=[0.,0.,0.][ , tag=None[ , dips=None[ , depths= None[ , w0_offsets=None[ , w1_ max=None ] ] ] ] ] ] ] ] ] ]
adds the fault \texttt{tag} to the fault system. \texttt{V0} defines the start point of fault named \( t = \texttt{tag} \). The polyline defining the fault segments on the surface are set by the strike angles \texttt{strikes} \( (=\sigma^t) \), \texttt{north} \( = \pi/2 \), the orientation is counterclockwise.) and the length \texttt{ls} \( (=l^t) \). In the 3D case one also needs to define the dip angles \texttt{dips} \( (=\delta^t) \), \texttt{vertical}=0, right-hand rule applies.) and the depth \texttt{depths} for each segment. \texttt{w1_max} defines the range of \( w_1 \). If not present the mean value over the depth of all segment edges in the fault is used. \texttt{w0_offsets} sets the offsets \( \Omega^t \). If not present it is chosen such that \( \Omega^t \Omega^{t-1} \) 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 1.6.
The \texttt{esys.finley} Module

\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 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 that has 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 u_{,l} + D \cdot u \, d\Omega + \int_{\Gamma} d \cdot v \, d\Gamma + \int_{\Gamma_{\text{contact}}} d_{\text{contact}} \cdot [v][u] \, 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 [v] \, d\Gamma
\end{equation}

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. quadrilaterals or, in the three-dimensional case, into tetrahedra and hexahedra. 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 its boundary to neighbour 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 [39, 13] for more details. \texttt{esys.finley} also supports macro elements. For these elements a piecewise linear approximation is used on an element which is further subdivided (in the case of \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 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 with 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-dimensional 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 need 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.

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 Line2>Contact. The nodes 9, 12, 6 and 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 Rec4Face<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 esys.finley file format is defined as follows. Each node $i$ has $dim$ spatial coordinates $Node[i]$, a reference number $Node\_ref[i]$, a degree of freedom $Node\_DOF[i]$ and a tag $Node\_tag[i]$. In most cases $Node\_DOF[i] = Node\_ref[i]$ however, for periodic boundary conditions, $Node\_DOF[i]$ is chosen differently, see example below. The tag can be used to mark nodes sharing the same properties. Element $i$ is defined by the $Element\_numNodes$ nodes $Element\_Nodes[i]$ which is a list of node reference numbers. The order of these is crucial. Each element has a reference number $Element\_ref[i]$ and a tag $Element\_tag[i]$. The tag can be used to mark elements sharing the same properties. For instance elements above a contact region are marked with tag 2 and elements below a contact region are marked with tag 1. $Element\_Type$ and $Element\_Num$ 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 esys.finley file format:

```python
print("%s\n"%mesh_name)
# node coordinates:
print("%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 %d\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
print("%s %d\n"%(FaceElement\_Type, FaceElement\_Num))
for i in range(FaceElement\_Num):
    print("%d %d"%(FaceElement\_ref[i], FaceElement\_tag[i]))
    for j in range(FaceElement\_numNodes): print(" %d"%FaceElement\_Nodes[i][j])
print("\n")
# contact elements
print("%s %d\n"%(ContactElement\_Type, ContactElement\_Num))
for i in range(ContactElement\_Num):
    print("%d %d"%(ContactElement\_ref[i], ContactElement\_tag[i]))
    for j in range(ContactElement\_numNodes): print(" %d"%ContactElement\_Nodes[i][j])
print("\n")
```

Figure 7.2: Mesh around a contact region (Rec4)
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 the function is to be calculated on faces and contacts, respectively.

```
# point sources (not supported yet)
print("Point1 0")
```

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 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 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 1 2 3 8 6
2 1 3 7 10 8
5 2 12 9 18 16
7 2 13 19 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 16 18 6 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 conditions 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 now reads:

```
2D Nodes 16
 0  0  0  0.  0.
 2  2  0  0.33  0.
 3  3  0  0.66  0.
 7  0  0  1.  0.
 5  5  0  0.  0.5
 6  6  0  0.33  0.5
 8  8  0  0.66  0.5
10 5  0  1.0  0.5
12 12  0  0.  0.5
 9  9  0  0.33  0.5
13 13  0  0.66  0.5
15 12  0  1.0  0.5
16 16  0  0.  0.5
18 18  0  0.33  1.0
19 19  0  0.66  1.0
20 16  0  1.0  1.0
```
\textbf{FIGURE 7.3:} Elements of order 1
Figure 7.4: Elements of order 2 and macro elements

Figure 7.5: Rec9 and Rec9Macro
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 elements. The nodes of a triangle and quadrilateral element are 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 defines 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 boundaries 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 uses 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, esys.finley uses the MKL solver package. If MKL is not available UMFPACK is used. If UMFPACK is not available a suitable iterative solver from PASO is used.

7.5 Functions

ReadMesh(fileName [, [integrationOrder=-1 ], optimize=True ])

creates a Domain object from the FEM mesh defined in file fileName. The file must be in the esys.finley file format. If integrationOrder is positive, a numerical integration scheme is chosen which is accurate on each element up to a polynomial of degree integrationOrder. Otherwise an

1If the stiffness matrix is non-regular MKL may return without a proper error code. If you observe suspicious solutions when using MKL, this may be caused by a non-invertible operator.
Table 7.2: Solvers available for esys.finley and the PASO package and the relevant options in SolverOptions. MKL supports MINIMUM_FILL_IN and NESTED_DISSECTION reordering. Currently the UMFPACK interface does not support any reordering.

<table>
<thead>
<tr>
<th>setSolverMethod</th>
<th>DIRECT</th>
<th>PCG</th>
<th>GMRES</th>
<th>TFQMR</th>
<th>MINRES</th>
<th>PRES20</th>
<th>BICGSTAB</th>
<th>lumping</th>
</tr>
</thead>
<tbody>
<tr>
<td>setReordering</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>setRestart</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>setMaxRestart</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td>setTruncation</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>setTolerance</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>setAbsoluteTolerance</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>setReordering</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
</tr>
</tbody>
</table>

Table 7.3: Preconditioners available for esys.finley and the PASO package and the relevant options in SolverOptions.

<table>
<thead>
<tr>
<th>setPreconditioner</th>
<th>NO_PRECONDITIONER</th>
<th>AMG</th>
<th>JACOBI</th>
<th>GAUSS_SEIDEL</th>
<th>REC_ILU</th>
<th>RILU</th>
<th>ILU0</th>
<th>DIRECT</th>
</tr>
</thead>
<tbody>
<tr>
<td>setLevelMax</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>setCoarseningThreshold</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>setMinCoarseMatrixSparsity</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>setNumSweeps</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>setNumPreSweeps</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>setNumPostSweeps</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>setRelaxationFactor</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
</tr>
</tbody>
</table>

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 [, [integrationOrder=-1], optimize=True [, useMacroElements=False ]])

creates a Domain object from the FEM mesh defined in file fileName. The file must be in the Gmsh[20] file format. If integrationOrder is positive, a numerical integration scheme is chosen which is accurate on each element up to a polynomial of degree 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 [, integrationOrder=-1 [, optimizeLabeling=True [, useMacroElements=False ] ]])

creates a esys.finley Domain object from a esys.pycad Design object using Gmsh[20]. The Design design defines the geometry. If integrationOrder is positive, a numerical integration scheme is chosen which is accurate on each element up to a polynomial of degree 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)

recovers a Domain object from a dump file fileName created by the dump method of a Domain object.

Rectangle(n0,n1.order=1,l0=1.,l1=1., integrationOrder=-1, periodic0=False, periodic1=False, useElementsOnFace=False, useMacroElements=False, 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 order =1 and order =2, elements of type 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 order =-1, Rec8Macro and Line3Macro are used. This option should be used when solving incompressible fluid flow problems, 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. If integrationOrder is positive, a numerical integration scheme is 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 $x_0$-direction are enforced. That means for any solution of a PDE solved by esys.finley the values on the line $x_0 = 0$ will be identical to the values on $x_0 = 10$. Correspondingly, periodic1 = True sets periodic boundary conditions in the $x_1$-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 $x_0$-axis, n1 elements along the $x_1$-axis and n2 elements along the $x_2$-axis. For order = 1 and order = 2, elements of type 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 are used. This option should be used when solving incompressible fluid flow problems, e.g. StokesProblemCartesian. If integrationOrder is positive, a numerical integration scheme is 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 $x_0$-direction are enforced. That means for any solution of a PDE solved by esys.finley the values on the line $x_0 = 0$ will be identical to the values on $x_0 = 10$. Correspondingly, periodic1 = True and periodic2 = True sets periodic boundary conditions in the $x_1$-direction and $x_2$-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 than tolerance times the diameter of the domain are merged. The corresponding face elements are removed from the mesh.

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 node coordinates differ by less than tolerance times the diameter of the domain are combined to form a contact element. The corresponding face elements are removed from the mesh.
CHAPTER
EIGHT

The esys.weipa Module and Data Visualization

The weipa C++ library and accompanying python module allow exporting esys.escript Data objects and their domain in a format suitable for visualization. Besides creating output files, weipa can also interface with the VisIt visualization software. This allows accessing the latest simulation data while the simulation is still running without the need to save any files.

8.1 The EscriptDataset class

class EscriptDataset()
holds an escript dataset including a domain and data variables for a single time step and offers methods to export the data in various formats. It is not recommended to create a dataset object directly in python using the constructor (it is not actually exposed) but use the createDataset function from esys.weipa instead, see Section 8.2.

The following methods are available:

setDomain(domain)
sets the Domain for this dataset. Note that the domain can only be set once and all Data objects added to this dataset must be defined on the same domain.

addData(data, name [, units=""])
adds the Data object data to this dataset which will be exported by the given name. Some export formats support data units which can be set through the units parameter, e.g. "km/h". Before calling this method a domain must be set with setDomain and all Data objects added must be defined on the same domain. There is no restriction, however, on the FunctionSpace used.

setCycleAndTime(cycle, time)
sets the cycle and time values for this dataset. The cycle is an integer value which usually corresponds with the loop counter of the simulation script. That is, every time a new data file is created this counter is incremented. The value of time on the other hand is a floating point number that encodes some form of simulation time. Both, cycle and time may be read by analysis tools and shown alongside other metadata to the user.

setMeshLabels(x, y [, z="" ])
sets the labels of the X, Y, and Z axis. By default, visualization tools display default strings such as "X-Axis" or "X" along the axes. Some export formats allow overriding these with more specific strings such as "Width", "Horizontal Distance", etc.
setMeshUnits(x, y [, z=""])
sets the units to be displayed along the X, Y, and Z axis in visualization tools (if supported). Not all export formats will use these values.

setMetadataSchemaString([ schema="" [, metadata="" ] ])
adds custom metadata and/or XML schema strings to VTK files. The content of schema is added to the top-level VTKFile element so care must be taken to keep the resulting file valid. As an example, schema may contain the string xmlns:gml="http://www.opengis.net/gml". The content of metadata will be written enclosed in <MetaData> tags. Thus, a valid example would be <dataSource>something</dataSource>. Note that these values are ignored by other exporters.

saveSilo(filename)
saves the dataset in the SILO file format to a file named filename. The file extension .silo will be automatically added if not present.

saveVTK(filename)
saves the dataset in the VTK file format to a file named filename. The file extension .vtu will be automatically added if not present. Certain combinations of function spaces cannot be written to a single VTK file due to format restrictions. In these cases this method will save separate files where each file contains compatible data. The function space name is appended to the filename to distinguish them.

8.2 Functions

createDataset(domain, **data)
creates an EscriptDataset object, sets its domain, populates it with the given Data objects and returns it. Note that it is not possible to set units for the data variables added with this function. If this is required, it is recommended to call this function with a domain only and use the addData method subsequently.

saveVTK(filename [, domain=None [, metadata="" [, metadata_schema=None ] ] ]], **data)
convenience function that creates a dataset with the given domain and Data objects and saves it to a file in the VTK file format. If domain is None the domain will be determined by the Data objects. See the setDomain, addData, saveVTK, and setMetadataSchemaString methods of the EscriptDataset class for details. Unlike the class method, the metadata_schema parameter should be a dictionary that maps namespace name to URI, e.g. {"gml" : "http://www.opengis.net/gml"}.

saveSilo(filename [, domain=None ], **data)
convenience function that creates a dataset with the given domain and Data objects and saves it to a file in the SILO file format. If domain is None the domain will be determined by the Data objects. See the setDomain, addData, and saveSilo methods of the EscriptDataset class for details.

visitInitialize(simFile [, comment=""])
initializes the VisIt simulation interface which is responsible for the communication with a VisIt client. This function will create a file by the name given via simFile (extension .sim2) which can be loaded by a compatible VisIt client in order to connect to the simulation. The optional comment string is forwarded to the client. Note that this function only succeeds if escript was compiled with support for VisIt and the appropriate libraries are found in the runtime environment. Clients wanting to connect can only do so if the version number matches the version number used to compile esys.weipa. Calling this function does not make any data available yet, see the visitPublishData function.

visitPublishData(dataset)
publishes an EscriptDataset object through the VisIt simulation interface, checks for client requests and handles any outstanding ones. Before publishing any data, the visitInitialize function must
be called to set up the interface. Since this function not only publishes new data but polls for incoming connections and handles requests, it should be called as often as practical (even with the same dataset) to avoid timeout errors from clients. On the other hand it should be noted that the same process(es) deal with visualization requests that run your simulation. So a request for an expensive task by a VisIt client will pause the simulation code while it is being processed.

8.3 Visualizing escript Data

This section gives a very brief overview on how data exported through esys.weipa can be visualized. While there are many visualization packages available that are compatible with VTK and SILO files produced by escript, this discussion will refer to VisIt [35], an actively maintained open source package optimally suited to visualize and analyze large datasets both interactively and through python scripts. You can find a number of manuals, a wiki page and links to mailing lists on the VisIt website. It is assumed that you have a working VisIt installation that can be started by entering visit on the command line.

The examples that follow will use the output produced by the Elastic Deformation example from Section 1.4 (heatedblock.py in the example directory) which produces the file deform.vtu. This VTK file contains a 3D scalar variable called stress and a vector variable called disp, among others.

8.3.1 Using the VisIt GUI

Start the VisIt graphical user interface and open the file deform.vtu via the 'File' menu. Alternatively, you can directly open the file on startup by issuing

visit -o deform.vtu

You should see the VisIt GUI on the left hand side and an empty visualization window on the right. Click on 'Add' under Plots in the GUI to bring up a menu of plot types, then click on 'Pseudocolor' and select 'stress'. This will add a plot to the list which maps values of the 'stress' variable to colors. Note, that the plot will not be generated until you click on the 'Draw' button in the GUI. You should now see a coloured box in the visualization window which you can rotate around and inspect from different angles using your mouse. The example uses a coarse mesh of 10 by 10 by 10 elements which are clearly visible in this plot.

We can improve the visual effect by enabling interpolation between the elements. To do so, bring up the plot attributes by double-clicking the 'Pseudocolor - stress' plot entry in the GUI. Next, select 'Nodal' under 'Centering', click on 'Apply' and dismiss the dialog. Notice how the colours now smoothly blend into each other and the element boundaries are no longer visible.

Now we will add arrows to visualize the displacement vectors. Click on 'Add' and under 'Vector' select 'disp'. Once again click on 'Draw' to execute the new plot. By default only few vectors are shown but since the mesh is very coarse we can tell VisIt to draw all available vectors. Bring up the Vector plot attributes (double-click on the plot as before) and under 'Vector amount' select 'Stride', leaving the parameter as 1. Click on 'Apply' and dismiss the dialog.

As a final step we would like to see inside the plot. One possibility to do so is slicing. However, we want to keep all vectors while slicing only the Pseudocolor plot. In VisIt slicing is one of the Operators that may be added to all plots. To change this behaviour, uncheck the 'Apply operators to all plots' box which is located underneath the plot list in the GUI. Then select the Pseudocolor plot, bring up the Operators menu by clicking on 'Operators' and select 'ThreeSlice' from the 'Slicing' submenu. Again, click on 'Draw' to update the plots and notice how the box has now been sliced. We can move the slices to more suitable positions by editing the operator attributes. Click on the little triangle to the left of the Pseudocolor plot to reveal the list of elements that have been applied to it. Next, double-click the 'ThreeSlice' element to bring up the attribute window. Change the values to \( X = 0.3 \) and \( Y = 0.3 \), leaving \( Z = 0 \). Apply the changes and dismiss the dialog to see the result.

You can now create an image of the plots as shown in the window. First, adjust the save options to your needs in the 'Set Save options' dialog which is accessible from the 'File' menu. Then select 'Save Window' and you should find an image file with the name and location as entered in the options dialog.
8.3.2 Using the VisIt CLI

We will now perform exactly the same steps as in the last section but using the python interface of VisIt instead of the GUI. Start up the CLI by issuing

```bash
visit -cli
```

You should now see an empty visualization window but unlike in the previous section there will be no graphical user interface but a python command line instead. Enter the following commands, one by one, noticing the changes in the visualization window after every block of commands:

```python
OpenDatabase('deform.vtu')
AddPlot('Pseudocolor','stress')
DrawPlots()

p=PseudocolorAttributes()
p.centering=p.Nodal
SetPlotOptions(p)

AddOperator('ThreeSlice')
DrawPlots()

t=ThreeSliceAttributes()
t.x=0.3
t.y=0.3
SetOperatorOptions(t)

AddPlot('Vector', 'disp')
DrawPlots()

v=VectorAttributes()
v.useStride=1
SetPlotOptions(v)

s=SaveWindowAttributes()
#change settings as required
SaveWindow()
exit()
```

All but the last call to `DrawPlots()` is not required and was only put there for demonstrating the effects of the commands. You can save these commands to a file, e.g. `deformVis.py` and let VisIt process them non-interactively like so:

```bash
visit -cli -nowin -s deformVis.py
```

The `-nowin` option prevents the visualization window from being shown which is not required since the purpose of the script is to save an image file.

Obviously, we have barely touched on the powerful features of VisIt and this section was only meant to give you a minimal introduction. The VisIt website has a reference manual for the python interface that explains how to perform other operations programmatically, such as changing the view.
Einstein Notation

Compact notation is used in equations such 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 a 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, the expression
\[
y = a_0b_0 + a_1b_1 + \ldots + a_nb_n
\]
can be represented as
\[
y = \sum_{i=0}^{n} a_ib_i
\]
then in Einstein notation:
\[
y = a_ib_i
\]
Another example:
\[
\nabla p = \frac{\partial p}{\partial x_0}i + \frac{\partial p}{\partial x_1}j + \frac{\partial p}{\partial x_2}k
\]
can be expressed in Einstein notation as
\[
\nabla p = p,i
\]
where the comma ',' in the subscript indicates the partial derivative.

For a tensor:
\[
\sigma_{ij} = \begin{bmatrix} \sigma_{00} & \sigma_{01} & \sigma_{02} \\ \sigma_{10} & \sigma_{11} & \sigma_{12} \\ \sigma_{20} & \sigma_{21} & \sigma_{22} \end{bmatrix}
\]

The \(\delta_{ij}\) is the Kronecker \(\delta\)-symbol, which is a matrix with ones in its diagonal entries \((i = j)\) and zeros in the remaining entries \((i \neq j)\).
\[
\delta_{ij} = \begin{cases} 1, & \text{if } i = j \\ 0, & \text{if } i \neq j \end{cases}
\]
Changes from previous releases

3.1 to 3.2

- The deprecated name for the launcher has been removed. To run scripts use `run-escript` not `escript`.
- `esys.escript` is no longer automatically imported by importing `esys.finley`. You will need to import `escript` explicitly. (All of our example scripts do this anyway.)
- An experimental version of the new Dudley domain is now available.
- Various bug fixes and optimisations.
- New algorithms for gmsh support.
- Improvements to the AMG solver. AMG is the recommended solver for symmetric problems.
- Fixed compilation issues using netcdf.
- Redesigned configuration files to make it easier to compile from source without finding the locations of all your libraries.
- Faster rendering of documentation.
- Documentation is now hyperlinked.
- New data export module `esys.weipa`. The `saveVTK` functionality has been moved into this module, and while calling `saveVTK` from the `esys.escript` module still works it is discouraged and will be removed in a future release.
- New `esys.escript.DataManager` class for convenient checkpointing and exporting of escript data.
- `VisIt` simulation interface for online data access and visualization.
- Simpler interpolation and support for interpolation from 3D vectors.
- HRZ lumping has been added and some clarification on how to use it.
- Data objects populated with “random” values can be created.

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 the future.
- Lazy evaluation features have been improved and documented (see Section 2.5).
• The escript documentation now 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 (see Section 3.2.9).

• Data objects can be populated by interpolating from 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 arrays/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. 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 now accept numpy objects instead.

• The way to define solver options for LinearPDE objects has changed. There is now a SolverOptions object attached to the LinearPDE object which handles the options of solvers used to solve the PDE. The following changes apply:

  – The setTolerance and setAbsoluteTolerance methods have been removed. Instead use setTolerance and setAbsoluteTolerance on the SolverOptions object. For example: SolverOptions().setTolerance(...) 

  – The setSolverPackage and setSolverMethod methods have been removed. Instead use the methods setPackage, setSolverMethod and setPreconditioner. For example: getSolverOptions().setPackage(...).

  – 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` now takes 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 be used if you compile `esys.escript` from source. It will not be available in binary releases and its use is discouraged. *The documentation for `esys.pyvisi` is not available in this release.*
### Escript researchers and developers by release

<table>
<thead>
<tr>
<th>Name</th>
<th>Releases</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cihan Altinay</td>
<td>2.0</td>
</tr>
<tr>
<td>Artak Amirkyan</td>
<td>2.0</td>
</tr>
<tr>
<td>Joel Fenwick</td>
<td>2.0</td>
</tr>
<tr>
<td>Lin Gao</td>
<td>2.0</td>
</tr>
<tr>
<td>Lutz Gross</td>
<td>1.0</td>
</tr>
<tr>
<td>Peter Hornby</td>
<td>1.0</td>
</tr>
<tr>
<td>Ken Steube</td>
<td>1.0</td>
</tr>
<tr>
<td>Elspeth Thorne</td>
<td>1.0</td>
</tr>
<tr>
<td>Matt Davies</td>
<td>1.0</td>
</tr>
<tr>
<td>Brett Tully</td>
<td>2.0</td>
</tr>
<tr>
<td>John Smilie</td>
<td>1.0</td>
</tr>
<tr>
<td>Cochrane</td>
<td>1.0</td>
</tr>
<tr>
<td>Rob Woodcock</td>
<td>1.0</td>
</tr>
<tr>
<td>Derek Hawcroft</td>
<td>1.0</td>
</tr>
<tr>
<td>Jon Gui</td>
<td>2.0</td>
</tr>
<tr>
<td></td>
<td>Current</td>
</tr>
</tbody>
</table>
If you use escript in your research we would appreciate a citation (of course we do not require this). Possible references include:

```plaintext
@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}
}
```

```plaintext
@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 = {August},
    doi = {doi:10.1016/j.pepi.2007.04.004},
}
```

```plaintext
@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},
    publisher = {Springer}
}
```


[34] http://www.cise.ufl.edu/research/sparse/umfpack/.


