# esys User's Guide: Solving Partial Differential Equations with Escript and Finley

*Release 3.0*
*(r2601)*

Lutz Gross *et al.* (Editor)

Earth Systems Science Computational Centre (ESSCC)
School of Earth Sciences
The University of Queensland
Brisbane, Australia
Email: esys@esscc.uq.edu.au

Copyright (c) 2003-2009 by University of Queensland
Earth Systems Science Computational Center (ESSCC)
School of Earth Sciences
http://www.uq.edu.au/esscc
Primary Business: Queensland, Australia
Licensed under the Open Software License version 3.0
http://www.opensource.org/licenses/osl-3.0.php

**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. The `esys.pyvisi` module from previous releases has been deprecated. For more info on this and other changes from previous releases see Appendix A.2.

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

# CONTENTS

CHAPTER

**ONE**

# Tutorial: Solving PDEs

## 1.1 Installation

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

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

## 1.2 The First Steps

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

The PDE we wish to solve is the Poisson equation

$$-\Delta u = f \tag{1.1}$$

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

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

The domain is shown in Figure 1.1.

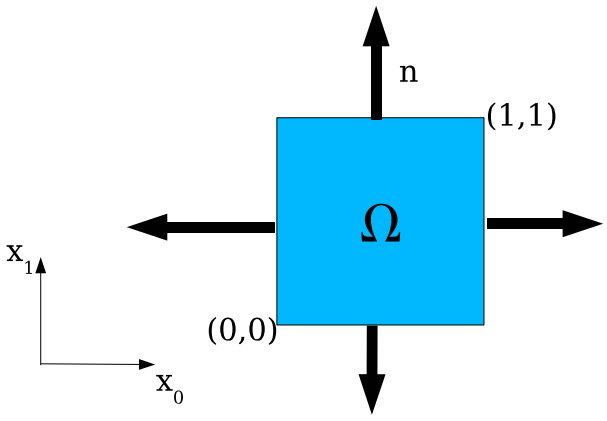



FIGURE 1.1: Domain $\Omega = [0,1]^2$ with outer normal field $n$.

$\Delta$ denotes the Laplace operator, which is defined by

$$\Delta u = (u_{,0})_{,0} + (u_{,1})_{,1} \tag{1.3}$$

where, for any function $u$ and any direction $i$, $u_{,i}$ denotes the partial derivative of $u$ with respect to $i$. [1] 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

$$\Delta u = u_{,00} + u_{,11} = \sum_{i=0}^{2} u_{,ii} \tag{1.4}$$

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

$$x_i y_i = \sum_{i=0}^{2} x_i y_i \tag{1.5}$$

$$x_i u_{,i} = \sum_{i=0}^{2} x_i u_{,i} \tag{1.6}$$

$$u_{,ii} = \sum_{i=0}^{2} u_{,ii} \tag{1.7}$$

$$x_{ij} u_{i,j} = \sum_{j=0}^{2} \sum_{i=0}^{2} x_{ij} u_{i,j} \tag{1.8}$$

$$\tag{1.9}$$

With the summation convention we can write the Poisson equation as

$$-u_{,ii} = 1 \tag{1.10}$$

where $f = 1$ in this example.

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

$$n_i u_{,i} = 0 \ . \tag{1.11}$$

$n = (n_i)$ denotes the outer normal field of the domain, see Figure 1.1. Remember that we are applying the Einstein summation convention , i.e $n_i u_{,i} = n_0 u_{,0} + n_1 u_{,1}$. [2] 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:

$$\Gamma^N = \{(x_0; x_1) \in \Omega | x_0 = 1 \text{ or } x_1 = 1\} \tag{1.12}$$

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

$$\Gamma^D = \{(x_0; x_1) \in \Omega | x_0 = 0 \text{ or } x_1 = 0\} \tag{1.13}$$

the solution shall be identically zero:

$$u = 0 \ . \tag{1.14}$$

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

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

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

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

and Equation (1.1) as

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

[2] Some readers may familiar with the notation $\frac{\partial u}{\partial n} = n_i u_{,i}$ for the normal derivative.

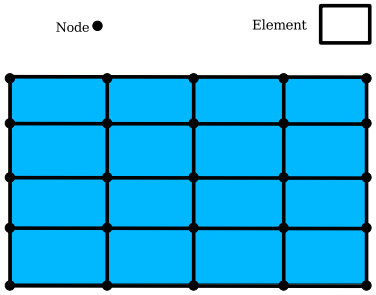



FIGURE 1.2: Mesh of 4 elements on a rectangular domain. Here each element is a quadrilateral and described by four nodes, namely the corner points. The solution is interpolated by a bi-linear polynomial.

is subdivided into smaller sub-domains called elements . On each element the solution is represented by a polynomial of a certain degree through its values at the nodes located in the element. The nodes and its connection through elements is called a mesh. Figure 1.2 shows an example of a FEM mesh with four elements in the $x0$ and four elements in the $x1$ direction over the unit square. For more details we refer the reader to the literature, for instance Reference [28, 5].

The `esys.escript` solver we want to use to solve this problem is embedded into the python interpreter language. So you can solve the problem interactively but you will learn quickly that is more efficient to use scripts which you can edit with your favorite editor. To enter the escript environment you use **escript** command[3]:

```
escript
```

which will pass you on to the python prompt

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

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

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

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

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

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

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

[3]**escript** is not available under Windows yet. If you run under windows you can just use the **python** command and the **OMP_NUM_THREADS** environment variable to control the number of threads.

$(l0, l1) = (1, 1)$. The arguments *n0* and *n1* define the number of elements in $x_0$ and $x_1$-direction respectively. For more details on `Rectangle` and other `Domain` generators within the `esys.finley` module, see Chapter 7.

The following statements define the `Poisson` class object *mypde* with domain *mydomain* and the right hand side $f$ of the PDE to constant 1:

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

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

```
x=mydomain.getX()
```

The method `getX` of the `Domain` *mydomain* gives access to locations in the domain defined by *mydomain*. The object *x* is actually a `Data` object which will be discussed in Chapter 3 in more detail. What we need to know here is that

*x* has rank (number of dimensions) and a shape (list of dimensions) which can be viewed by calling the `getRank` and `getShape` methods:

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

This will print something like

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

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

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

will print

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

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

```
x0=x[0]
```

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

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

will print

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

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

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

`whereZero(x[0])` creates function which equals 1 where `x[0]` is (almost) equal to zero and 0 elsewhere. Similarly, `whereZero(x[1])` creates function which equals 1 where `x[1]` is equal to zero and 0 elsewhere. The sum of the results of `whereZero(x[0])` and `whereZero(x[1])` gives a function on the domain *mydomain* which is strictly positive where $x_0$ or $x_1$ is equal to zero. Note that *gammaD* has the same rank , shape and `FunctionSpace` like *x0* used to define it. So from

```
print gammaD.getRank(),gammaD.getShape(),gammaD.getFunctionSpace()
```

one gets

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

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

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

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

Now we can write the script to solve our Poisson problem

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

The question is what we do with the calculated solution *u*. Besides postprocessing, eg. calculating the gradient or the average value, which will be discussed later, plotting the solution is one one things you might want to do. `esys.escript` offers two ways to do this, both base 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, see [12]. However, there are limitations when using this tool, eg. in problem size and when solving 3D problems. Therefore `esys.escript` provides a second options based on *VTK* files which is especially designed for large scale and 3D problem and which can be read by a variety of software packages such as *mayavi* [2], *VisIt* [24].

### 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 [4]. We will make use of the `numpy` module.

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

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

*x_grid* is an array defining the x coordinates of the grids while *y_grid* defines the y coordinates of the grid. In this case we use $50$ points over the interval $[0, 1]$ in both directions.

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

[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.

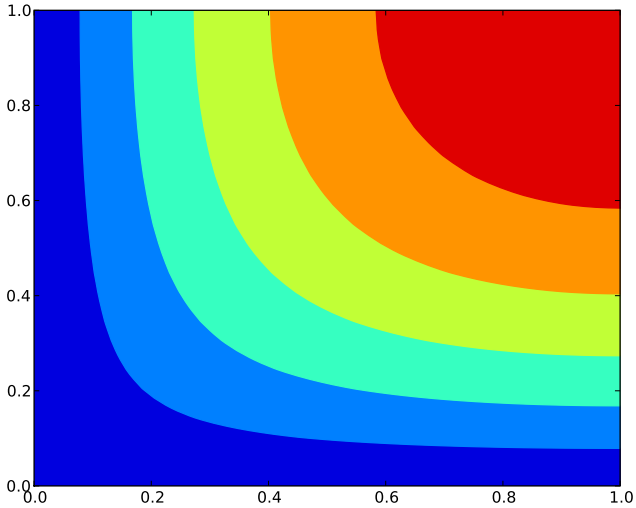



FIGURE 1.3: Visualization of the Poisson Equation Solution for $f = 1$ using `matplotlib`.

```
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 us 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 now the values at the points with the coordinates given by $x$ and $y$. These values are now interpolated to the grid defined by *x_grid* and *y_grid* by using

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

*z_grid* gives now the values of the PDE solution $u$ at the grid. The values can be plotted now using the `contourf`:

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

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

```
matplotlib.pyplot.contourf(x_grid, y_grid, z_grid, 5)
matplotlib.pyplot.show()
```

which gives an interactive browser window.

Now we can write the script to solve our Poisson problem

```
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])
```

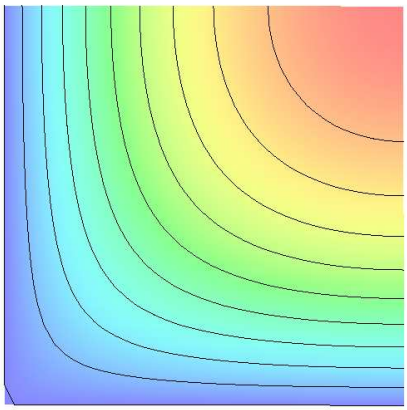


FIGURE 1.4: Visualization of the Poisson Equation Solution for $f = 1$

```
# define PDE and get its solution u
mypde = Poisson(domain=mydomain)
mypde.setValue(f=1,q=gammaD)
u = mypde.getSolution()
# interpolate u to a matplotlib grid:
x_grid = numpy.linspace(0.,1.,50)
y_grid = numpy.linspace(0.,1.,50)
x=mydomain.getX()[0].toListOfTuples()
y=mydomain.getX()[1].toListOfTuples()
z=interpolate(u,mydomain.getX().getFunctionSpace())
z_grid = matplotlib.mlab.griddata(x,y,z,xi=x_grid,yi=y_grid )
# interpolate u to a rectangular grid:
matplotlib.pyplot.contourf(x_grid, y_grid, z_grid, 5)
matplotlib.pyplot.savefig("u.png")
```

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

```
escript poisson_matplotlib.py
```

This will create the 'u.png', see Figure Figure 1.3. For details on the usage of the `matplotlib` module we refer to the documentation [12].

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

### 1.2.2 Visualization using *VTK*

As an alternative *escript* supports the usage of visualization tools which base on *VTK*, eg. mayavi [2], *VisIt* [24]. In this case the solution is written to a file in the *VTK* format. This file the can read by the tool of choice. Using *VTK* file is *MPI* safe.

To write the solution $u$ in Poisson problem to the file 'u.xml' one need to add the line

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

[5] The phrase 'safe under *MPI* ' means that a program will produce correct results when run on more than one processor under *MPI* .

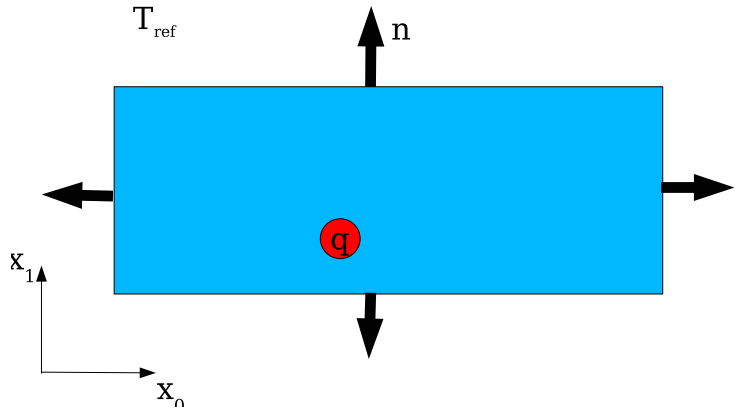



FIGURE 1.5: Temperature Diffusion Problem with Circular Heat Source

The solution $u$ is now available in the 'u.xml' tagged with the name "sol".

The Poisson problem script is now

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

The entire code is available as 'poisson_VTK.py' in the example directory

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

```
escript poisson\hackscore VTK.py
mayavi2 -d u.xml -m SurfaceMap
```

The result is shown in Figure Figure 1.4.

## 1.3 The Diffusion Problem

### 1.3.1 Outline

In this chapter we will discuss how to solve a time-dependent temperature diffusion PDE for a given block of material. Within the block there is a heat source which drives the temperature diffusion. On the surface, energy can radiate into the surrounding environment. Figure 1.5 shows the configuration.

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

### 1.3.2 Temperature Diffusion

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

$$\rho c_p T_{,t} - (\kappa T_{,i})_{,i} = q_H \tag{1.15}$$

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

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

for all $x$ in the domain and all time $t > 0$.

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

$$\kappa T_{,i} n_i = \eta(T_{ref} - T) \tag{1.17}$$

$\eta$ is a given material coefficient depending on the material of the block and the surrounding medium. $n_i$ is the $i$-th component of the outer normal field at the surface of the domain.

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

$$T(x,0) = T_{ref} \tag{1.18}$$

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

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

$$T^{(n)} \approx T^{(n-1)} + T^{(n)}_{,t}(t^{(n)} - t^{(n-1)}) = T^{(n-1)} + h \cdot T^{(n)}_{,t} \tag{1.19}$$

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

$$\frac{\rho c_p}{h} T^{(n)} - (\kappa T^{(n)}_{,i})_{,i} = q_H + \frac{\rho c_p}{h} T^{(n-1)} \tag{1.20}$$

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

$$\kappa T^{(n)}_{,i} n_i = \eta(T_{ref} - T^{(n)}) \tag{1.21}$$

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

### 1.3.3 Helmholtz Problem

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

$$\omega T^{(n)} - (\kappa T^{(n)}_{,i})_{,i} = f \tag{1.22}$$

and we set

$$\omega = \frac{\rho c_p}{h} \text{ and } f = q_H + \frac{\rho c_p}{h} T^{(n-1)} \ . \tag{1.23}$$

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

$$\kappa T^{(n)}_{,i} n_i = g - \eta T^{(n)} \text{ on } \Gamma \tag{1.24}$$

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

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

$$-(A_{jl}u_{,l})_{,j} + Du = Y \ . \tag{1.25}$$

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

$$n_j A_{jl} u_{,l} + du = y \ . \tag{1.26}$$

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

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

$$\begin{array}{lll} A_{ij} = \kappa \delta_{ij} & D = \omega & Y = f \\ d = \eta & y = g & \end{array} \tag{1.28}$$

$\delta_{ij}$ is the Kronecker symbol defined by $\delta_{ij} = 1$ for $i = j$ and 0 otherwise. Undefined coefficients are assumed to be not present.[6] In this diffusion example we do not need to define a characteristic function $q$ because the boundary conditions we consider in Equation (1.24) are just the natural boundary conditions which are already defined in the `LinearPDE` class (shown in Equation (1.26)).

The Helmholtz equation can be set up by following way [7] :

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

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

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

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

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

[7] Please, note that this is not a complete code. The complete code can be found in "helmholtz.py".

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

$$A_{jl} = A_{lj} \ . \tag{1.29}$$

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.

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

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

```
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 $q_H = 0$) and then calculate the right hand side terms $f$ and $g$ such that the test solution becomes the solution of the problem. If we assume $\kappa$ as being constant, an easy calculation shows that we have to choose $f = \omega \cdot x_0$. On the boundary we get $\kappa n_i u_{,i} = \kappa n_0$. So we have to set $g = \kappa n_0 + \eta x_0$. The following script 'helmholtz.py' which is available in the example directory implements this test problem using the `esys.finley` PDE solver:

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

To visualize the solution 'x0. xml' just use the command

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

and it is easy to see that the solution $T = x_0$ is calculated.

The script is similar to the script 'poisson.py' discussed in Chapter 1.2. `mydomain.getNormal()` returns the outer normal field on the surface of the domain. The function `Lsup` imported by the `from esys.escript import *` statement and returns the maximum absolute value of its argument. The error shown by the print statement should be in the order of $10^{-7}$. As piecewise bi-linear interpolation is used by `esys.finley` approximate the solution and our solution is a linear function of the spatial coordinates one might expect that the error would be zero or in the order of machine precision (typically $\approx 10^{-15}$). However most PDE packages use an

iterative solver which is terminated when a given tolerance has been reached. The default tolerance is $10^{-8}$. This value can be altered by using the `setTolerance` of the `LinearPDE` class.

### 1.3.4 The Transition Problem

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

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

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

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

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

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

$$\|y\| = \sqrt{y_i y_i} = \texttt{esys.escript.length}(y) \tag{1.30}$$

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

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

```
from esys.escript import *
from esys.escript.linearPDEs import LinearPDE
from esys.finley import Rectangle
#... set some parameters ...
xc=[0.02,0.002]
r=0.001
qc=50.e6
Tref=0.
rhocp=2.6e6
eta=75.
kappa=240.
tend=5.
# ... time, time step size and counter ...
```

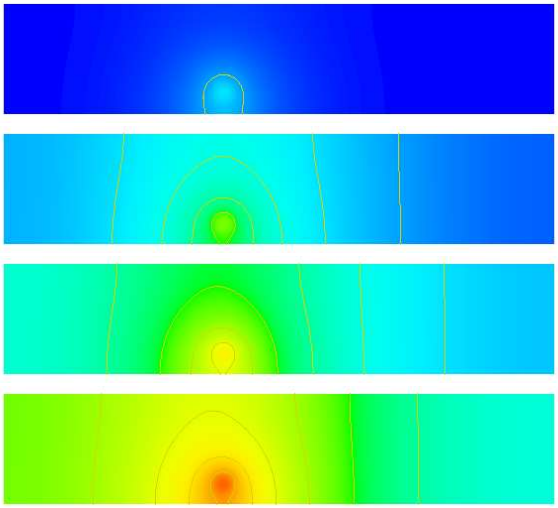


FIGURE 1.6: Results of the Temperature Diffusion Problem for Time Steps 1 16, 32 and 48.

```
t=0
h=0.1
i=0
#... generate domain ...
mydomain = Rectangle(l0=0.05,l1=0.01,n0=250, n1=50)
#... open PDE ...
mypde=LinearPDE(mydomain)
mypde.setSymmetryOn()
mypde.setValue(A=kappa*kronecker(mydomain),D=rhocp/h,d=eta,y=eta*Tref)
# ... set heat source: ....
x=mydomain.getX()
qH=qc*whereNegative(length(x-xc)-r)
# ... set initial temperature ....
T=Tref
# ... start iteration:
while t<tend:
      i+=1
      t+=h
      print "time step :",t
      mypde.setValue(Y=qH+rhocp/h*T)
      T=mypde.getSolution()
      saveVTK("T.%d.xml"%i,temp=T)
```

The script will create the files 'T.1.xml', 'T.2.xml', . . ., 'T.50.xml' in the directory where the script has been started. The files give the temperature distributions at time steps $1, 2, \ldots, 50$ in the *VTK* file format.

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

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

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

## 1.4 3-D Wave Propagation

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

$$\rho u_{i,tt} - \sigma_{ij,j} = 0 \tag{1.31}$$

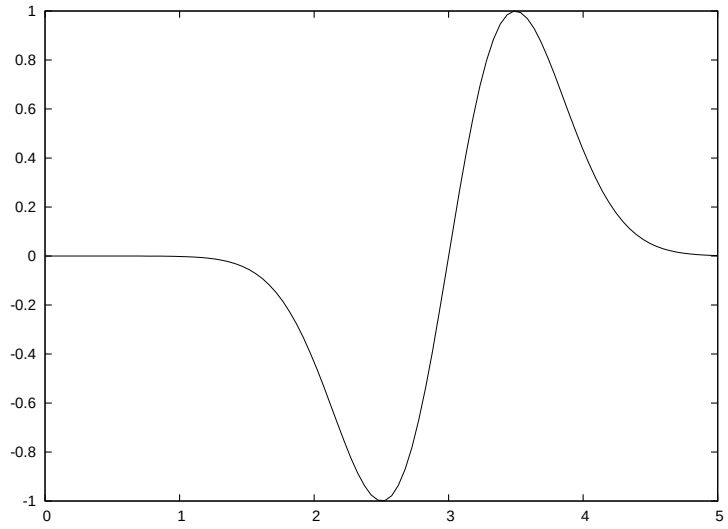



FIGURE 1.7: Input Displacement at Source Point ($\alpha = 0.7$, $t_0 = 3$, $U_0 = 1$).

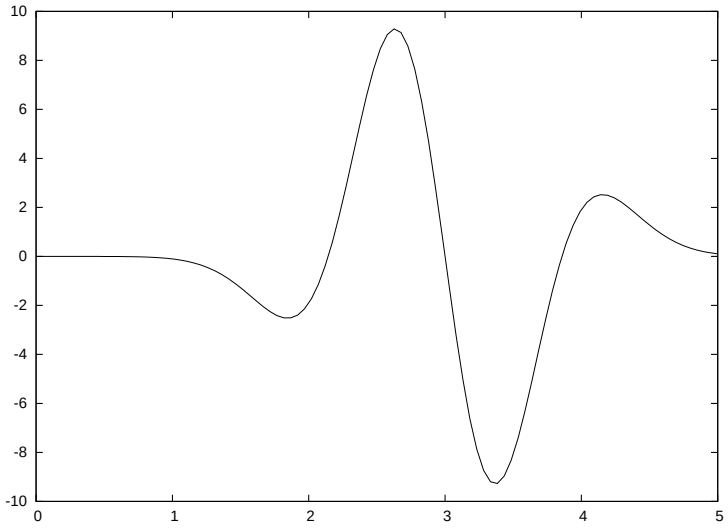



FIGURE 1.8: Input Acceleration at Source Point ($\alpha = 0.7$, $t_0 = 3$, $U_0 = 1$).

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

$$\sigma_{ij} \quad = \quad \lambda u_{k,k}\delta_{ij} + \mu(u_{i,j} + u_{j,i}) \tag{1.32}$$

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

$$\sigma_{ij}n_j = 0 \tag{1.33}$$

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

$$u_0(x_C, t) = U_0 \sqrt{2}\frac{(t - t_0)}{\alpha}e^{\frac{1}{2} - \frac{(t-t_0)^2}{\alpha^2}} \tag{1.34}$$

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

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

$$u^{(n)} = 2u^{(n-1)} - u^{(n-2)} + h^2 a^{(n)} \tag{1.35}$$

$$\tag{1.36}$$

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

$$u_{,tt} = G(u) \tag{1.37}$$

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

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

$$\rho a_i^{(n)} = \sigma_{ij,j}^{(n-1)} \tag{1.38}$$

and boundary conditions

$$\sigma_{ij}^{(n-1)} n_j = 0 \tag{1.39}$$

derived from Equation (1.33) where

$$\sigma_{ij}^{(n-1)} \quad = \quad \lambda u_{k,k}^{(n-1)} \delta_{ij} + \mu (u_{i,j}^{(n-1)} + u_{j,i}^{(n-1)}). \tag{1.40}$$

We also need to apply the constraint

$$a_0^{(n)}(x_C, t) = U_0 \frac{\sqrt{(2.)}}{\alpha^2} (4 \frac{(t-t_0)^3}{\alpha^3} - 6 \frac{t-t_0}{\alpha}) e^{\frac{1}{2} - \frac{(t-t_0)^2}{\alpha^2}} \tag{1.41}$$

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 now depends on the solution at the previous two time steps, $u^{(n-1)}$ and $u^{(n-2)}$.

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

$$D_{ij} a_j^{(n)} = -X_{ij,j} \ . \tag{1.42}$$

The natural boundary condition

$$n_j X_{ij} = 0 \tag{1.43}$$

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

$$D_{ij} = \rho \delta_{ij} \quad X_{ij} = -\sigma_{ij}^{(n-1)} \tag{1.44}$$

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

$$q_i(x) = \begin{cases} 1 & \|x - xc\| \leq R \\ 0 & \text{otherwise} \end{cases} \tag{1.45}$$

The following script defines a 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 *tend* are the time step size and the end time of the simulation. *lam*, *mu* and *rho* are material properties.

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

   x=domain.getX()
   # ... open new PDE ...
   mypde=LinearPDE(domain)
```

```
mypde.getSolverOptions().setSolverMethod(mypde.getSolverOptions().LUMPING)
kronecker=identity(mypde.getDim())
dunit=numpy.array([1.,0.,0.]) # defines direction of point source
mypde.setValue(D=kronecker*rho, q=whereNegative(length(x-xc)-src_radius)*dunit)
# ... set initial values ....
n=0
# for first two time steps
u=Vector(0.,Solution(domain))
u_last=Vector(0.,Solution(domain))
t=0
# define the location of the point source
L=Locator(domain,xc)
# find potential at point source
u_pc=L.getValue(u)
print "u at point charge=",u_pc
# open file to save displacement at point source
u_pc_data=FileWriter('./data/U_pc.out')
ts.append(t); u_pc0.append(u_pc[0]), u_pc1.append(u_pc[1]), u_pc2.append(u_pc[2])

while t<tend:
  t+=h
  # ... get current stress ....
  g=grad(u)
  stress=lam*trace(g)*kronecker+mu*(g+transpose(g))
  # ... get new acceleration ....
  amplitude=U0*(4*(t-t0)**3/alpha**3-6*(t-t0)/alpha)*sqrt(2.)/alpha**2 \
                                          *exp(1./2.-(t-t0)**2/alpha**2)
  mypde.setValue(X=-stress, r=dunit*amplitude)
  a=mypde.getSolution()
  # ... get new displacement ...
  u_new=2*u-u_last+h**2*a
  # ... shift displacements ....
  u_last=u
  u=u_new
  n+=1
  print n,"-th time step t ",t
  u_pc=L.getValue(u)
  print "u at point charge=",u_pc
  # save displacements at point source to file for t > 0
  ts.append(t); u_pc0.append(u_pc[0]), u_pc1.append(u_pc[1]), \
                                                u_pc2.append(u_pc[2])

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

return ts, u_pc0, u_pc1, u_pc2
```

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

The statement

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

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

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

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

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

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

```
L=Locator(domain,xc)
u=...
u_pc=L.getValue(u)
```

The return value `u_pc` is the value of `u` at the location `xc`[8]. The values are collected in the lists *u_pc0*, *u_pc1* and *u_pc2* together with the corresponding time marker in *ts*. The values are handed back to the caller. Later we will show to ways to access these data.

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

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

$$h = \frac{1}{5}\sqrt{\frac{\rho}{\lambda+2\mu}}\Delta x \tag{1.46}$$

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

The following script uses the `wavePropagation` function to solve the wave equation for a point source located at the bottom face of a block. The width of the block in each direction on the bottom face is 10km ($x_0$ and $x_1$ directions ie. `l0` and `l1`). The `ne` gives the number of elements in $x_0$ and $x_1$ directions. The depth of the block is aligned with the $x_2$-direction. The depth (`l2`) of the block in the $x_2$-direction is chosen so that there are 10 elements and the magnitude of the of the depth is chosen such that the elements become cubic. We chose 10 for the number of elements in $x_2$-direction so that the computation would be faster. `Brick`$(n_0, n_1, n_2, l_0, l_1, l_2)$ is an `esys.finley` function which creates a rectangular mesh with $n_0 \times n_1 \times n_2$ elements over the brick $[0, l_0] \times [0, l_1] \times [0, l_2]$.

```
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
# 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,l0=width,l1=width,l2=10.*width/32.)
h=(1./5.)*inf(sqrt(rho/(lam+2*mu))*inf(domain.getSize())
```

[8] In fact the finite element node which is closest to the given position. The usage of `Locator` is MPI save.

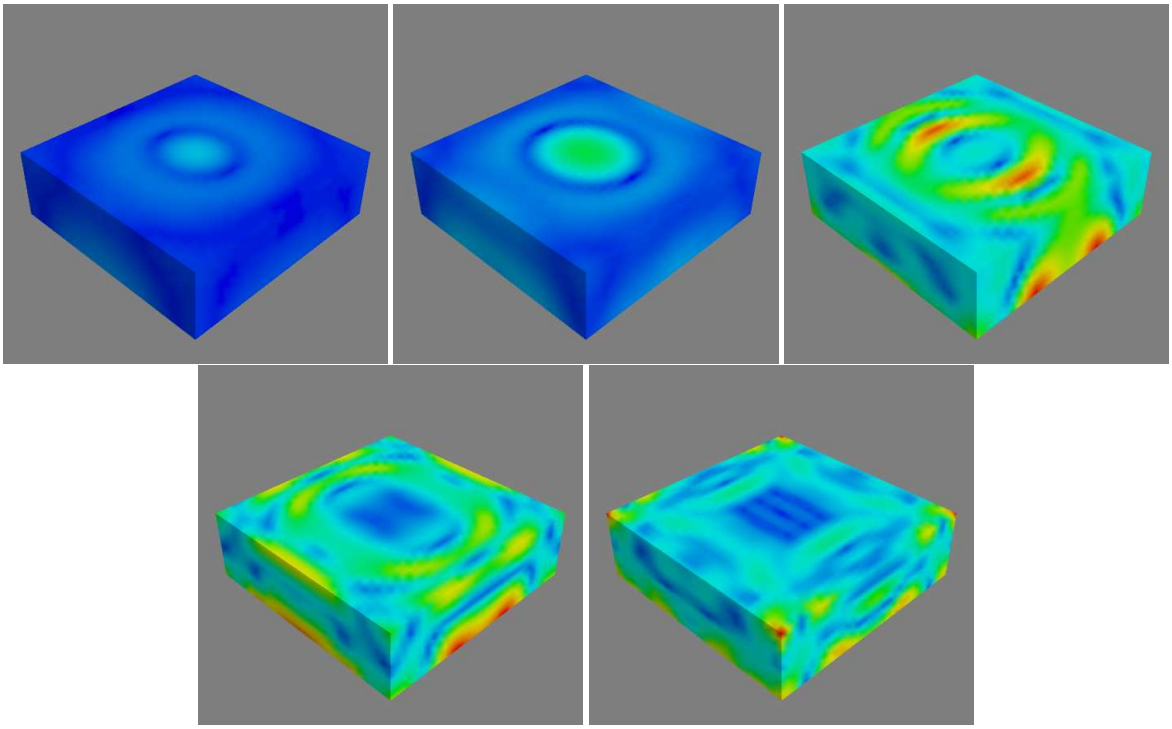


FIGURE 1.9: Selected time steps ($n = 11, 22, 32, 36$) of a wave propagation over a 10km $\times$ 10km $\times$ 3.125km block from a point source initially at $(5\text{km}, 5\text{km}, 0)$ with time step size $h = 0.02083$. Color represents the displacement. Here the view is oriented onto the bottom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

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

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

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

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

You can rotate this figure by clicking on it with the mouse and moving it around. Again use `Configure Data` to move backwards and forward in time, and also to choose the results (acceleration, displacement or $u_x$) 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* [26], excel or OpenOffice to read the data for further analysis. The following code shows one possible form to write the data to the file './data/U_pc.out':

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

The `U_pc.out` stores 4 columns of data: $t, u_x, u_y, u_z$ respectively, where $u_x, u_y, u_z$ are the $x_0, x_1, x_2$ components of the displacement vector $u$ at the point source. These can be plotted easily using any plotting package. In *gnuplot* [26]the command:

```
plot 'U_pc.out' u 1:2 title 'U_x' w l lw 2, 'U_pc.out' u 1:3 title 'U_y' w l lw 2,
'U_pc.out' u 1:4 title 'U_z' w l lw 2
```

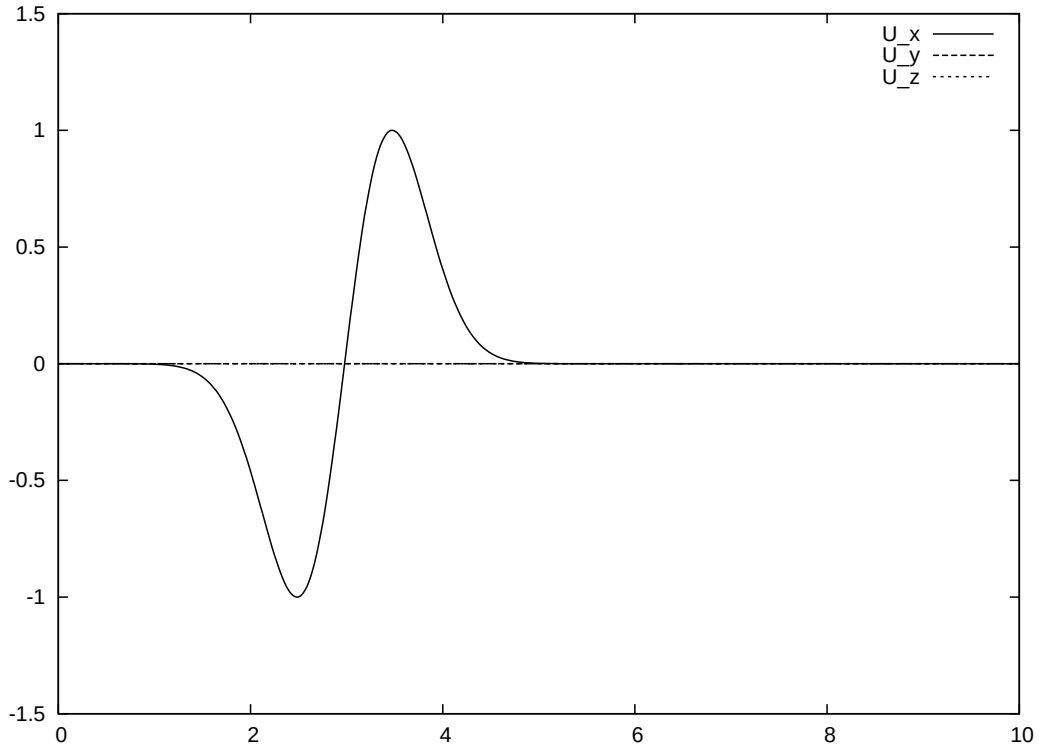



FIGURE 1.10: Amplitude at Point source from the Simulation.

will reproduce Figure 1.10 (As expected this is identical to the input signal shown in Figure 1.7)2. It is pointed out that we are not using the standard *python* `open` to write to the file `U_pc.out` as it is not safe when running `esys.escript` under MPI, see chapter 2 for more details.

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

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

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

Both options for processing the point source data are include in the example file 'wave.py'. There other options available to process these data in particular through the *SciPy*[7] package , eg Fourier transformations. It is beyond the scope of this users guide to document the usage of *SciPy*[7] for time series analysis but is highly recommended that users use *SciPy*[7] to any further data processing.

## 1.5 Elastic Deformation

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

$$-\sigma_{ij,j} = 0 \tag{1.47}$$

where the stress $\sigma$ is given by

$$\sigma_{ij} = \lambda u_{k,k}\delta_{ij} + \mu(u_{i,j} + u_{j,i}) - (\lambda + \frac{2}{3}\mu)\ \alpha\ (T - T_{ref})\delta_{ij}\ . \tag{1.48}$$

In this formula $\lambda$ and $\mu$ are the Lame coefficients, $\alpha$ is the temperature expansion coefficient, $T$ is the temperature distribution and $Tref$ a reference temperature. Note that Equation (1.47) is similar to Equation (1.31) introduced in section Section 1.4 but the inertia term $\rho u_{i,tt}$ has been dropped as we assume a static scenario here. Moreover, in comparison to the Equation (1.32) definition of stress $\sigma$ in Equation (1.48) an extra term is introduced to bring in stress due to volume changes trough temperature dependent expansion.

Our domain is the unit cube

$$\Omega = \{(x_i)|0 \leq x_i \leq 1\} \tag{1.49}$$

On the boundary the normal stress component is set to zero

$$\sigma_{ij}n_j = 0 \tag{1.50}$$

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

$$u_i(x) = 0 \quad \text{where} \quad x_i = 0 \tag{1.51}$$

For the temperature distribution we use

$$T(x) = T_0 e^{-\beta\|x-x^c\|}; \tag{1.52}$$

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

When we insert Equation (1.48) we get a second order system of linear PDEs for the displacements $u$ which is called the Lame equation. We want to solve this using the `LinearPDE` class to this. For a system of PDEs and a solution with several components the `LinearPDE` class takes PDEs of the form

$$-(A_{ijkl}u_{k,l})_{,j} = -X_{ij,j}\ . \tag{1.53}$$

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

$$n_j A_{ijkl}u_{k,l} = n_j X_{ij}\ . \tag{1.54}$$

Constraints take the form

$$u_i = r_i \text{ where } q_i > 0 \tag{1.55}$$

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

$$A_{ijkl} = \lambda\delta_{ij}\delta_{kl} + \mu(\delta_{ik}\delta_{jl} + \delta_{il}\delta_{jk}) \tag{1.56}$$

$$X_{ij} = (\lambda + \frac{2}{3}\mu)\ \alpha\ (T - T_{ref})\delta_{ij} \tag{1.57}$$

$$\tag{1.58}$$

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

$$q_i(x) = \begin{cases} 1 & x_i = 0 \\ 0 & \text{otherwise} \end{cases} \tag{1.59}$$

Under the assumption that $\lambda$, $\mu$, $\beta$ and $T_{ref}$ are constant we may use $Y_i = (\lambda + \frac{2}{3}\mu)\ \alpha\ T_i$. However, this choice would lead to a different natural boundary condition which does not set the normal stress component as defined in Equation (1.48) to zero.

Analogously to concept of symmetry for a single PDE, we call the PDE defined by Equation (1.53) symmetric if

$$A_{ijkl} = A_{klij} \tag{1.60}$$
$$\tag{1.61}$$

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

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

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

The following script, which is available in 'heatedbox.py' in the example directory, solves the Lame equation and writes the displacements and the von–Mises stress into a file 'deform.xml' in the *VTK* file format:

```
from esys.escript import *
from esys.escript.linearPDEs import LinearPDE
from esys.finley import Brick
#... set some parameters ...
lam=1.
mu=0.1
alpha=1.e-6
xc=[0.3,0.3,1.]
beta=8.
T_ref=0.
T_0=1.
#... generate domain ...
mydomain = Brick(l0=1.,l1=1., l2=1.,n0=10, n1=10, n2=10)
x=mydomain.getX()
#... set temperature ...
T=T_0*exp(-beta*length(x-xc))
#... open symmetric PDE ...
mypde=LinearPDE(mydomain)
mypde.setSymmetryOn()
#... set coefficients ...
C=Tensor4(0.,Function(mydomain))
for i in range(mydomain.getDim()):
  for j in range(mydomain.getDim()):
     C[i,i,j,j]+=lam
     C[j,i,j,i]+=mu
     C[j,i,i,j]+=mu
msk=whereZero(x[0])*[1.,0.,0.] \
   +whereZero(x[1])*[0.,1.,0.] \
   +whereZero(x[2])*[0.,0.,1.]
sigma0=(lam+2./3.*mu)*alpha*(T-T_ref)*kronecker(mydomain)
mypde.setValue(A=C,X=sigma0,q=msk)
#... solve pde ...
u=mypde.getSolution()
#... calculate von-Misses stress
g=grad(u)
sigma=mu*(g+transpose(g))+lam*trace(g)*kronecker(mydomain)-sigma0
sigma_mises=sqrt(((sigma[0,0]-sigma[1,1])**2+(sigma[1,1]-sigma[2,2])**2+ \
                  (sigma[2,2]-sigma[0,0])**2)/2. \
                 +3*(sigma[0,1]**2 + sigma[1,2]**2 + sigma[2,0]**2))
#... output ...
saveVTK("deform.xml",disp=u,stress=sigma_mises)
```

Finally the the results can be visualize by calling

```
mayavi -d deform.xml -f CellToPointData -m VelocityVector -m SurfaceMap &
```

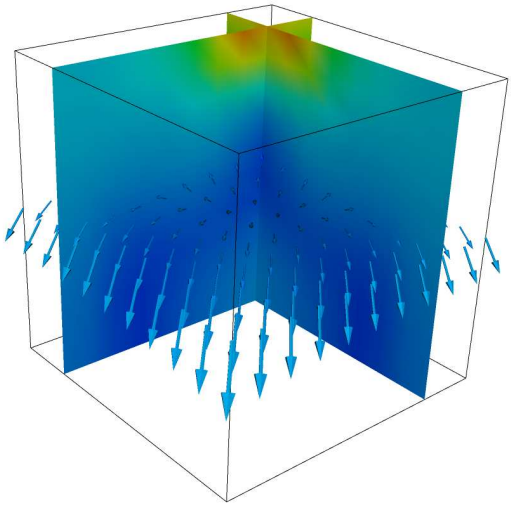


FIGURE 1.11: von–Mises Stress and Displacement Vectors.

Note that the filter CellToPointData is applied to create smooth representation of the von–Mises stress. Figure 1.11 shows the results where the vertical planes showing the von–Mises stress and the horizontal plane shows the vector representing displacements.

## 1.6 Stokes Flow

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

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

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 $f = -g\rho\delta_{i3}$. The Stokes equations is a saddle point problem, and can be solved using a Uzawa scheme. A class called StokesProblemCartesian in Escript can be used to solve for velocity and pressure; more detail on the class can be view in Chapter 6. In order to keep numerical stability, the time-step size needs to be kept below a certain value, to satisfy the Courant condition. The Courant number is defined as:

$$C = \frac{v\delta t}{h}. \tag{1.67}$$

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. The gravitational force is calculated base on the fluid density and the acceleration due to gravity. The boundary conditions are set for a slip condition at the base of the mesh; fluid movement in the x-direction is free, but fixed in the y-direction. An instance of the StokesProblemCartesian is defined for the given computational mesh, and the solver tolerance set. Inside the while loop, the boundary conditions, viscosity and body force are initialized. The Stokes equation is then solved for velocity and pressure. The time-step size is calculated base on the Courant condition, to ensure stable solutions. The nodes in the mesh are then displaced based on the current velocity and time-step size, to move the body of fluid. The output for the simulation of velocity and pressure is then save to file for visualization.

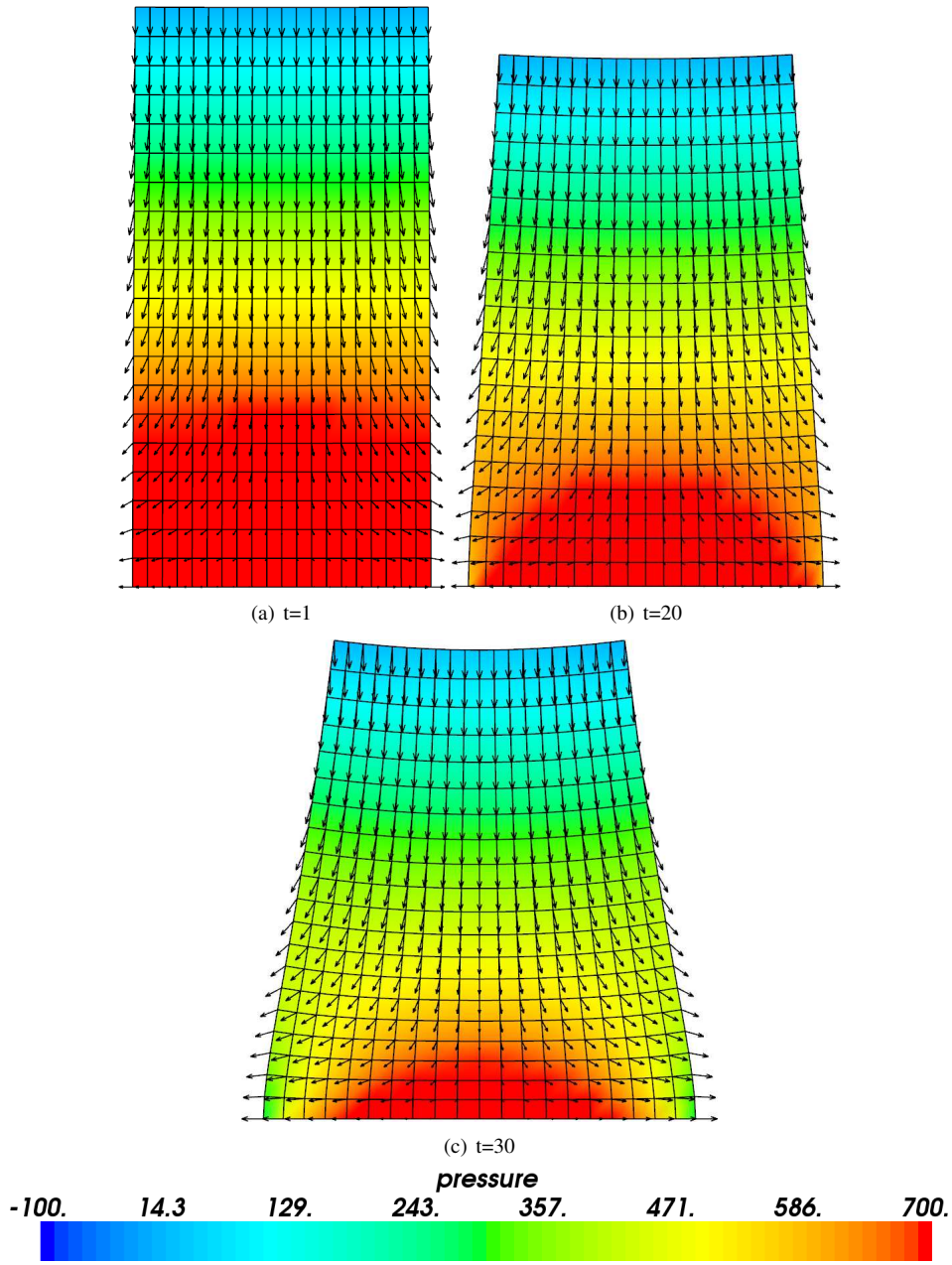
```
from esys.escript import *
import esys.finley
from esys.escript.linearPDEs import LinearPDE
from esys.escript.models import StokesProblemCartesian

#physical constants
eta=1.0
rho=100.0
g=10.0

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

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

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

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

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

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

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

while t <= t_end:

  print " ----- Time step = %s -----"%( 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"

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

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

time += dt

vel_mag = length(velocity)

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

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

```
mayavi -d vel.00.vtu -m SurfaceMap
```

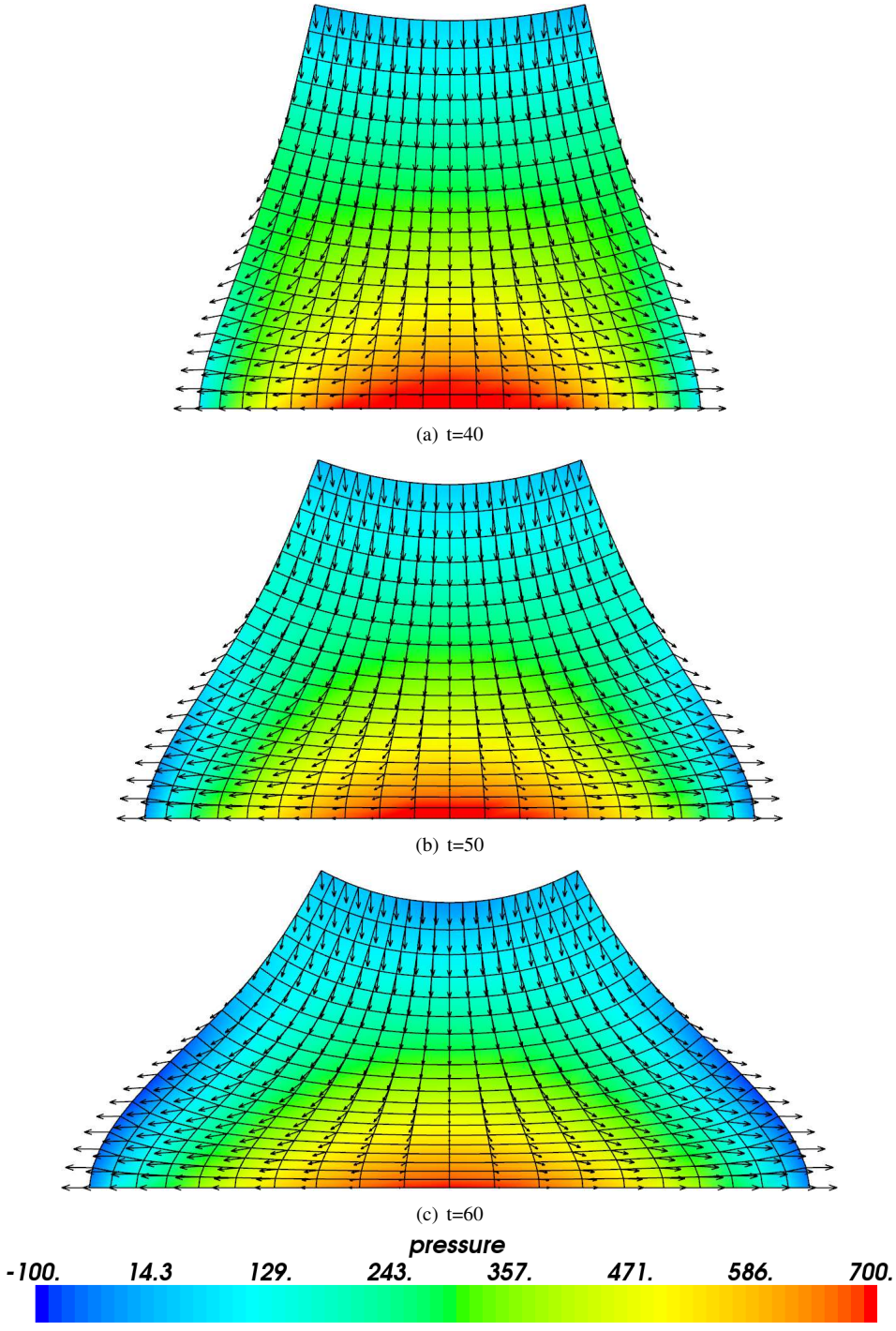
Colour coded scalar maps and velocity flow fields can be viewed by selecting them in the menu. The time-steps can be swept through to view a movie of the simulation. Figures 1.12 and 1.13 shows the simulation output. Velocity vectors and a colour map for pressure are shown. As the time progresses the body of fluid falls under the influence of gravity. The view used here to track the fluid is the Lagrangian view, since the mesh moves with the fluid. One of the disadvantages of using the Lagrangian view is that the elements in the mesh become severely distorted after a period of time and introduce solver errors. To get around this limitation the Level Set Method can be used, with the Eulerian point of view for a fixed mesh.




(a) t=1 (b) t=20

(c) t=30

FIGURE 1.12: Simulation output for Stokes flow. Fluid body starts off as a rectangular shape, then progresses downwards under the influence of gravity. Color coded distribution represents the scalar values for pressure. Velocity vectors are displayed at each node in the mesh to show the flow field. Computational mesh used was $20 \times 20$ elements.




(a) t=40

(b) t=50

(c) t=60

FIGURE 1.13: Simulation output for Stokes flow.

CHAPTER

**TWO**

# Execution of an *escript* Script

## 2.1 Overview

A typical way of starting your *escript* script 'myscript.py' is with the **escript** command[1]:

```
escript myscript.py
```

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

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

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

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

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

This will require *escript* to be compiled for *OpenMP* [18].

To start **escript** using *MPI* [14] with 8 processes you use

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

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

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

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

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

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

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

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

[1] The **escript** launcher is not supported under *MS Windows* yet.
[2] For this discussion, it is assumed that **escript** is included in your **PATH** environment. See installation guide for details.
[3] For simplicity, we will use the term node to refer to either a node in a super computer or an individual machine in a cluster

## 2.2 Options

The general form of the **escript** launcher is as follows:

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

The options are used as follows:

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

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

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

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

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

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

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

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

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

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

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

**-v** prints some diagnostic information.

### 2.2.1 Notes

- Make sure that **mpiexec** is in your **PATH**.
- For MPICH and INTELMPI and for the case a hostfile is present **escript** will start the **mpd** demon before execution.

## 2.3 Input and Output

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

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

## 2.4 Hints for MPI Programming

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

When *MPI* is used on more than one process ($nn \cdot np > 1$) the user needs to keep in mind that several copies of his script are executed at the same time [5] while data exchange is performed through the `esys.escript` module. At any time, `esys.escript` assumes that an argument of the type *int*, *float*, *str* and `numpy` has an identical value across all processors. All values of these types returned by `esys.escript` have the same value on all processors. If values produced by other modules are used as 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.

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

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

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

It is highly recommendable to use this class rather than the build `open` function as it will guarantee a script which will run in single processor mode as well as under *MPI* .

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

[4] That is, it has a non-empty value.
[5] In case of OpenMP only one copy is running but `esys.escript` temporarily spawns threads.

CHAPTER

**THREE**

# The Module `esys.escript`

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

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

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

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

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

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

The reduced smoothness for PDE solution is often used to fulfill the Ladyzhenskaya-Babuska-Brezzi condition [11] when solving saddle point problems , eg. the Stokes equation. A discontinuity is a region within the domain across which functions may be discontinuous. The location of discontinuity is defined in the `Domain` object. Figure 3.1 shows the dependency between the types of function spaces in Finley (other libraries may have different relationships).

The solution of a PDE is a continuous function. Any continuous function can be seen as a general function on the domain and can be restricted to the boundary as well as to one side of discontinuity (the result will be different depending on which side is chosen). Functions on any side of the discontinuity can be seen as a function on the corresponding other side.

A function on the boundary or on one side of the discontinuity cannot be seen as a general function on the domain as there are no values defined for the interior. For most PDE solver libraries the space of the solution and continuous functions is identical, however in some cases, eg. when periodic boundary conditions are used in `esys.finley`, a solution fulfills periodic boundary conditions while a continuous function does not have to be periodic.

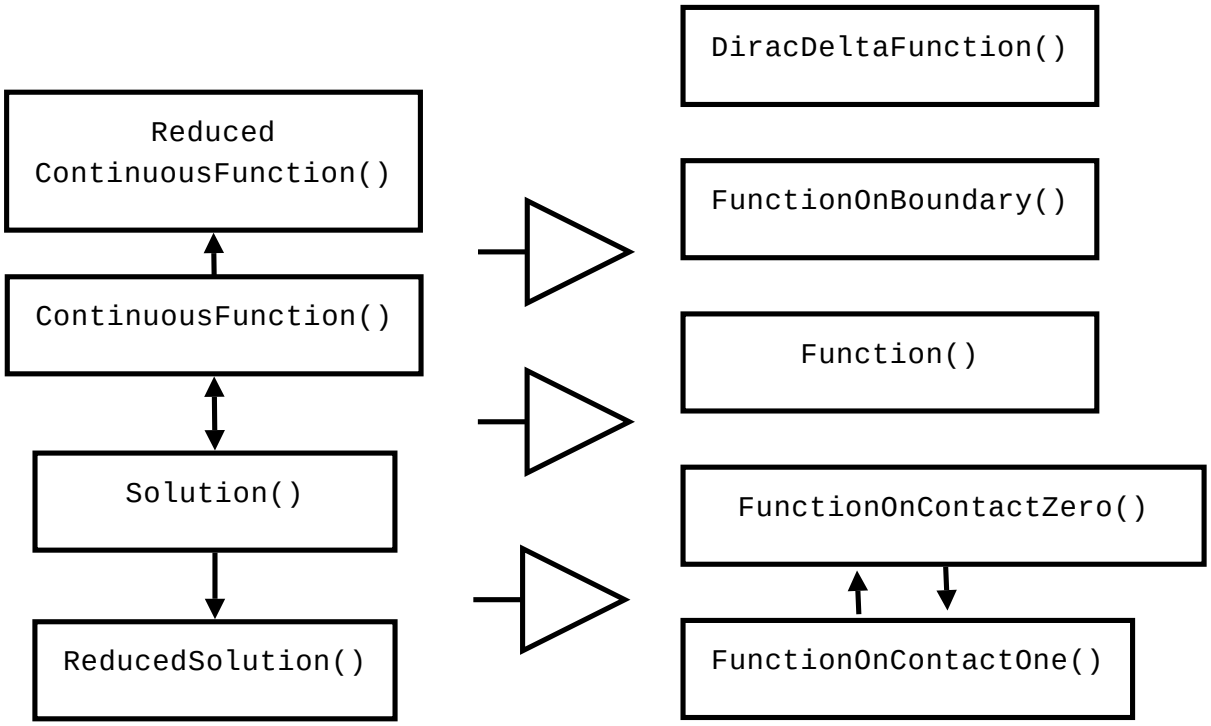



FIGURE 3.1: Dependency of Function Spaces in Finley. An arrow indicates that a function in the function space at the starting point can be interpolated to the function space of the arrow target. All functionspaces on the left side can be interpolated to any of the functionspaces on the right.

The concept of function spaces describes the properties of functions and allows abstraction from the actual representation of the function in the context of a particular application. For instance, in the FEM context a function of the general `FunctionSpace` type (written as *Function()* in Figure 3.1) is usually represented by its values at the element center, but in a finite difference scheme the edge midpoint of cells is preferred. By changing its function space you can use the same function in a Finite Difference scheme instead of Finite Element scheme. Changing the function space of a particular function will typically lead to a change of its representation. So, when seen as a general function, a continuous function which is typically represented by its values on the node of the FEM mesh or finite difference grid must be interpolated to the element centers or the cell edges, respectively. Interpolation happens automatically in `esys.escript` whenever it is required.

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

The values of the function have a rank which gives the number of indices, and a shape defining the range of each index. The rank in `esys.escript` is limited to the range $0$ through $4$ and it is assumed that the rank and shape is the same for all data sample points. The shape of a `Data` object is a tuple (list) *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` [6]. 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 a `numpy.ndarray` filled with ones. The shape of the created `Data` object it taken from the shape of the array. In the second case, the creator converts the initial value, which is a list of lists, and converts it into a `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))* is the same as *Data(0,Function(myDomain),(d,d))*, eg. a stress field.
- *Tensor4(0,Function(mydomain))* is the same as *Data(0,Function(myDomain),(d,d,d,d))* eg. 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 shape of the `Data` object to be created can be used as the initial value.

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

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

$$\sigma_{ij} = \lambda u_{k,k}\delta_{ij} + \mu(u_{i,j} + u_{j,i}) \tag{3.1}$$

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 *lam* 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
```

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

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

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

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

The function *lam* belongs to the continuous `FunctionSpace` function space but with *g* the function *trace(g)* is in the general `FunctionSpace` function space. In the evaluation of the product *lam*trace(g)* we have different function spaces (on the nodes versus in the centers) and at first glance we have incompatible data.

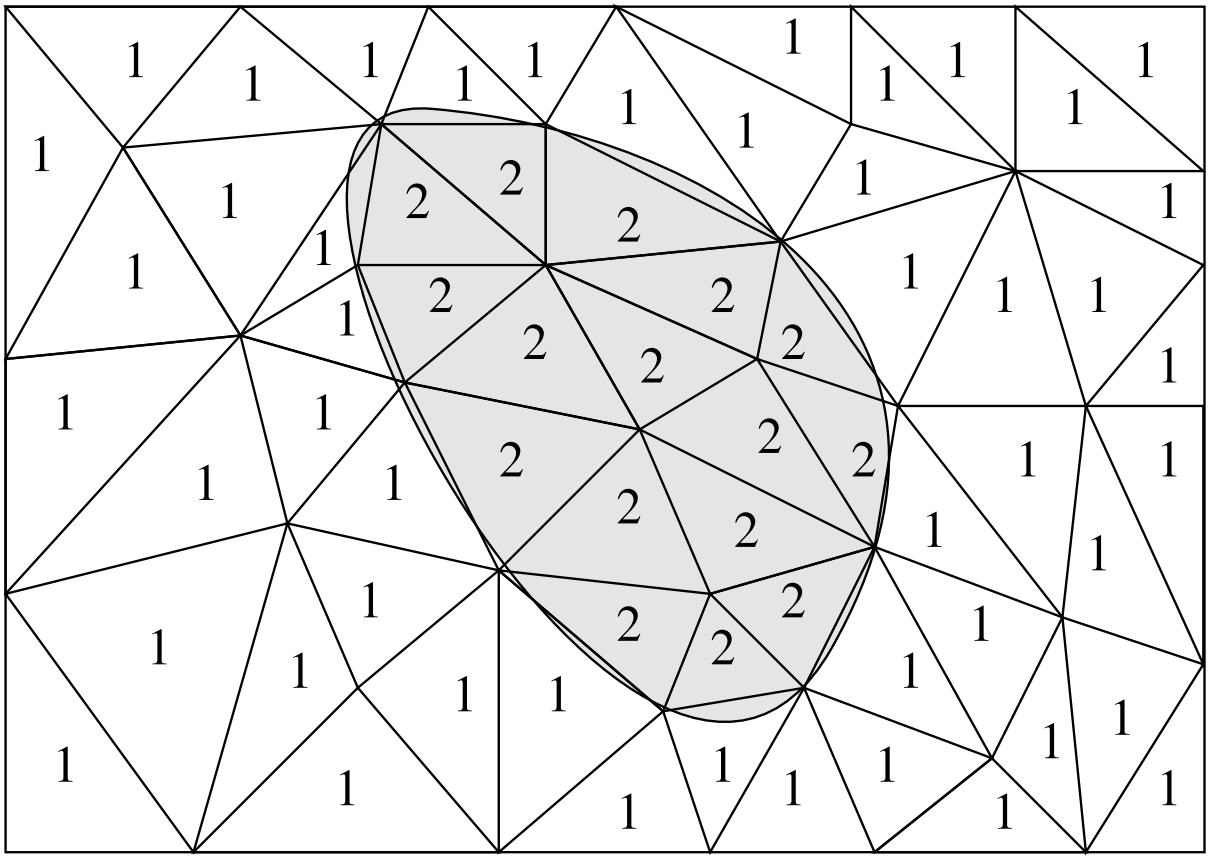


FIGURE 3.2: Element Tagging. A rectangular mesh over a region with two rock types *white* and *gray*. The number in each cell refers to the major rock type present in the cell (1 for *white* and 2 for *gray*).

`esys.escript` converts the arguments in an appropriate function space according to Table 3.1. In this example that means `esys.escript` sees *lam* as a function of the general `FunctionSpace` function space. In the context of FEM this means the nodal values of *lam* are interpolated to the element centers. The interpolation is automatic and requires no special handling.

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

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

```
lam=Scalar(value=2.,what=Function(mydomain))
insertTaggedValue(lam,white=30.,gray=5000.)
s=getStress(u,lam,2.)
```

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

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

There are three main ways in which `Data` objects are represented internally: constant, tagged, and expanded. In

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

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

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

`Data` objects can be written to disk files and read with *dump* and *load*, both of which use *netCDF* [16]. Use these to save data for visualization, checkpoint/restart or simply to save and reuse data that was expensive to compute.

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

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

To recover the object *x* and *mydomain* was a `esys.finley` mesh use

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

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

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

## 3.1 `esys.escript` Classes

### 3.1.1 `Domain` class

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

The following methods are available:

**`getDim`()**
returns the spatial dimension of the `Domain`.

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

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

**`setX`**(*newX*)
assigns a new location to the `Domain`. *newX* has to have shape $(d,)$ where $d$ is the spatial dimension of

the domain. Typically *newX* must be in the continuous `FunctionSpace` but the space actually to be used depends on the `Domain` implementation.

**getNormal**()
: returns the surface normals on the boundary of the `Domain` as `Data` object.

**getSize**()
: returns the local sample size, e.g. the element diameter, as `Data` object.

**setTagMap**(*tag_name, tag*)
: defines a mapping of the tag name *tag_name* to the *tag*.

**getTag**(*tag_name*)
: returns the tag associated with the tag name *tag_name*.

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

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

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

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

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

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

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

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

### 3.1.2 `FunctionSpace` class

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

The following methods are available:

**getDim**()
: returns the spatial dimension of the `Domain` of the `FunctionSpace`.

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

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

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

**getDomain**()
returns the Domain of the FunctionSpace.

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

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

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

__**str**__(*g*)
(python str() function) returns string representation of the Domain.

The following function provide generators for FunctionSpace objects:

**Function**(*domain*)
returns the general FunctionSpace on the Domain *domain*. Data objects in this type of general FunctionSpace are defined over the whole geometric region defined by *domain*.

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

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

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

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

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

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

### 3.1.3 Data Class

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

| expression | Description |
|---|---|
| *+arg0* | identical to *arg* |
| *-arg0* | negation |
| *arg0+arg1* | adds *arg0* and *arg1* |
| *arg0*arg1* | multiplies *arg0* and *arg1* |
| *arg0-arg1* | difference *arg1* from*arg1* |
| *arg0/arg1* | divide *arg0* by *arg1* |
| *arg0**arg1* | raises *arg0* to the power of *arg1* |

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:

| expression | Description |
|---|---|
| *arg0+=arg2* | adds *arg0* to *arg2* |
| *arg0*=arg2* | multiplies *arg0* with *arg2* |
| *arg0-=arg2* | subtracts *arg2* from*arg2* |
| *arg0/=arg2* | divides *arg0* by *arg2* |
| *arg0**=arg2* | raises *arg0* by *arg2* |

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

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

| rank of *arg* | slicing expression | shape of returned and assigned object |
|---|---|---|
| 0 | no slicing | - |
| 1 | *arg[l0:u0]* | (*u0-l0*,) |
| 2 | *arg[l0:u0,l1:u1]* | (*u0-l0*,*u1-l1*) |
| 3 | *arg[l0:u0,l1:u1,l2:u2]* | (*u0-l0*,*u1-l1*,*u2-l2*) |
| 4 | *arg[l0:u0,l1:u1,l2:u2,l3:u3]* | (*u0-l0*,*u1-l1*,*u2-l2*,*u3-l3*) |

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

$$0 \le l0 \le u0 \le s[0],$$

$$0 \le l1 \le u1 \le s[1],$$

$$0 \le l2 \le u2 \le s[2],$$

$$0 \le l3 \le u3 \le s[3].$$

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

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

### 3.1.4 Generation of `Data` objects

**class `Data`** (*value=0,shape=(,),what=FunctionSpace(),expand=`False`*)
creates a `Data` object with shape *shape* in the `FunctionSpace` *what*. The values at all data sample points are set to the double value *value*. If *expanded* is `True` the `Data` object is represented in expanded from.

**class `Data`** (*value,what=FunctionSpace(),expand=`False`*)
creates a `Data` object in the `FunctionSpace` *what*. The value for each data sample points is set to *value*, which could be a `numpy`, `Data` object *value* 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(),expand=`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 from.

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

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

**`Tensor3`** (*value=0.,what=FunctionSpace(),expand=`False`*)
returns a `Data` object of shape *(d,d,d)* in the `FunctionSpace` *what*, where *d* is the spatial dimension of the `Domain` of *what*. Values are initialed with *value*, a double precision quantity. If *expanded* is `True` the `Data` object is re*arg*presented in expanded from.

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

**`load`** (*filename,domain*)
recovers a `Data` object on `Domain` *domain* from the file *filename*, which was created by `dump`.

### 3.1.5 `Data` methods

These are the most frequently-used methods of the `Data` class. A complete list of methods can be found on http://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` id the `Data` object is the empty `Data` object. Otherwise `False` is returned. Note that this is not the same as asking if the object contains no data sample points.

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

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

**__str__**()
returns a string representation of the object.

### 3.1.6 Functions of `Data` objects

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

**saveVTK** (*filename,**kwdata*)
writes `Data` defined by keywords in the file with *filename* using the vtk file format *VTK* file format. The key word is used as an identifier. The statement

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

will write the scalar *T* as *temperature* and the vector *v* as *velocity* into the file 'out.xml'. Restrictions on the allowed combinations of `FunctionSpace` apply.

**saveDX** (*filename,**kwdata*)
writes `Data` defined by keywords in the file with *filename* using the vtk file format *OpenDX* [17] file format. The key word is used as an identifier. The statement

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

will write the scalar *T* as *temperature* and the vector *v* as *velocity* into the file 'out.dx'. Restrictions on the allowed combinations of `FunctionSpace` apply.

**kronecker** ($d$)
returns a rank-2 `Data` object `Data` object in `FunctionSpace` $d$ such that

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

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

**identityTensor** ($d$)
is a synonym for `kronecker` (see above).

**identityTensor4** ($d$)
returns a rank-4 `Data` object `Data` object in `FunctionSpace` $d$ such that

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

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

**unitVector** (*i,d*)
returns a rank-1 `Data` object `Data` object in `FunctionSpace` $d$ such that

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

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 points the minimum value over all components.

**maxval**(*a*)
returns at each data sample points the maximum value over all components.

**length**(*a*)
returns at Euclidean norm at each data sample points. For a rank-4 `Data` object *a* this is

$$\texttt{length(a)} = \sqrt{\sum_{ijkl} a\left[i,j,k,l\right]^2} \tag{3.5}$$

**trace**(*a*[,*axis_offset=0*])
returns the trace of *a*. This is the sum over components *axis_offset* and *axis_offset+1* with the same index. For instance in the case of a rank-2 `Data` object function and this is

$$\texttt{trace(a)} = \sum_i a\left[i,i\right] \tag{3.6}$$

and for a rank-4 `Data` object function and `axis_offset=1` this is

$$\texttt{trace(a,1)}\ \left[i,j\right] = \sum_k a\left[i,k,k,j\right] \tag{3.7}$$

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

$$\texttt{transpose(a)}\ \left[i,j\right] = a\left[j,i\right] \tag{3.8}$$

and for a rank-4 `Data` object function and `axis_offset=1` this is

$$\texttt{transpose(a,1)}\ \left[i,j,k,l\right] = a\left[j,k,l,i\right] \tag{3.9}$$

**swap_axes**(*a*[, *axis0=0* [, *axis1=1* ]])
returns *a* but with swapped components *axis0* and *axis1*. The argument *a* must be at least of rank-2 `Data` object. For instance in the for a rank-4 `Data` object argument, `axis0=1` and `axis1=2` this is

$$\texttt{swap\_axes(a,1,2)}\ \left[i,j,k,l\right] = a\left[i,k,j,l\right] \tag{3.10}$$

**symmetric**(*a*)
returns the symmetric part of *a*. This is `(a+transpose(a))/2`.

**nonsymmetric**(*a*)
returns the non–symmetric part of *a*. This is `(a-transpose(a))/2`.

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

$$\texttt{matrix\_mult(inverse(a),a)=kronecker(d)} \tag{3.11}$$

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

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

$$\texttt{matrix\_mult(a,V)=e[i]*V} \tag{3.12}$$

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

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

$$\texttt{matrix\_mult(a,V[:,i])=e[i]*V[:,i]} \tag{3.13}$$

where `e,V=eigenvalues_and_eigenvectors(a)`. The eigenvectors *V* are orthogonal and normalized, ie.

$$\texttt{matrix\_mult(transpose(V),V)=kronecker(d)} \tag{3.14}$$

if *a* has shape `(d,d)`. The eigenvalues are ordered in increasing size. The argument *a* has to be the symmetric, ie. `a=symmetric(a)`. The current implementation is restricted to arguments of shape `(2,2)` and `(3,3)`.

**maximum**(**a*)
returns the maximum value over all arguments at all data sample points and for each component. For instance

$$\texttt{maximum(a0,a1)}\ [i,j] = max(a0\,[i,j]\,, a1\,[i,j]) \tag{3.15}$$

at all data sample points.

**minimum**(**a*)
returns the minimum value over all arguments at all data sample points and for each component. For instance

$$\texttt{minimum(a0,a1)}\ [i,j] = min(a0\,[i,j]\,, a1\,[i,j]) \tag{3.16}$$

at all data sample points.

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

**inner**(*a0,a1*)
returns the inner product of *a0* and *a1*. For instance in the case of rank-2 `Data` object arguments and this is

$$\texttt{inner(a)} = \sum_{ij} a0\,[j,i] \cdot a1\,[j,i] \tag{3.17}$$

and for a rank-4 `Data` object arguments this is

$$\texttt{inner(a)} = \sum_{ijkl} a0\,[i,j,k,l] \cdot a1\,[j,i,k,l] \tag{3.18}$$

**matrix_mult**(*a0,a1*)
returns the matrix product of *a0* and *a1*. If *a1* is rank-1 `Data` object this is

$$\texttt{matrix\_mult(a)}\ [i] = \sum_{k} a0 \cdot [i,k]\, a1\,[k] \tag{3.19}$$

and if *a1* is rank-2 `Data` object this is

$$\texttt{matrix\_mult(a)}\ [i,j] = \sum_{k} a0 \cdot [i,k]\, a1\,[k,j] \tag{3.20}$$

**transposed_matrix_mult**(*a0,a1*)
returns the matrix product of the transposed of *a0* and *a1*. The function is equivalent to `matrix_mult(transpose(a0),a1)`. If *a1* is rank-1 `Data` object this is

$$\texttt{transposed\_matrix\_mult(a)}\,[i] = \sum_k a0 \cdot [k,i]\, a1\,[k] \tag{3.21}$$

and if *a1* is rank-2 `Data` object this is

$$\texttt{transposed\_matrix\_mult(a)}\,[i,j] = \sum_k a0 \cdot [k,i]\, a1\,[k,j] \tag{3.22}$$

**matrix_transposed_mult**(*a0,a1*)
returns the matrix product of *a0* and the transposed of *a1*. The function is equivalent to `matrix_mult(a0,transpose(a1))`. If *a1* is rank-2 `Data` object this is

$$\texttt{matrix\_transposed\_mult(a)}\,[i,j] = \sum_k a0 \cdot [i,k]\, a1\,[j,k] \tag{3.23}$$

**outer**(*a0,a1*)
returns the outer product of *a0* and *a1*. For instance if *a0* and *a1* both are rank-1 `Data` object then

$$\texttt{outer(a)}\,[i,j] = a0\,[i] \cdot a1\,[j] \tag{3.24}$$

and if *a0* is rank-1 `Data` object and *a1* is rank-3 `Data` object

$$\texttt{outer(a)}\,[i,j,k] = a0\,[i] \cdot a1\,[j,k] \tag{3.25}$$

**tensor_mult**(*a0,a1*)
returns the tensor product of *a0* and *a1*. If *a1* is rank-2 `Data` object this is

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

and if *a1* is rank-4 `Data` object this is

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

**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 rank-2 `Data` object this is

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

and if *a1* is rank-4 `Data` object this is

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

**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 rank-2 `Data` object this is

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

and if *a1* is rank-4 `Data` object this is

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

**grad**(*a*[, *where=None* ])
returns the gradient of *a*. If *where* is present the gradient will be calculated in `FunctionSpace` *where* otherwise a default `FunctionSpace` is used. In case that *a* has rank-2 `Data` object one has

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

**integrate**(*a*[ ,*where=None* ])
returns the integral of *a* where the domain of integration is defined by the `FunctionSpace` of *a*. If *where* is present the argument is interpolated into `FunctionSpace` *where* before integration. For instance in the case of a rank-2 `Data` object argument in continuous `FunctionSpace` it is

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

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

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

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

**interpolate**(*a,where*)
interpolates argument *a* into the `FunctionSpace` *where*.

**div**(*a*[ ,*where=None* ])
returns the divergence of *a*. This

$$\texttt{div(a)} = trace(grad(a), where) \tag{3.35}$$

**jump**(*a*[ ,*domain=None* ])
returns the jump of *a* over the discontinuity in its domain or if `Domain` *domain* is present in *domain*.

$$\begin{aligned}\texttt{jump(a)} \ = \ & \texttt{interpolate(a,FunctionOnContactOne(domain))} \\ & -\texttt{interpolate(a,FunctionOnContactZero(domain))}\end{aligned} \tag{3.36}$$

**L2**(*a*)
returns the $L^2$-norm of *a* in its function space. This is

$$\texttt{L2(a)=integrate(length(a)}^2\texttt{)}\ . \tag{3.37}$$

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

**sin**(*a*)
applies sine function to *a*.

**cos**(*a*)
applies cosine function to *a*.

**tan**(*a*)
applies tangent function to *a*.

**asin**(*a*)
applies arc (inverse) sine function to *a*.

**acos**(*a*)
applies arc (inverse) cosine function to *a*.

**atan**(*a*)
applies arc (inverse) tangent function to *a*.

**sinh**(*a*)
applies hyperbolic sine function to *a*.

**cosh**(*a*)
applies hyperbolic cosine function to *a*.

**tanh**(*a*)
applies hyperbolic tangent function to *a*.

**asinh**(*a*)
applies arc (inverse) hyperbolic sine function to *a*.

**acosh**(*a*)
applies arc (inverse) hyperbolic cosine function to *a*.

**atanh**(*a*)
applies arc (inverse) hyperbolic tangent function to *a*.

**exp**(*a*)
applies exponential function to *a*.

**sqrt**(*a*)
applies square root function to *a*.

**log**(*a*)
applies the natural logarithm to *a*.

**log10**(*a*)
applies the base-10 logarithm to *a*.

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

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

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

**whereNonNegative**(*a*)
returns a function which is 1 where *a* is non–negative and 0 otherwise.

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

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

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

### 3.1.7 `Operator` Class

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

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

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

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

**solves**(*rhs*)

solves the operator equation with right hand side *rhs*

**of**(*u*)
: applies the operator to the `Data` object *u*

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

## 3.2 Physical Units

`esys.escript` provides support for physical units in the SI system including unit conversion. So the user can define variables in the form

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

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

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

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

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

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

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

can be used print the pressure in units of atmosphere.

This is an incomplete list of supported physical units:

**km**
: unit of kilo meter

**m**
: unit of meter

**cm**
: unit of centi meter

**mm**
: unit of milli meter

**sec**
: unit of second

**minute**
: unit of minute

**h**
: unit of hour

**day**
: unit of day

**yr**
: unit of year

**gram**
unit of gram

**kg**
unit of kilo gram

**lb**
unit of pound

**ton**
metric ton

**A**
unit of Ampere

**Hz**
unit of Hertz

**N**
unit of Newton

**Pa**
unit of Pascal

**atm**
unit of atmosphere

**J**
unit of Joule

**W**
unit of Watt

**C**
unit of Coulomb

**V**
unit of Volt

**F**
unit of Farad

**Ohm**
unit of Ohm

**K**
unit of Kelvin

**Celsius**
unit of Celsius

**Fahrenheit**
unit of Fahrenheit

Moreover unit prefixes are supported:

**Yotta**
prefix yotta = $10^{24}$.

**Zetta**
prefix zetta= $10^{21}$.

**Exa**
prefix exa= $10^{18}$.

**Peta**
prefix peta= $10^{15}$.

**Tera**
prefix tera= $10^{12}$.

**Giga**
prefix giga= $10^9$.

**Mega**
prefix mega= $10^6$.

**Kilo**
prefix kilo= $10^3$.

**Hecto**
prefix hecto= $10^2$.

**Deca**
prefix deca= $10^1$.

**Deci**
prefix deci= $10^{-1}$.

**Centi**
prefix centi= $10^{-2}$.

**Milli**
prefix milli= $10^{-3}$.

**Micro**
prefix micro= $10^{-6}$.

**Nano**
prefix nano= $10^{-9}$.

**Pico**
prefix pico= $10^{-12}$.

**Femto**
prefix femto= $10^{-15}$.

**Atto**
prefix atto= $10^{-18}$.

**Zepto**
prefix zepto= $10^{-21}$.

**Yocto**
prefix yocto= $10^{-24}$.

## 3.3 Utilities

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

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

The following methods are available:

**`close`**()
closes the file.

**`flush`**()
flushes the internal buffer to disk.

**`write`**(*txt*)

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

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

**closed**
`True` if file is closed.

**mode**
access mode.

**name**
file name.

**newlines**
line separator

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

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

**listEscriptParams**(*a*)
returns a list of valid parameters and their description.

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

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

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

**getMPIWorldMax**(*a*)
returns the maximum value of the integer *a* across all processors within **MPI_COMM_WORLD**.

CHAPTER

**FOUR**

# The Module esys.escript.linearPDEs

## 4.1 Linear Partial Differential Equations

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

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

$$-(A_{jl}u_{,l})_{,j} - (B_j u)_{,j} + C_l u_{,l} + Du = -X_{j,j} + Y \ . \tag{4.1}$$

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

$$n_j(A_{jl}u_{,l} + B_j u) + du = n_j X_j + y \ . \tag{4.2}$$

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

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

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

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

$$-(A_{ijkl}u_{k,l})_{,j} - (B_{ijk}u_k)_{,j} + C_{ikl}u_{k,l} + D_{ik}u_k = -X_{ij,j} + Y_i \ . \tag{4.4}$$

$A$ 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:

$$n_j(A_{ijkl}u_{k,l} + B_{ijk}u_k) + d_{ik}u_k = n_j X_{ij} + y_i \ . \tag{4.5}$$

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

$$u_i = r_i \text{ where } q_i > 0 \tag{4.6}$$

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

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

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

and several components of the solution defined as

$$J_{ij} = A_{ijkl} u_{k,l} + B_{ijk} u_k - X_{ij} \tag{4.7}$$

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

$$J_j = A_{jl} u_{,l} + B_j u_k - X_j \tag{4.8}$$

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

$$n_j J^0_{ij} = n_j J^1_{ij} = y^{contact}_i - d^{contact}_{ik} [u]_k \ . \tag{4.9}$$

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

$$n_j J^0_j = n_j J^1_j = y^{contact} - d^{contact} [u] \tag{4.10}$$

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

The PDE is symmetrical if

$$A_{jl} = A_{lj} \text{ and } B_j = C_j \tag{4.11}$$

The system of PDEs is symmetrical if

$$\begin{aligned} A_{ijkl} &= A_{klij} && (4.12)\\ B_{ijk} &= C_{kij} && (4.13)\\ D_{ik} &= D_{ki} && (4.14)\\ d_{ik} &= d_{ki} && (4.15)\\ d^{contact}_{ik} &= d^{contact}_{ki} && (4.16) \end{aligned}$$

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

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

```
from esys.escript import *
from esys.escript.linearPDEs import LinearPDE
from esys.finley import Rectangle
mydomain = Rectangle(l0=1.,l1=1.,n0=40, n1=20)
mypde=LinearPDE(mydomain)
mypde.setSymmetryOn()
mypde.setValue(A=kappa*kronecker(mydomain),D=1,Y=1)
u=mypde.getSolution()
```

We refer to chapter 1 for more details.

An instance of the `SolverOptions` class is attached to the `LinearPDE` class object. It is used to set options of the solver used to solve the PDE. In the following code the `getSolverOptions` is used to access the `SolverOptions` attached to *mypde*:

```
from esys.escript import *
from esys.escript.linearPDEs import LinearPDE, SolverOptions
from esys.finley import Rectangle
mydomain = Rectangle(l0=1.,l1=1.,n0=40, n1=20)
mypde=LinearPDE(mydomain)
mypde.setValue(A=kappa*kronecker(mydomain),D=1,Y=1)
mypde.getSolverOptions().setVerbosityOn()
mypde.getSolverOptions().setSolverMethod(SolverOptions.PCG)
mypde.getSolverOptions().setPreconditioner(SolverOptions.AMG)
mypde.getSolverOptions().setTolerance(1e-8)
mypde.getSolverOptions().setIterMax(1000)
u=mypde.getSolution()
```

In this code the preconditoned conjugate gradient method `SolverOptions.PCG` with preconditioner `SolverOptions.AMG` . The relative tolerance is set tto $10^{-8}$ and the maximum number of iteration steps to 1000.

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

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

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

### 4.1.1 Classes

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

### 4.1.2 `LinearPDE` class

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

**class `LinearPDE`** (*domain,numEquations=0,numSolutions=0*)
opens a linear, steady, second order PDE on the `Domain` *domain*. *numEquations* and *numSolutions* gives the number of equations and the number of solution components. If *numEquations* and *numSolutions* is non-positive, the number of equations and the number solutions, respectively, stay undefined until a coefficient is defined.

#### `LinearPDE` methods

**`setValue`** ( [*A*][, *B*], [, *C*][, *D*] [, *X*][, *Y*] [, *d*][, *y*] [, *d_contact*][, *y_contact*] [, *q*][, *r*])
assigns new values to coefficients. By default all values are assumed to be zero[2] If the new coefficient value is not a `Data` object, it is converted into a `Data` object in the appropriate `FunctionSpace`.

**`getCoefficient`** (*name*)
return the value assigned to coefficient *name*. If *name* is not a valid name an exception is raised.

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

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

**`setDebugOn`** ()
switches on debug mode.

**`setDebugOff`** ()
switches off debug mode.

**`getSolverOptions`** ()
returns the solver options for solving the PDE. In fact the method returns a `SolverOptions` class object which can be used to modify the tolerance, the solver or the preconditioner, see Section 4.2 for details.

[2] In fact it is assumed they are not present by assigning the value `escript.Data()`. The can by used by the solver library to reduce computational costs.

**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 4.2 for details. Otherwise the solver options are reset to the default.

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

**getDomain**()
returns the `Domain` of the PDE.

**getDim**()
returns the spatial dimension of the PDE.

**getNumEquations**()
returns the number of equations.

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

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

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

**isSymmetric**()
returns `True` if the PDE has been indicated to be symmetric. Otherwise `False` is returned.

**setSymmetryOn**()
indicates that the PDE is symmetric.

**setSymmetryOff**()
indicates that the PDE is not symmetric.

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

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

**getOperator**()
returns the `Operator` of the PDE.

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

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

**getSolution**()
returns (an approximation of) the solution of the PDE. This call will invoke the discretization of the PDE and the solution of the resulting system of linear equations. Keep in mind that this call is typically computational expensive and can - depending on the PDE and the discretiztion - 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

$$-u_{,ii} = f \ . \tag{4.17}$$

with homogeneous boundary conditions

$$n_i u_{,i} = 0 \tag{4.18}$$

and homogeneous constraints

$$u = 0 \text{ where } q > 0 \tag{4.19}$$

$f$ 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

$$\omega\ u - (k\ u_{,j})_{,j} = f \tag{4.20}$$

with natural boundary conditions

$$k\ u_{,j} n_{,j} = g - \alpha\ u \tag{4.21}$$

and constraints:

$$u = r \text{ where } q > 0 \tag{4.22}$$

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

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

**`setValue`** ( [*omega*] [, *k*] [, *f*] [, *alpha*] [, *g*] [, *r*] [, *q*] )
assigns new values to *omega*, *k*, *f*, *alpha*, *g*, *r*, *q*. By default all values are set to be zero.

### 4.1.5 The `Lame` Class

The `Lame` class defines a Lame equation problem:

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

with natural boundary conditions:

$$n_j(\mu\ (u_{i,j} + u_{j,i}) + \lambda * u_{k,k}) = f_i + n_j \sigma_{ij} \tag{4.24}$$

and constraint

$$u_i = r_i \text{ where } q_i > 0 \tag{4.25}$$

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

**class `Lame`** (*domain*)
opens a Lame equation on the `Domain` domain. `Lame` is derived from `LinearPDE`.

**`setValue`** ( [*lame_lambda*] [, *lame_mu*] [, *F*] [, *sigma*] [, *f*] [, *r*] [, *q*] )
assigns new values to *lame_lambda*, *lame_mu*, *F*, *sigma*, *f*, *r* and *q* By default all values are set to be zero.

## 4.2 Solver Options

**class `SolverOptions`** ()
This class defines the solver options for a linear or non-linear solver. The option also supports the handling of diagnostic informations.

**getSummary**()
Returns a string reporting the current settings

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

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

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

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

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

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

**getPackage**()
Returns the solver package key

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

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

- "num_iter": the number of iteration steps
- "cum_num_iter": the cumulative number of iteration steps
- "num_level": the number of level in multi level solver
- "num_inner_iter": the number of inner iteration steps
- "cum_num_inner_iter": the cumulative number of inner iteration steps
- "time": execution time
- "cum_time": cumulative execution time
- "set_up_time": time to set up of the solver, typically this includes factorization and reordering
- "cum_set_up_time": cumulative time to set up of the solver

- "net_time": net execution time, excluding setup time for the solver and execution time for preconditioner
- "cum_net_time": cumulative net execution time
- "residual_norm": norm of the final residual
- "converged": return self.__converged

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

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

**getCoarsening**()
Returns the key of the coarsening algorithm to be applied `SolverOptions.AMG` .

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

**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 performing a restart. If *restart* is equal to *None* no restart is performed.

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

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

**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=0.05*])
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

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

**getMinCoarseMatrixSize**()

Returns the minumum size of the coarsest level matrix in AMG.

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

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

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

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

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

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

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

**getTolerance**()
Returns the relative tolerance for the solver

**setAbsoluteTolerance**([*atol=0.*])
Sets the absolute tolerance for the solver. The actually meaning of tolerance depends on the underlying PDE library. In most cases, the tolerance will only consider the error from solving the discrete problem but will not consider any discretization error.

**getAbsoluteTolerance**()
Returns the absolute tolerance for the solver

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

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

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

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

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

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

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

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

**isSymmetric**()
Returns `True` is the descrete system is indicated as symmetric.

**setSymmetryOn**()

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

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

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

**setVerbosityOn**()
Switches the verbosity of the solver on.

**setVerbosityOff**()
Switches the verbosity of the solver off.

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

**setInnerToleranceAdaptionOn**()
Switches the automatic selection of inner tolerance on

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

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

**getInnerIterMax**()
Returns maximum number of inner iteration steps.

**acceptConvergenceFailure**()
Returns `True` if a failure to meet the stopping criteria within the given number of iteration steps is not raising in exception. This is useful if a solver is used in a non-linear context where the non-linear solver can continue even if the returned the 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 [13][3].

**UMFPACK**
the `UMFPACK` , Reference [23]. Remark: `UMFPACK` is not parallelized.

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

**ITERATIVE**
the default iterative method and preconditioner. The actually used method depends on the PDE solver library and the solver package been chosen. Typically, `SolverOptions.PCG` is used for symmetric PDEsand `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 [20]. The solver will require a symmetric PDE.

[3]The `MKL` library will only be available when the Intel compilation environment is used.

**PCG**
preconditioned conjugate gradient method, see Reference [25]. The solver will require a symmetric PDE.

**TFQMR**
transpose-free quasi-minimal residual method, see Reference [25].

**GMRES**
the GMRES method, see Reference [25]. Truncation and restart are controlled by the parameters *truncation* and *restart* of `getSolution`.

**MINRES**
minimal residual method method,

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

**PRES20**
the GMRES method with truncation after five residuals and restart after 20 steps, see Reference [25].

**CGS**
conjugate gradient squared method, see Reference [25].

**BICGSTAB**
stabilized bi-conjugate gradients methods, see Reference [25].

**SSOR**
symmetric successive over-relaxation method, see Reference [25]. 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 [20].

**ILUT**
the incomplete LU factorization preconditioner with fill-in, see Reference [20]. During the LU-factorization element with relative size less then `getDropTolerance` are dropped. Moreover, the size of the LU-factorization is restricted to the `getDropStorage`-fold of the stiffness matrix. `getDropTolerance` and `getDropStorage` are both set in the `getSolution` call.

**JACOBI**
the Jacobi preconditioner, see Reference [20].

**AMG**
the algebraic–multi grid method, see Reference [21]. This method can be used as linear solver method but is more robust when used in a preconditioner.

**GAUSS_SEIDEL**
the symmetric Gauss-Seidel preconditioner, see Reference [20]. `getNumSweeps()` is the number of sweeps used.

**RILU**
relaxed incomplete LU factorization preconditioner, see Reference [**?**]. This method is similar to `SolverOptions.ILU0` 0 but dropped elements are added to the main diagonal with the relaxation factor `getRelaxationFactor`

**REC_ILU**
recursive incomplete LU factorization preconditioner, see Reference [27]. This method is similar to `SolverOptions.ILU0` 0 but applies reordering during the factorization.

**NO_REORDERING**
no ordering is used during factorization.

**DEFAULT_REORDERING**
the default reordering method during factorization.

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

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

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

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

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

**YAIR_SHAPIRA_COARSENING**
`SolverOptions.AMG` coarsening method by Yair-Shapira

**RUGE_STUEBEN_COARSENING**
`SolverOptions.AMG` coarsening method by Ruge and Stueben

**AGGREGATION_COARSENING**
`SolverOptions.AMG` coarsening using (symmetric) aggregation

**NO_PRECONDITIONER**
no preconditioner is applied.

CHAPTER

**FIVE**

# The Module `esys.pycad`

## 5.1 Introduction

`esys.pycad` provides a simple way to build a mesh for your finite element simulation. You begin by building what we call a *Design* using primitive geometric objects, and then to go on to build a mesh from the *Design*. The final step of generating the mesh from a *Design* uses freely available mesh generation software, such as *Gmsh*[10] .

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

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

```
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, integrationOrder=-1, reducedIntegrationOrder=-1, optimizeLabeling=Tru

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

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

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

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

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

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

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

## 5.2 `esys.pycad` Classes

### 5.2.1 Primitives

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

**class `Point`** (*x=0.,y=0.,z=0.*[,*local_scale=1.*])
Create a point with from coordinates with local characteristic length *local_scale*

**class `Line`** (*point1, point2*)
Create a line with between starting and ending 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 naighboured elements. If *createBump* is set progression is applied towards the center of the line.

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

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

**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 line. If set it overwrites the local length setting which would be applied. The progression factor *progression* defines the change of element size between naighboured elements. If *createBump* is set progression is applied towards the center of the line.

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

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

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

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

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

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

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

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

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

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

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

**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 naighboured elements. If *createBump* is set progression is applied towards the center of the line.

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

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

**class CurveLoop**(*list*)
Create a closed curve from the `list` of `Line`, `Arc`, `Spline`, `BSpline`, `BrezierSpline`.

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

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

**setTransfiniteMeshing**([*orientation="Left"*])
applies 2D transfinite meshing to the surface. *orientation* defines the orientation of triangles. Allowed values are *"Left"*, *"Right"* or *"Alternate"*. The boundary of the surface muist be defined by three or four lines where an element distribution must be defined on all faces where opposite faces uses the same element

distribution. No holes must be present.

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

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

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

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

**class `Volume`** (*loop,* [*holes=[list]* ])
Create a volume given a `SurfaceLoop`, which may have one or more holes define by the list of `SurfaceLoop`.

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

### 5.2.2 Transformations

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

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

**class `Rotatation`** ( [*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 normalized but must have positive length. The right hand rule [1] applies.

**class `Dilation`** ( [*factor=1.,* [*center=[0,0,0]* ] ] )
defines a dilation by the expansion/contraction *factor* with *center* as the dilation center. *center* can be any object that can be converted into a `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 normalized but must have positive length.

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

### 5.2.3 Properties

If you are building a larger geometry you may find it convenient to create it in smaller pieces and then assemble them into the whole. Property sets make this easy, and they allow you to name the smaller pieces for convenience.

Property sets are used to bundle a set of geometrical objects in a group. The group is identified by a name. Typically a property set is used to mark subregions with share the same material properties or to mark portions of

the boundary. For efficiency, the `Design` class object assigns a integer to each of its property sets, a so-called tag . The appropriate tag is attached to the elements at generation time.

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

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

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

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

**`getName`**()
returns the name of the set

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

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

**`getItems`**()
returns the list of items

**`clearItems`**()
clears the list of items

**`getTag`**()
returns the tag used for this property set

## 5.3 Interface to the mesh generation software

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

**class `Design`** ( [*dim=3,* [*element_size=1.,* [*order=1,* [*keep_files=False* ] ] ] ] )
The `Design` describes the geometry defined by primitives to be meshed. The *dim* specifies the spatial dimension. The argument *element_size* defines the global element size which is multiplied by the local scale to set the element size at each `Point` . The argument *order* defines the element order to be used. If *keep_files* is set to `True` temporary files a kept otherwise they are removed when the instance of the class is deleted.

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

**`getDim`**()
returns the spatial dimension.

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

**`getElementOrder`**()
returns the element order.

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

**`getElementSize`**()
returns the global element size.

**DELAUNAY**
the gmsh Delauny triangulator.

**TETGEN**
the TetGen [22] triangulator.

**NETGEN**
the NETGEN [9] triangulator.

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

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

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

**setScriptFileName**([*name=None*])
set the filename for the gmsh input script. if no name is given a name with extension "geo" is generated.

**getScriptFileName**()
returns the name of the file for the gmsh script.

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

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

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

**getItems**()
returns a list of the items

**clearItems**()
resets the items in design

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

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

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

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

**getTagMap**()
returns a `TagMap` to map the name `PropertySet` in the class to tag numbers generated by gmsh.

CHAPTER

**SIX**

# Models

The following sections give a breif overview of the model classes and their corresponding methods.

## 6.1 Stokes Problem

The velocity field $v$ and pressure $p$ of an incompressible fluid is given as the solution of the Stokes problem

$$-\left(\eta(v_{i,j}+v_{i,j})\right)_{,j}+p_{,i}=f_i-\sigma_{ij,j} \tag{6.1}$$

where $\eta$ is the viscosity, $F_i$ defines an internal force and $\sigma_{ij}$ is an intial stress . We assume an incompressible media:

$$-v_{i,i}=0 \tag{6.2}$$

Natural boundary conditions are taken in the form

$$\left(\eta(v_{i,j}+v_{i,j})\right)n_j-n_ip=s_i+\sigma_{ij}n_j \tag{6.3}$$

which can be overwritten by constraints of the form

$$v_i(x)=v_i^D(x) \tag{6.4}$$

at some locations $x$ at the boundary of the domain. The index $i$ may depend on the location $x$ on the boundary. $v^D$ is a given function on the domain.

### 6.1.1 Solution Method

In block form equation equations 6.1 and 6.2 takes the form of a saddle point problem

$$\begin{bmatrix} A & B^* \\ B & 0 \end{bmatrix}\begin{bmatrix} v \\ p \end{bmatrix}=\begin{bmatrix} G \\ 0 \end{bmatrix} \tag{6.5}$$

where $A$ is coercive, self-adjoint linear operator in a suitable Hilbert space, $B$ is the $(-1)\cdot$ divergence operator and $B^*$ is it adjoint operator (=gradient operator). For more details on the mathematics see references [3, 4]. We use iterative techniques to solve this problem. To make sure that the incomressibilty condition holds with sufficient accuracy we check for

$$\|v_{k,k}\|_{\leq}\epsilon\|\sqrt{v_{j,k}v_{j,k}}\| \tag{6.6}$$

where $\epsilon$ is the desired relative accuracy and

$$\|p\|^2=\int_\Omega p^2\,dx \tag{6.7}$$

defines the $L^2$-norm. We use the Uzawa scheme to solve the problem.

In fact the first equation in 6.5 gives for a known pressure

$$v=A^{-1}(G-B^*p) \tag{6.8}$$

which is inserted into the second equation leading to

$$Sp = BA^{-1}G \tag{6.9}$$

with the Schur complement $S = BA^{-1}B^*$. This problem can be solved iteratively with the preconditioner $\hat{S}$ defined as $q = \hat{S}^{-1}p$ by solving

$$\frac{1}{\eta}q = p \tag{6.10}$$

see [8] for more details. Note that the residual for the current approximation $p$ is given as

$$r = BA^{-1}(G - B^*p) = Bv \tag{6.11}$$

where $v$ is given by 6.8.

If one uses the generalized minimal residual method (GMRES) the method is directly applied to the preconditioned system

$$\hat{S}^{-1}Sp = \hat{S}^{-1}BA^{-1}G \tag{6.12}$$

We use the norm

$$\|p\|_{GMRES} = \|\hat{S}p\| \tag{6.13}$$

Notice that for the residual $\hat{r} = \hat{S}^{-1}r$ one has

(6.14)

If $p^0$ provides an initial guess for the pressure we use 6.8 to get a first initial guess for the velocity $v^0$ which we use to set an absolute tolerance $ATOL = \epsilon\|\sqrt{v^0_{j,k}v^0_{j,k}}\|$. The GMRES is terminated when

$$\|\hat{r}\|_{GMRES} \leq ATOL \tag{6.15}$$

Notice that $\|\hat{r}\|_{GMRES} = \|r\| = \|Bv\| = \|v_{k,k}\|$ so we we can expect that the target stopping criterion 6.6 is fullfilled. However, if $v$ is very different from the initial choice of $v^0$ the value of $ATOL$ is corrected and GMRES is restarted with a new tolerance. For time dependend problems this apprach works well as value for $p$ form a previous time step provides a good initial guess.

Alternatively, as $S$ is symmetric and positive definite one can apply the preconditioned conjugate gradient method (PCG) . PCG use the norm

$$\|r\|^2_{PCG} = \int_\Omega r\hat{S}^{-1}r \; dx = \int_\Omega \eta r^2 \; dx \tag{6.16}$$

To take the extra factor $\eta$ into consideration when checking the stopping criterion we use the following definition for $ATOL$:

$$ATOL = \epsilon\frac{\|\sqrt{v^0_{j,k}v^0_{j,k}}\|}{\|v^0_{k,k}\|}\|v^0_{k,k}\|_{PCG} \tag{6.17}$$

### 6.1.2 Functions

**class `StokesProblemCartesian`** (*domain*[, *adaptSubTolerance*=`True`])
opens the Stokes problem on the `Domain` domain. The approximation order needs to be two. If *adaptSubTolerance* is `True` the tolerances for all subproblems are set automatically.

**`initialize`** ([*f=Data()*, [*fixed_u_mask=Data()*, [*eta=1*, [*surface_stress=Data()*, [*stress=Data()*]]]]])
assigns values to the model parameters. In any call all values must be set. $f$ defines the external force $f$, *eta* the viscosity $\eta$, *surface_stress* the surface stress $s$ and *stress* the initial stress $\sigma$. The locations and compontents where the velocity is fixed are set by the values of *fixed_u_mask*. The method will try to cast the given values to appropriate `Data` class objects.

**`solve`** (*v,p* [, *max_iter=100* [, *verbose=False* [, *usePCG=True* ]]])
solves the problem and return approximations for velocity and pressure. The arguments *v* and *p* define initial guess. The values of *v* marked by *fixed_u_mask* remain unchanged. If *usePCG* is set to `True` reconditioned conjugate gradient method (PCG) scheme is used. Otherwise the problem is solved generalized minimal residual method (GMRES) . In most cases the PCG scheme is more efficient. *max_iter* defines the maximum number of iteration steps.

If *verbose* is set to `True` informations on the progress of of the solver are printed.

**setTolerance**([*tolerance=1.e-4*])
sets the tolerance in an appropriate norm relative to the right hand side. The tolerance must be non-negative and less than 1.

**getTolerance**()
returns the current relative tolerance.

**setAbsoluteTolerance**([*tolerance=0.*])
sets the absolute tolerance for the error in the relevant norm. The tolerance must be non-negative. Typically the absolute talerance is set to 0.

**getAbsoluteTolerance**()
sreturns the current absolute tolerance.

**getSolverOptionsVelocity**()
returns the solver options used solve the equations (6.8) for velocity.

**getSolverOptionsPressure**()
returns the solver options used solve the equation (6.10) for pressure.

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

### 6.1.3 Example: Lit Driven Cavity

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

```
from esys.escript import *
from esys.finley import Rectangle
from esys.escript.models import StokesProblemCartesian
NE=25
dom = Rectangle(NE,NE,order=2)
x = dom.getX()
sc=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]
sc.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=sc.solve(v,p, verbose=True)
saveVTK("u.xml",velocity=v,pressure=p)
```

## 6.2 Darcy Flux

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

$$\begin{aligned} u_i + \kappa_{ij} p_{,j} &= g_i \\ u_{k,k} &= f \end{aligned} \tag{6.18}$$

with the boundary conditions

$$\begin{aligned} u_i\, n_i = u_i^N\, n_i \quad &\text{on} \quad \Gamma_N \\ p = p^D \quad &\text{on} \quad \Gamma_D \end{aligned} \tag{6.19}$$

where $\Gamma_N$ and $\Gamma_D$ are a partition of the boundary of $\Omega$ with $\Gamma_D$ non empty, $n_i$ is the outer normal field of the boundary of $\Omega$, $u_i^N$ and $p^D$ are given functions on $\Omega$, $g_i$ and $f$ are given source terms and $\kappa_{ij}$ is the given permability. We assume that $\kappa_{ij}$ is symmetric (which is not really required) and positive definite, i.e there are positive constants $\alpha_0$ and $\alpha_1$ wich are independent from the location in $\Omega$ such that

$$\alpha_0\, x_i x_i \leq \kappa_{ij} x_i x_j \leq \alpha_1\, x_i x_i \tag{6.20}$$

for all $x_i$.

### 6.2.1 Solution Method

In practical applications it is an advantage to calculate the pressure $p$ as a correction of a 'static' pressure $p^{ref}$ which is the solution of

$$-(\kappa_{ki}\kappa_{kj}p^{ref}_{,j})_{,i} = -(\kappa_{ki}(g_k - u^N_k))_{,i} \text{ with } p^{ref} = p^D \text{ on } \Gamma_D \tag{6.21}$$

With setting $u \leftarrow u - u^N$ and $p \leftarrow p - p^{ref}$ and

$$\begin{array}{rcl} g_i & \leftarrow & g_i - u^N_i - \kappa_{ij}p^{ref}_{,j} \\ f & \leftarrow & f - u^N_{k,k} \end{array} \tag{6.22}$$

we can assume that $u^N_i\ n_i = 0$ and $p^D = 0$. We set

$$V = \{q \in H^1(\Omega) : q = 0 \text{ on } \Gamma_D\} \tag{6.23}$$

and

$$W = \{v \in (L^2(\Omega))^d : v_{k,k} \in L^2(\Omega) \text{ and } u_i\ n_i = 0 \text{ on } \Gamma_N\} \tag{6.24}$$

and define the operator $Q : V \to (L^2(\Omega))^d$ defined by

$$(Qp)_i = \kappa_{ij}p_{,j} \tag{6.25}$$

and the operator $D : W \to L^2(\Omega)$ defined by

$$Dv = v_{k,k} \tag{6.26}$$

In operator notation the Darcy problem 6.18 is written in the form

$$\begin{array}{rcl} u + Qp & = & g \\ Du & = & f \end{array} \tag{6.27}$$

We solve this equation by minimising the functional

$$J(u,p) := \|u + Qp - g\|^2_0 + \|Du - f\|^2_0 \tag{6.28}$$

over $W \times V$ where $\|.\|_0$ denotes the norm in $L^2(\Omega)$. A simple calculation shows that one has to solve

$$(v + Qq, u + Qp - g) + (Dv, Du - f) = 0 \tag{6.29}$$

for all $v \in W$ and $q \in V$.which translates back into operator notation

$$\begin{array}{rcl} (I + D^*D)u + Qp & = & D^*f + g \\ Q^*u + Q^*Qp & = & Q^*g \end{array} \tag{6.30}$$

where $D^*$ and $Q^*$ denote the adjoint operators. In [19] it has been shown that this problem is continuous and coercive in $W \times V$ and therefore has a unique solution. Also standart FEM methods can be used for discretization. It is also possible to solve the problem is coupled form, however this approach leads in some cases to a very ill-conditioned stiffness matrix in particular in the case of a very small or large permability ($\alpha_1 \ll 1$ or $\alpha_0 \gg 1$)

The approach we are taking is to eliminate the velocity $u$ from the problem. Assuming that $p$ is known we have

$$v = (I + D^*D)^{-1}(D^*f + g - Qp) \tag{6.31}$$

(notice that $(I + D^*D)$ is coercive in $W$) which is inserted into the second equation

$$Q^*(I + D^*D)^{-1}(D^*f + g - Qp) + Q^*Qp = Q^*g \tag{6.32}$$

which is

$$Q^*(I - (I + D^*D)^{-1})Qp = Q^*(g - (I + D^*D)^{-1}(D^*f + g)) \tag{6.33}$$

We use the PCG method to solve this. The residual $r$ ($\in V^*$) is given as

$$\begin{array}{rcl} r & = & Q^*\left(g - (I + D^*D)^{-1}(D^*f + g) - Qp + (I + D^*D)^{-1}Qp\right) \\ & = & Q^*\left(g - Qp - (I + D^*D)^{-1}(D^*f + g - Qp)\right) \\ & = & Q^*\left(g - Qp - v\right) \end{array} \tag{6.34}$$

So in a partical implementation we use $\hat{r} = g - Qp - v$ to represent the residual. The evaluation of the iteration operator for a given $p$ is then returning $Qp + v$ where $v$ is the solution of

$$(I + D^*D)v = Qp \tag{6.35}$$

We use $(Q^*Q)^{-1}$ as a preconditioner for the iteration operator $Q^*(I - (I + D^*D)^{-1})Q$. So the application of the preconditioner to $\hat{r}$ representing the residual is given by solving implemented by solving

$$Q^*Qq = Q^*\hat{r} \tag{6.36}$$

The residual norm used in the PCG is given as

$$\|r\|_{PCG}^2 = \int r \cdot (Q^*Q)^{-1} r \, dx = \int \hat{r} \cdot Q(Q^*Q)^{-1}Q^*\hat{r} \, dx \approx \|\hat{r}\|_0^2 \tag{6.37}$$

The iteration is terminated if

$$\|r\|_{PCG} \leq \text{ATOL} \tag{6.38}$$

where we set

$$\text{ATOL} = \text{atol} + \text{rtol} \cdot \left( \frac{1}{\|v\|_0} + \frac{1}{\|Qp\|_0} \right)^{-1} \tag{6.39}$$

where rtol is a given relative tolerance and atol is a given absolute tolerance (typically $= 0$). Notice that if $Qp$ and $v$ both are zero, the pair $(0, p)$ is a solution. The problem is that ATOL is depending on the solution $p$ (and $v$ calculated form 6.31). In partcice one use the initial guess for $p$ to get a first value for ATOL. If the stopping crierion is met in the PCG iteration, a new $v$ is calculated from the current pressure approximation and ATOL is recalculated. If 6.38 is still fullfilled the calculation is terminated and $(v, p)$ is returned. Otherwise PCG is restarted with a new ATOL.

### 6.2.2 Functions

**class `DarcyFlow`** (*domain* [, *adaptSubTolerance=`True`*])
opens the Darcy flux problem on the `Domain` domain. If *adaptSubTolerance* is set to `True`, the relative tolerances for solving (6.31), (6.35) and (6.36) are set automatically.

**`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 dependend problem only values that change over time needs to be reset. The permability can be defined as scalar (isotropic), a vector (orthotropic) or a matrix (anisotropic). *f* and *g* are the corresponding parameters in 6.18. The locations and compontents 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 6.39.

**`setAbsoluteTolerance`** ([*atol=0.*])
sets the absolute tolerance atol in 6.39.

**`getSolverOptionsFlux`** ()
Returns the solver options used to solve the flux problems (6.31) and (6.35). Use the returned `SolverOptions` object to control the solution algorithms. If the adaption of subtolerance is choosen, the tolerance will be overwritten before the solver is called.

**`getSolverOptionsPressure`** ()
Returns the solver options used to solve the pressure problems (6.36). Use the returned `SolverOptions` object to control the solution algorithms. If the adaption of subtolerance is choosen, the tolerance will be overwritten before the solver is called.

**solve** (*u0,p0,* [*max_iter=100,* [*verbose=False* ] ])
solves the problem. and returns approximations for the flux $v$ and the pressure $p$. *u0* and *p0* define initial guess for flux and pressure. Values marked by positive values *location_of_fixed_flux* and *location_of_fixed_pressure*, respectively, are kept unchanged. *max_iter* sets the maximum number of iterations steps allowed for solving the coupled problem.

### 6.2.3 Example: Gravity Flow

later

## 6.3 Isotropic Kelvin Material

As proposed by Kelvin [15] material strain $D_{ij} = \frac{1}{2}(v_{i,j} + v_{j,i})$ can be decomposed into an elastic part $D^{el}_{ij}$ and visco-plastic part $D^{vp}_{ij}$:

$$D_{ij} = D^{el}_{ij} + D^{vp}_{ij} \tag{6.40}$$

with the elastic strain given as

$$D^{el'}_{ij} = \frac{1}{2\mu}\dot{\sigma}'_{ij} \tag{6.41}$$

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

$$D^{vp'}_{ij} = \sum_q D^{q'}_{ij} \tag{6.42}$$

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

$$D^{q'}_{ij} = \frac{1}{2\eta^q}\sigma'_{ij} \tag{6.43}$$

where $\eta^q$ is the viscosity of material $q$. We assume the following betwee the the strain in material $q$

$$\eta^q = \eta^q_N \left(\frac{\tau}{\tau^q_t}\right)^{1-n^q} \text{ with } \tau = \sqrt{\frac{1}{2}\sigma'_{ij}\sigma'_{ij}} \tag{6.44}$$

for a given power law coefficients $n^q \geq 1$ and transition stresses $\tau^q_t$, see [15]. Notice that $n^q = 1$ gives a constant viscosity. After inserting equation 6.43 into equation 6.42 one gets:

$$D'^{vp}_{ij} = \frac{1}{2\eta^{vp}}\sigma'_{ij} \text{ with } \frac{1}{\eta^{vp}} = \sum_q \frac{1}{\eta^q} \ . \tag{6.45}$$

and finally with 6.40

$$D'_{ij} = \frac{1}{2\eta^{vp}}\sigma'_{ij} + \frac{1}{2\mu}\dot{\sigma}'_{ij} \tag{6.46}$$

The total stress $\tau$ needs to fullfill the yield condition

$$\tau \leq \tau_Y + \beta\, p \tag{6.47}$$

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

$$-\sigma'_{ij,j} + p_{,i} = F_i \tag{6.48}$$

where $F_j$ is a given external fource. We assume an incompressible media:

$$-v_{i,i} = 0 \tag{6.49}$$

Natural boundary conditions are taken in the form

$$\sigma'_{ij}n_j - n_i p = f \tag{6.50}$$

which can be overwritten by a constraint

$$v_i(x) = 0 \tag{6.51}$$

where the index $i$ may depend on the location $x$ on the bondary.

### 6.3.1 Solution Method

By using a first order finite difference approximation wit step size $dt > 0$ 6.41 get the form

$$\dot{\sigma}_{ij} = \frac{1}{dt}\left(\sigma_{ij} - \sigma_{ij}^{-}\right) \tag{6.52}$$

and

$$D'_{ij} = \left(\frac{1}{2\eta^{vp}} + \frac{1}{2\mu dt}\right)\sigma'_{ij} - \frac{1}{2\mu dt}\sigma_{ij}^{-'} \tag{6.53}$$

where $\sigma_{ij}^{-}$ is the stress at the precious time step. With

$$\dot{\gamma} = \sqrt{2\left(D'_{ij} + \frac{1}{2\mu\ dt}\sigma_{ij}^{-'}\right)^2} \tag{6.54}$$

we have

$$\tau = \eta_{eff} \cdot \dot{\gamma} \tag{6.55}$$

where

$$\eta_{eff} = min(\left(\frac{1}{\mu\ dt} + \frac{1}{\eta^{vp}}\right)^{-1}, \eta_{max}) \text{ with } \eta_{max} = \begin{cases} \frac{\tau_Y + \beta\ p}{\dot{\gamma}} & \dot{\gamma} > 0 \\ \text{if} & \\ \infty & \text{otherwise} \end{cases} \tag{6.56}$$

The upper bound $\eta_{max}$ makes sure that yield condtion 6.47 holds. With this setting the eqaution 6.53 takes the form

$$\sigma'_{ij} = 2\eta_{eff}\left(D'_{ij} + \frac{1}{2\mu\ dt}\sigma_{ij}^{'-}\right) \tag{6.57}$$

After inserting 6.57 into 6.48 we get

$$-\left(\eta_{eff}(v_{i,j} + v_{i,j})\right)_{,j} + p_{,i} = F_i + \left(\frac{\eta_{eff}}{\mu dt}\sigma_{ij}^{'-}\right)_{,j} \tag{6.58}$$

Combining this with the incomressibilty condition 6.40 we need to solve a Stokes problem as discussed in section 6.1.1 in each time step.

If we set

$$\frac{1}{\eta(\tau)} = \frac{1}{\mu\ dt} + \frac{1}{\eta^{vp}} \tag{6.59}$$

we need to solve the nonlinear problem

$$\eta_{eff} - min(\eta(\dot{\gamma} \cdot \eta_{eff}), \eta_{max}) = 0 \tag{6.60}$$

We use the Newton-Raphson Scheme to solve this problem

$$\eta_{eff}^{(n+1)} = \min(\eta_{max}, \eta_{eff}^{(n)} - \frac{\eta_{eff}^{(n)} - \eta(\tau^{(n)})}{1 - \dot{\gamma} \cdot \eta'(\tau^{(n)})}) = \min(\eta_{max}, \frac{\eta(\tau^{(n)}) - \tau^{(n)} \cdot \eta'(\tau^{(n)})}{1 - \dot{\gamma} \cdot \eta'(\tau^{(n)})}) \tag{6.61}$$

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

Looking at the evaluation of $\eta$ in 6.59 it makes sense formulate the iteration 6.61 using $\Theta = \eta^{-1}$. In fact we have

$$\eta' = -\frac{\Theta'}{\Theta^2} \text{ with } \Theta' = \sum_q \left(\frac{1}{\eta^q}\right)' \tag{6.62}$$

As

$$\left(\frac{1}{\eta^q}\right)' = \frac{n^q - 1}{\eta_N^q} \cdot \frac{\tau^{n^q - 2}}{(\tau_t^q)^{n^q - 1}} = \frac{n^q - 1}{\eta^q} \cdot \frac{1}{\tau} \tag{6.63}$$

we have

$$\Theta' = \frac{1}{\tau}\omega \text{ with } \omega = \sum_q \frac{n^q - 1}{\eta^q} \tag{6.64}$$

which leads to

$$\eta_{eff}^{(n+1)} = \min(\eta_{max}, \eta_{eff}^{(n)} \frac{\Theta^{(n)} + \omega^{(n)}}{\eta_{eff}^{(n)}\Theta^{(n)^2} + \omega^{(n)}}) \tag{6.65}$$

### 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 cooridninates on the domain *domain*. *stress*, *v*, *p*, and *t* set the initial deviatoric stress, velocity, pressure and time. *numMaterials* specifies the number of materials used in the power law model. Some progress information are printed if *verbose* is set to `True`. If *adaptSubTolerance* is equal to True the tolerances for subproblems are set automatically.

**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 effectice viscosity. Iteration on a time step is completed if the realtive of the effective viscosity is less than *rtol*.

**setDruckerPragerLaw**([*tau_Y=None,* [*friction=None* ]])
Sets the parameters $\tau_Y$ and $\beta$ for the Drucker-Prager model in condition 6.47. If *tau_Y* is set to None (default) Drucker-Prager condition is not applied.

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

**setPowerLaws**(*eta_N, tau_t, power*)
Sets the parameters of the power-law for all materials as defined in equation 6.44. *eta_N* is the list of viscosities $\eta_N^q$, *tau_t* is the list of reference stresses $\tau_t^q$, and *power* is the list of power law coefficients $n^q$.

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

### 6.3.3 Example

later

CHAPTER

**SEVEN**

# The Module `esys.finley`

*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 `esys.finley` provides an access to the library through the `LinearPDE` class of `esys.escript` supporting its full functionality. *finley* is parallelized using the OpenMP paradigm.

## 7.1 Formulation

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

$$\begin{aligned} & \int_\Omega A_{jl} \cdot v_{,j} u_{,l} + B_j \cdot v_{,j} u + C_l \cdot v u_{,l} + D \cdot v u \, d\Omega \\ + & \int_\Gamma d \cdot v u \, d\Gamma + \int_{\Gamma^{contact}} d^{contact} \cdot [v][u] \, d\Gamma \\ = & \int_\Omega X_j \cdot v_{,j} + Y \cdot v \, d\Omega \\ + & \int_\Gamma y \cdot v \, d\Gamma + \int_{\Gamma^{contact}} y^{contact} \cdot [v] \, d\Gamma \end{aligned} \tag{7.1}$$

## 7.2 Meshes

To understand the usage of `esys.finley` one needs to have an understanding of how the finite element meshes are defined. Figure 7.1 shows an example of the subdivision of an ellipse into so called elements . In this case, triangles have been used but other forms of subdivisions can be constructed, e.g. into quadrilaterals or, in the three dimensional case, into tetrahedrons and hexahedrons. The idea of the finite element method is to approximate the solution by a function which is a polynomial of a certain order and is continuous across it boundary to neighbor elements. In the example of Figure 7.1 a linear polynomial is used on each triangle. As one can see, the triangulation is quite a poor approximation of the ellipse. It can be improved by introducing a midpoint on each element edge then positioning those nodes located on an edge expected to describe the boundary, onto the boundary. In this case the triangle gets a curved edge which requires a parametrization 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 [28, 5] for more details.

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 reference number 20 which has type *Line2* is defined by the nodes with the reference numbers 11 and 0. Again the order is crucial, if moving from the first to second node the domain has to lie on the left hand side (in the case of a two dimension surface element the domain has to lie on the left hand side when moving counterclockwise). If the gradient on the surface of the domain is to be calculated rich face elements face to be used. Rich elements on a face are identical to interior elements but with a modified order

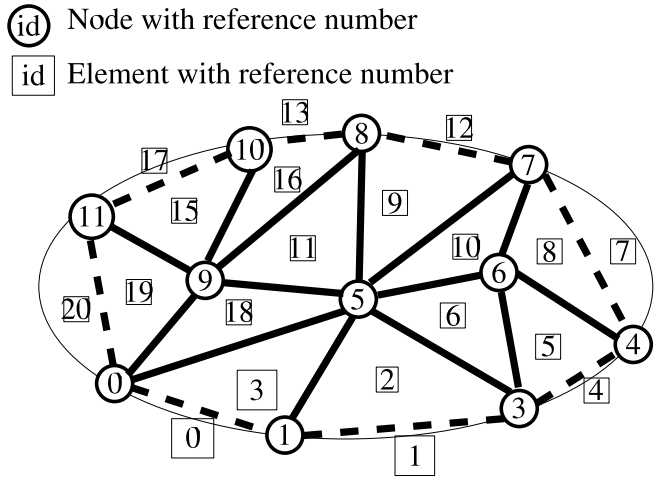



FIGURE 7.1: Subdivision of an Ellipse into triangles order 1 (*Tri3*)

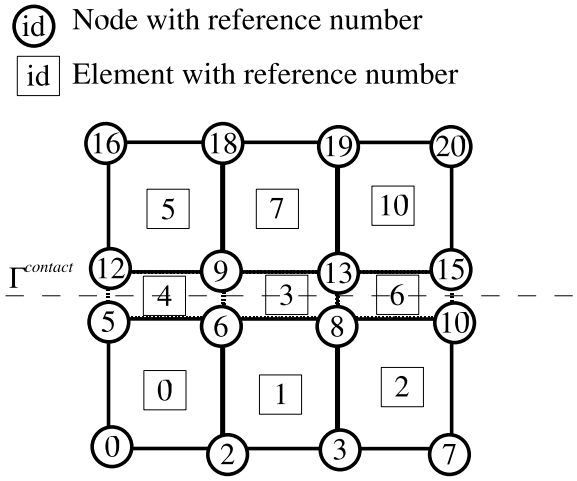



FIGURE 7.2: Mesh around a contact region (*Rec4*)

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 $\Gamma^{contact}$ even if $d^{contact}$ and $y^{contact}$ are zero. Figure 7.2 shows a simple example of a mesh of rectangular elements around a contact region $\Gamma^{contact}$ . The contact region is described by the elements 4, 3 and 6. Their element type is *Line2_Contact*. The nodes 9, 12, 6, 5 define contact element 4, where the coordinates of nodes 12 and 5 and nodes 4 and 6 are identical with the idea that nodes 12 and 9 are located above and nodes 5 and 6 below the contact region. Again, the order of the nodes within an element is crucial. There is also the option of

| interior | face | rich face | contact | rich contact |
|---|---|---|---|---|
| *Line2* | *Point1* | *Line2Face* | *Point1_Contact* | *Line2Face_Contact* |
| *Line3* | *Point1* | *Line3Face* | *Point1_Contact* | *Line3Face_Contact* |
| *Tri3* | *Line2* | *Tri3Face* | *Line2_Contact* | *Tri3Face_Contact* |
| *Tri6* | *Line3* | *Tri6Face* | *Line3_Contact* | *Tri6Face_Contact* |
| *Rec4* | *Line2* | *Rec4Face* | *Line2_Contact* | *Rec4Face_Contact* |
| *Rec8* | *Line3* | *Rec8Face* | *Line3_Contact* | *Rec8Face_Contact* |
| *Rec9* | *Line3* | *Rec9Face* | *Line3_Contact* | *Rec9Face_Contact* |
| *Tet4* | *Tri6* | *Tet4Face* | *Tri6_Contact* | *Tet4Face_Contact* |
| *Tet10* | *Tri9* | *Tet10Face* | *Tri9_Contact* | *Tet10Face_Contact* |
| *Hex8* | *Rec4* | *Hex8Face* | *Rec4_Contact* | *Hex8Face_Contact* |
| *Hex20* | *Rec8* | *Hex20Face* | *Rec8_Contact* | *Hex20Face_Contact* |

Table 7.1: Finley elements and corresponding elements to be used on domain faces and contacts. The rich types have to be used if the gradient of function is to be calculated on faces and contacts, respectively.

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 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 is crucial. It 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 2 and elements below a contact region are marked with 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:

```
print "%s\n"%mesh_name
# node coordinates:
print "%dD-nodes %d\n"%(dim,numNodes)
for i in range(numNodes):
   print "%d %d %d"%(Node_ref[i],Node_DOF[i],Node_tag[i])
   for j in range(dim): print " %e"%Node[i][j]
   print "\n"
# interior elements
print "%s %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"
# point sources (not supported yet)
```

```
write("Point1 0",face_element_type,numFaceElements)
```

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
```

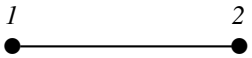
Notice that the order in which the nodes and elements are given is arbitrary. In the case that rich contact elements are used the contact element section gets the form

```
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
```

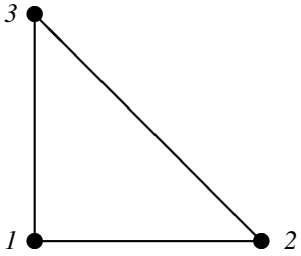
Periodic boundary condition can be introduced by altering *Node_DOF*. It allows identification of nodes even if they have different physical locations. For instance, to enforce periodic boundary conditions at the face $x0 = 0$ and $x0 = 1$ one identifies the degrees of freedom for nodes 0, 5, 12 and 16 with the degrees of freedom for 7, 10, 15 and 20, respectively. The node section of the `esys.finley` mesh gets now the form:

```
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.   1.0
18 18 0 0.33 1.0
19 19 0 0.66 1.0
20 16 0 1.0  1.0
```

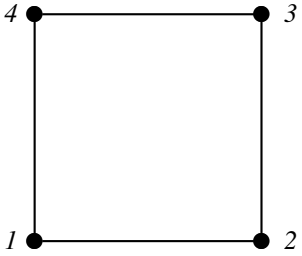

Point1



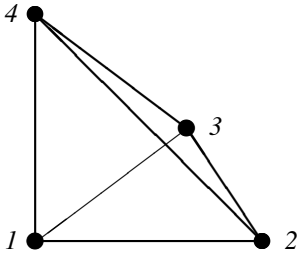

LINE2



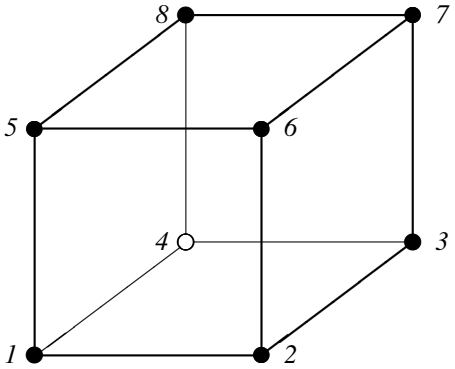

Tri3




Rec4




Tet4




Hex8

FIGURE 7.3: Elements of order 1

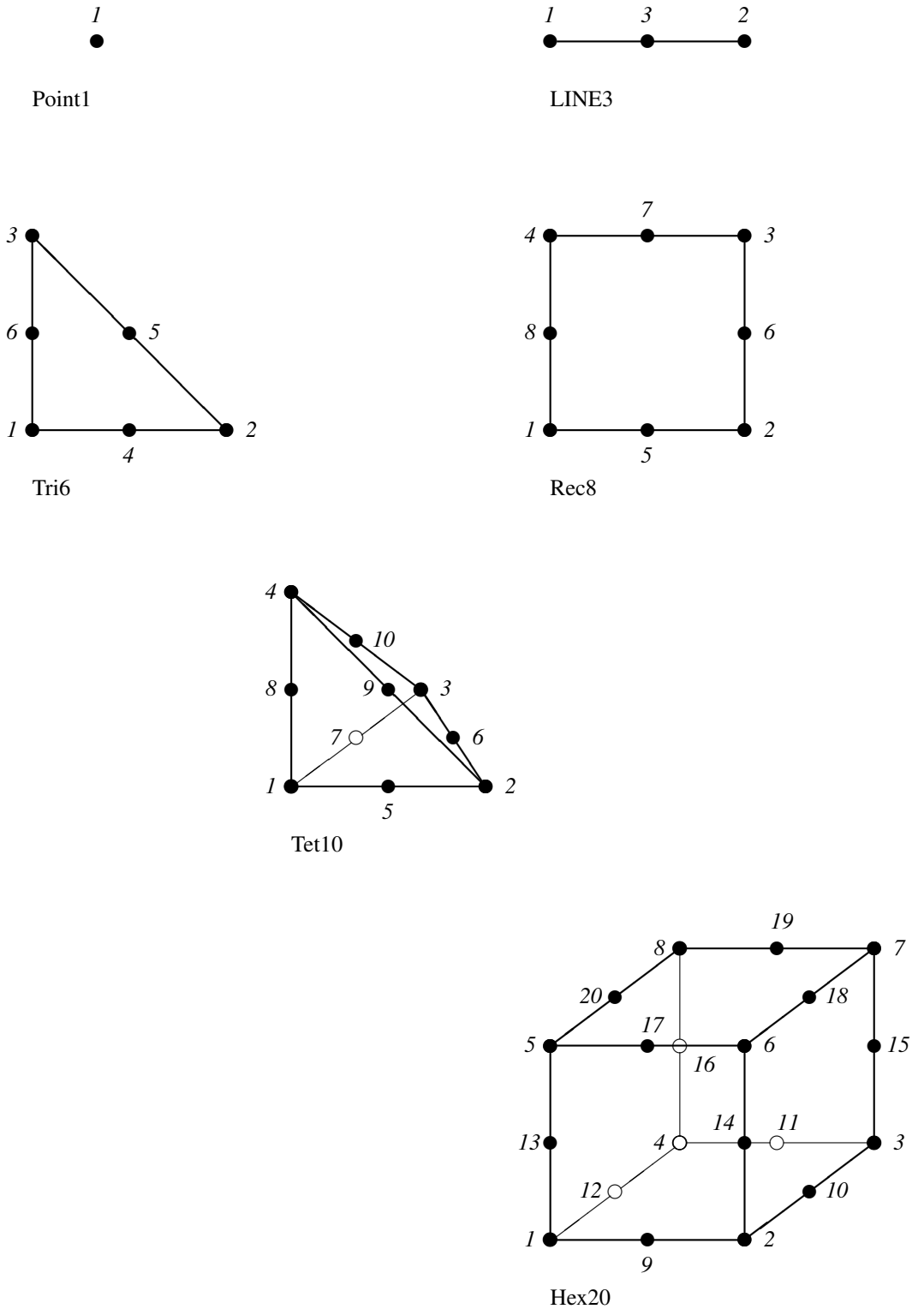



FIGURE 7.4: Elements of order 2

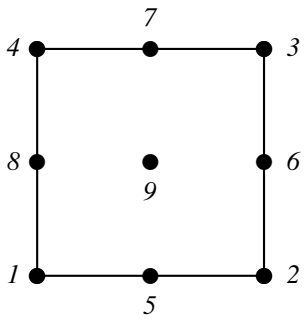



FIGURE 7.5: Additional shape functions

| setSolverMethod | DIRECT | PCG | GMRES | TFQMR | MINRES | PRES20 | BICGSTAB | LUMPING |
|---|---|---|---|---|---|---|---|---|
| setReordering | ✓ | | | | | | | |
| setRestart | | | ✓ | | | 20 | | |
| setTruncation | | | ✓ | | | 5 | | |
| setIterMax | | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | |
| setTolerance | | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | |
| setAbsoluteTolerance | | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | |
| setReordering | ✓ | | | | | | | |

Table 7.2: Solvers available for esys.finley and the PASO package and the relevant options in SolverOptions. MKL supports SolverOptions.MINIMUM_FILL_IN and SolverOptions.NESTED_DISSECTION reordering. Currently the UMFPACK interface does not support any reordering.

| setPreconditioner | NO_PRECONDITIONER | AMG | JACOBI | GAUSS_SEIDEL | REC_ILU | RILU | ILU0 | DIRECT |
|---|---|---|---|---|---|---|---|---|
| status: | later | later | ✓ | ✓ | ✓ | later | ✓ | later |
| setCoarsening | | ✓ | | | | | | |
| setLevelMax | | ✓ | | | | | | |
| setCoarseningThreshold | | ✓ | | | | | | |
| setMinCoarseMatrixSize | | ✓ | | | | | | |
| setNumSweeps | | | ✓ | ✓ | | | | |
| setNumPreSweeps | | ✓ | | | | | | |
| setNumPostSweeps | | ✓ | | | | | | |
| setInnerTolerance | | | | | | | | |
| setDropTolerance | | | | | | | | |
| setDropStorage | | | | | | | | |
| setRelaxationFactor | | | | | | ✓ | | |
| adaptInnerTolerance | | | | | | | | |
| setInnerIterMax | | | | | | | | |

Table 7.3: Preconditioners available for esys.finley and the PASO package and the relevant options in SolverOptions.

### 7.2.1 Linear Solvers in SolverOptions

Table 7.2 and Table 7.3 show the solvers and preconditioners supported by esys.finley through the PASO library. Currently direct solvers are not supported under MPI. By default, esys.finley is using the iterative solvers SolverOptions.PCG for symmetric and SolverOptions.BICGSTAB for non-symmetric problems. If the direct solver is selected which can be useful when solving very ill-posedequations 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 the PASO is used.

### 7.2.2 Functions

**ReadMesh** (*fileName,integrationOrder=-1*)
creates a Domain object form the FEM mesh defined in file *fileName*. The file must be given the esys.finley file format. If *integrationOrder* is positive, a numerical integration scheme chosen which is accurate on each element up to a polynomial of degree *integrationOrder* . Otherwise an appropriate integration order is chosen independently.

**load** (*fileName*)
recovers a Domain object from a dump file created by the eateseates a Domain object form the FEM mesh defined in file *fileName*. The file must be given the esys.finley file format. If *integrationOrder* is positive, a numerical integration scheme chosen which is accurate on each element up to a polynomial of degree *integrationOrder* . Otherwise an appropriate integration order is chosen independently.

**Rectangle** (*n0,n1,order=1,l0=1.,l1=1., integrationOrder=-1,*
*periodic0=False,periodic1=False,useElementsOnFace=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 *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. 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 chosen which is accurate on each element up to a polynomial of degree *integrationOrder* . Otherwise an appropriate integration order is chosen independently. If *periodic0*=`True`, periodic boundary conditions along the $x0$-directions are enforced. That means when for any solution of a PDE solved by `esys.finley` the value on the line $x0 = 0$ will be identical to the values on $x0 = l0$. Correspondingly, *periodic1*=`False` sets periodic boundary conditions in $x1$-direction. If *optimize*=`True` mesh node relabeling will be attempted to reduce the computation and also ParMETIS will be used to improve the mesh partition if running on multiple CPUs with MPI.

**`Brick`**(*n0,n1,n2,order=1,l0=1.,l1=1.,l2=1., integrationOrder=-1,*
*periodic0=`False`,periodic1=`False`,periodic2=`False`,useElementsOnFace=`False`,optimize=`False`*)
Generates a `Domain` object representing a three dimensional brick between $(0, 0, 0)$ and $(l0, l1, l2)$ with orthogonal faces. The brick is filled with *n0* elements along the $x0$-axis, *n1* elements along the $x1$-axis and *n2* elements along the $x2$-axis. For *order*=1 and *order*=2 *Hex8* and *Hex20* are used, respectively. In the case of *useElementsOnFace*=`False`, *Rec4* and *Rec8* are used to subdivide the faces of the brick, respectively. In the case of *useElementsOnFace*=`True` (this option should be used if gradients are calculated on domain faces), *Hex8Face* and *Hex20Face* are used on the brick faces, respectively. If *integrationOrder* is positive, a numerical integration scheme chosen which is accurate on each element up to a polynomial of degree *integrationOrder* . Otherwise an appropriate integration order is chosen independently. If *periodic0*=`True`, periodic boundary conditions along the $x0$-directions are enforced. That means when for any solution of a PDE solved by `esys.finley` the value on the plane $x0 = 0$ will be identical to the values on $x0 = l0$. Correspondingly, *periodic1*=`False` and *periodic2*=`False` sets periodic boundary conditions in $x1$-direction and $x2$-direction, respectively. If *optimize*=`True` mesh node relabeling will be attempted to reduce the computation and also ParMETIS will be used to improve the mesh partition if running on multiple CPUs with MPI.

**`GlueFaces`**(*meshList,safetyFactor=0.2,tolerance=1.e-13*)
Generates a new `Domain` object from the list *meshList* of `esys.finley` meshes. Nodes in face elements whose difference of coordinates is less then *tolerance* times the diameter of the domain are merged. The corresponding face elements are removed from the mesh.

TODO: explain *safetyFactor* and show an example.

**`JoinFaces`**(*meshList,safetyFactor=0.2,tolerance=1.e-13*)
Generates a new `Domain` object from the list *meshList* of `esys.finley` meshes. Face elements whose nodes coordinates have difference is less then *tolerance* times the diameter of the domain are combined to form a contact element The corresponding face elements are removed from the mesh.

TODO: explain *safetyFactor* and show an example.

APPENDIX

**A**

# Misc

## A.1 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 involing tensors more compact, by shortening and simplifying them.

There are two rules which make up the convention:

firstly, the rank of the tensor is represented by an index. For example, $a$ is a scalar; $b_i$ represents a vector; and $c_{ij}$ represents a matrix.

Secondly, if an expression contains subscripted variables, they are assumed to be summed over all possible values, from $0$ to $n$. For example, for the following expression:

$$y = a_0 b_0 + a_1 b_1 + \ldots + a_n b_n \tag{A.1}$$

can be represented as:

$$y = \sum_{i=0}^{n} a_i b_i \tag{A.2}$$

then in Einstein notion:

$$y = a_i b_i \tag{A.3}$$

Another example:

$$\nabla p = \frac{\partial p}{\partial x_0}\mathbf{i} + \frac{\partial p}{\partial x_1}\mathbf{j} + \frac{\partial p}{\partial x_2}\mathbf{k} \tag{A.4}$$

can be expressed in Einstein notation as:

$$\nabla p = p_{,i} \tag{A.5}$$

where the comma ',' indicates the partial derivative.

For a tensor:

$$\sigma_{ij} = \begin{bmatrix} \sigma_{00} & \sigma_{01} & \sigma_{02} \\ \sigma_{10} & \sigma_{11} & \sigma_{12} \\ \sigma_{20} & \sigma_{21} & \sigma_{22} \end{bmatrix} \tag{A.6}$$

The $\delta_{ij}$ is the Kronecker $\delta$-symbol, which is a matrix with ones for its diagonal entries ($i = j$) and zeros for the remaining entries ($i \neq j$).

$$\delta_{ij} = \begin{cases} 1, & \text{if } i = j \\ 0, & \text{if } i \neq j \end{cases} \tag{A.7}$$

## A.2 Changes from previous releases

### 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 matricies or higher ranked data, the tuples will be nested. Use `numpy`.array(data.getTupleForDataPoint()) if a `numpy.ndarray` object is required.
  - getValueOfGlobalDataPoint has similarly been replaced by getTupleForGlobalDataPoint().
  - integrate(data) now returns a `numpy.ndarray` instead of a `numarray`.array.

  Any python methods which previously accepted `numarray` objects will accept `numpy` objects instead.
- The way solver options are defined for `LinearPDE` objects has been changed. There is now a `SolverOptions` object attached to the `LinearPDE` object which is handeling the options of solvers used to solve the PDE. The following changes apply:
  - The `setTolerance` and `setAbsoluteTolerance` methods have been removed. Use now `getSolverOptions().setTolerance` and `getSolverOptions().setAbsoluteTolerance`
  - The `setSolverPackage` and `setSolverMethod` methods have been removed. Use now `getSolverOptions().setPackage`, `getSolverOptions().setSolverMethod` and `getSolverOptions().setPreconditioner`.
  - The `setSolverPackage` and `setSolverMethod` methods have been removed. Use now `getSolverOptions().setPackage`, `getSolverOptions().setSolverMethod` and `getSolverOptions().setPreconditioner`.
  - The static class variables defining packages, solvers and preconditioners have been removed and are now accessed via the corresponding static class variables in `SolverOptions`. For instance use `SolverOptions.PCG` instead of `LinearPDE.PCG` to select the preconditioned conjugate gradient method.
  - The `getSolution` takes now no argument. Use the corresponding methods of the `SolverOptions` object returned by `getSolverOptions()` to set values, e.g. use `getSolverOptions().setVerbosityOn()` instead of argument `verbose=True` and `getSolverOptions().setIterMax(1000)` instead of argument `iter_max=1000`
- The `esys.pyvisi` module from previous releases has been deprecated and will no longer be supported. It is still present in the source code and can still be used if you compile `esys.escript` from source. It will not be available in binary releases. Its use is discouraged. The documentation for `esys.pyvisi` can be found in Appendix B.

## A.3 escript references

If you use escript in your research we would appreciate a citation (of course we do not require this). Possible references include:

```
@article{GROSS2006,
        author = {L. Gross and L. Bourgouin and A. J. Hale and H.-B Muhlhaus},
        title = {Interface Modeling in Incompressible Media
        using Level Sets in Escript},
        journal = {Physics of the Earth and Planetary Interiors},
        year = 2007,
        volume = {163},
        pages = {23--34},
        month = {Aug.},
        doi = {doi:10.1016/j.pepi.2007.04.004},
}
```

```
@article{GROSS2007,
        author = {L. Gross and B. Cumming and K. Steube and D. Weatherley},
        title = {A Python Module for PDE-Based Numerical Modelling},
        journal = {PARA},
        year = {2007},
        volume = {4699},
        pages = {270--279},
        doi = {doi:10.1007/978-3-540-75755-9},
        publisher = {Springer}
}
```

APPENDIX

**B**

# The Module `esys.pyvisi`

**Warning: The Module `esys.pyvisi` is no longer supported and will be removed from future releases.**
**Warning: The Module `esys.pyvisi` is not supported under MPI .**

## B.1 Introduction

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

The following outlines the general steps to use Pyvisi:

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

$$scene \rightarrow data\ input \rightarrow data\ visualization \rightarrow camera\ /\ light \rightarrow render \rightarrow movie$$

## B.2 `esys.pyvisi` Classes

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

### B.2.1 Scene Classes

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

### `Scene` class

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

The following are some of the methods available:

**`setBackground`** (*color*)
Set the background color of the scene.

**`render`** (*image_name = None*)
Render the object using either the Online, Offline or Display mode.

### `Camera` class

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

The following are some of the methods available:

**`setFocalPoint`** (*position*)
Set the focal point of the camera.

**`setPosition`** (*position*)
Set the position of the camera.

**`azimuth`** (*angle*)
Rotate the camera to the left and right. The angle parameter is in degrees.

**`elevation`** (*angle*)
Rotate the camera up and down (angle must be between -90 and 90).

**`backView`** ()
Rotate the camera to view the back of the rendered object.

**`topView`** ()
Rotate the camera to view the top of the rendered object.

**`bottomView`** ()
Rotate the camera to view the bottom of the rendered object.

**`leftView`** ()
Rotate the camera to view the left side of the rendered object.

**`rightView`** ()
Rotate the camera to view the right side of the rendered object.

**`isometricView`** ()
Rotate the camera to view an isometric projection of the rendered object.

**`dolly`** (*distance*)
Move the camera towards (greater than 1) the rendered object. However, it is not possible to move the camera away from the rendered object with this method.

### `Light` class

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

The following are some of the methods available:

**`setColor`** (*color*)
Set the light color.

**setFocalPoint** (*position*)
Set the focal point of the light.

**setPosition** (*position*)
Set the position of the light.

**setAngle** (*elevation = 0, azimuth = 0*)
An alternative to set the position and focal point of the light by using elevation and azimuth.

### B.2.2 Input Classes

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

#### `DataCollector` class

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

The following are some of the methods available:
**setFileName** (*file_name*)
Set the XML file name to read.

**setData** (***args*)
Create data using the <name>=<data>pairing. The method assumes that the data is given in the appropriate format.

**setActiveScalar** (*scalar*)
Specify the scalar field to load.

**setActiveVector** (*vector*)
Specify the vector field to load.

**setActiveTensor** (*tensor*)
Specify the tensor field to load.

#### `ImageReader` class

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

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

#### `Text2D` class

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

The following are some of the methods available:
**setFontSize** (*size*)
Set the 2D text size.

**boldOn** ()
Use bold font style for the text.

**setColor** (*color*)
Set the color of the 2D text.

Including methods from `Actor2D`.

### B.2.3 Data Visualization Classes

This subsection details the instances used to process and manipulate the source data. The typical usage of some of the classes is also shown. See Section B.5 for sample images generated with these classes.

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

#### `Map` class

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

Class that shows a scalar field on a domain surface. The domain surface can either be color or gray-scale, depending on the lookup table used.

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

A typical usage of `Map` is shown below.

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

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

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

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

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

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

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

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

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

```
dc2.setActiveScalar(scalar = SCALAR_FIELD_CELL_DATA)

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

# Create a Camera for the third viewport
c2 = Camera(scene = s, viewport = Viewport.NORTH_EAST)

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

### `MapOnPlaneCut` class

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

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

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

### `MapOnPlaneClip` class

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

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

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

### `MapOnScalarClip` class

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

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

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

### `MapOnScalarClipWithRotation` class

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

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

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

### `Velocity` class

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

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

using vector data or scalar data. Similarly, there are two possible types of arrows, either two-dimensional or three-dimensional.

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

### `VelocityOnPlaneCut` class

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

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

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

A typical usage of `VelocityOnPlaneCut` is shown below.

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

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

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

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

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

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

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

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

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

### `VelocityOnPlaneClip` class

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

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

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

### `Ellipsoid` class

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

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

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

### `EllipsoidOnPlaneCut` class

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

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

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

### `EllipsoidOnPlaneClip` class

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

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

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

A typical usage of `EllipsoidOnPlaneClip` is shown below.

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

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

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

TENSOR_FIELD_CELL_DATA = "stress_cell"
FILE_3D = "interior_3D.xml"
IMAGE_NAME = "ellipsoid.jpg"
JPG_RENDERER = Renderer.ONLINE_JPG
```

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

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

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

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

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

### `Contour` class

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

Class that shows a scalar field using contour surfaces. The contour surfaces can either be color or gray-scale, depending on the lookup table used. This class can also be used to generate isosurfaces.

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

A typical usage of `Contour` is shown below.

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

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

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

SCALAR_FIELD_POINT_DATA = "temperature"
FILE_3D = "interior_3D.xml"
IMAGE_NAME = "contour.jpg"
JPG_RENDERER = Renderer.ONLINE_JPG

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

# Create a DataCollector reading a XML file.
```

```
dc1 = DataCollector(source = Source.XML)
dc1.setFileName(file_name = os.path.join(PYVISI_EXAMPLE_MESHES_PATH, FILE_3D))
dc1.setActiveScalar(scalar = SCALAR_FIELD_POINT_DATA)

# Create three contours.
ctr1 = Contour(scene = s, data_collector = dc1, viewport = Viewport.SOUTH_WEST,
        lut = Lut.COLOR, cell_to_point = False, outline = True)
ctr1.generateContours(contours = 3)

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

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

### `ContourOnPlaneCut` class

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

This class works in a similar way to `MapOnPlaneCut`, except that it shows a scalar field using contour surfaces cut using a plane.

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

### `ContourOnPlaneClip` class

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

This class works in a similar way to `MapOnPlaneClip`, except that it shows a scalar field using contour surfaces clipped using a plane.

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

### `StreamLine` class

**class `StreamLine`** (*scene, data_collector, viewport = Viewport.SOUTH_WEST, color_mode = ColorMode.VECTOR, lut = Lut.COLOR, cell_to_point = False, outline = True*)

Class that shows the direction of particles of a vector field using streamlines. The streamlines can either be color or gray-scale, depending on the lookup table used. If the streamlines are colored, there are two possible coloring modes, either using vector data or scalar data.

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

A typical usage of `StreamLine` is shown below.

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

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

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

```
Y_SIZE = 400

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

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

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

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

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

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

### `Carpet` class

**class `Carpet`** (*scene, data_collector, viewport = Viewport.Viewport.SOUTH_WEST, warp_mode = WarpMode.SCALAR, lut = Lut.COLOR, cell_to_point = False, outline = True*)

This class works in a similar way to `MapOnPlaneCut`, except that it shows a scalar field cut on a plane and deformed (warped) along the normal. The plane can either be color or gray-scale, depending on the lookup table used. Similarly, the plane can be deformed either using scalar data or vector data.

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

A typical usage of `Carpet` is shown below.

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

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

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

SCALAR_FIELD_CELL_DATA = "temperature_cell"
FILE_3D = "interior_3D.xml"
IMAGE_NAME = "carpet.jpg"
JPG_RENDERER = Renderer.ONLINE_JPG
```

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

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

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

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

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

### `Legend` class

**class `Legend`** (*scene, data_collector, viewport = Viewport.SOUTH_WEST, lut = Lut.COLOR, legend = LegendType.SCALAR*)

Class that shows a scalar field on a domain surface. The domain surface can either be color or gray-scale, depending on the lookup table used

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

### `Rectangle` class

**class `Rectangle`** (*scene, viewport = Viewport.SOUTH_WEST*)

Class that generates a rectangle box.

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

### `Image` class

**class `Image`** (*scene, image_reader, viewport = Viewport.SOUTH_WEST*)

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

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

A typical usage of `Image` is shown below.

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

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

```
import os

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

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

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

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

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

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

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

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

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

### `Logo` class

**class `Logo`** (*scene, image_reader, viewport = Viewport.SOUTH_WEST*)

Class that displays a static image, in particular a logo (e.g. company symbol) and has NO interaction capability. The position and size of the logo can be specified.

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

### `Movie` class

**class `Movie`** (*parameter_file = "make_movie"*)

This class is used to create movies out of a series of images. The parameter specifies the name of a file that will contain the required information for the 'ppmtompeg' command which is used to generate the movie.

The following are some of the methods available:

**imageRange** (*input_directory, first_image, last_image*)
Use this method to specify that the movie is to be generated from image files with filenames in a certain range (e.g. 'image000.jpg' to 'image050.jpg').

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

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

A typical usage of `Movie` is shown below.

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

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

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

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

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

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

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

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

# Create a movie.
mov = Movie()
lst = []

# Read in one file one after another and render the object.
for i in range(938, 949):
    dc1.setFileName(file_name =  os.path.join(PYVISI_EXAMPLE_MESHES_PATH, \
                FILE_2D + "%06d.vtu") % i)

    s.render(image_name = os.path.join(PYVISI_EXAMPLE_IMAGES_PATH, \
            IMAGE_NAME + "%06d.jpg" % i))

    lst.append(IMAGE_NAME + "%06d.jpg" % i)

# Images (first and last inclusive) from which the movie is to be generated.
mov.imageRange(input_directory = PYVISI_EXAMPLE_IMAGES_PATH,
        first_image = IMAGE_NAME + "000938.jpg",
```

```
        last_image = IMAGE_NAME + "000948.jpg")

# Alternatively, a list of images can be specified.
#mov.imageList(input_directory = PYVISI_EXAMPLE_IMAGES_PATH, image_list = lst)

# Generate the movie.
mov.makeMovie(os.path.join(PYVISI_EXAMPLE_IMAGES_PATH, "movie.mpg"))
```

### B.2.4 Coordinate Classes

This subsection details the instances used to position rendered objects.

#### `LocalPosition` class

**class `LocalPosition`** (*x_coor, y_coor*)
Class that defines a position (X and Y) in the local 2D coordinate system.

#### `GlobalPosition` class

**class `GlobalPosition`** (*x_coor, y_coor, z_coor*)
Class that defines a position (X, Y and Z) in the global 3D coordinate system.

### B.2.5 Supporting Classes

This subsection details the supporting classes and their corresponding methods inherited by the input (see Section B.2.2) and data visualization classes (see Section B.2.3).

#### `Actor3D` class

Class that defines a 3D actor.

The following are some of the methods available:

**`setOpacity`** (*opacity*)
Set the opacity (transparency) of the 3D actor.

**`setColor`** (*color*)
Set the color of the 3D actor.

**`setRepresentationToWireframe`** ()
Set the representation of the 3D actor to wireframe.

#### `Actor2D` class

Class that defines a 2D actor.

The following are some of the methods available:

**`setPosition`** (*position*)
Set the position (XY) of the 2D actor. Default position is the lower left hand corner of the window / viewport.

#### `Clipper` class

Class that defines a clipper.

The following are some of the methods available:

**setInsideOutOn**()
Clips one side of the rendered object.

**setInsideOutOff**()
Clips the other side of the rendered object.

**setClipValue**(*value*)
Set the scalar clip value (instead of using a plane) for the clipper.

### ContourModule class

Class that defines the contour module.

The following are some of the methods available:

**generateContours**(*contours = None, lower_range = None, upper_range = None*)
Generate the specified number of contours within the specified range. In order to generate a single isosurface, the 'lower_range' and 'upper_range' must be set to the same value.

### Glyph3D class

Class that defines 3D glyphs.

The following are some of the methods available:

**setScaleModeByVector**()
Set the 3D glyph to scale according to the vector data.

**setScaleModeByScalar**()
Set the 3D glyph to scale according to the scalar data.

**setScaleFactor**(*scale_factor*)
Set the 3D glyph scale factor.

### TensorGlyph class

Class that defines tensor glyphs.

The following are some of the methods available:

**setScaleFactor**(*scale_factor*)
Set the scale factor for the tensor glyph.

**setMaxScaleFactor**(*max_scale_factor*)
Set the maximum allowable scale factor for the tensor glyph.

### PlaneSource class

Class that defines a plane source. A plane source is defined by an origin and two other points, which form the axes (X and Y).

The following are some of the methods available:

**setOrigin**(*position*)
Set the origin of the plane source.

**setPoint1**(*position*)
Set the first point from the origin of the plane source.

**setPoint2**(*position*)
Set the second point from the origin of the plane source.

### `PointSource` class

Class that defines the source (location) to generate points. The points are generated within the radius of a sphere.

The following are some of the methods available:

**setPointSourceRadius**(*radius*)
Set the radius of the sphere.

**setPointSourceCenter**(*center*)
Set the center of the sphere.

**setPointSourceNumberOfPoints**(*points*)
Set the number of points to generate within the sphere (the larger the number of points, the more streamlines are generated).

### `Sphere` class

Class that defines a sphere.

The following are some of the methods available:

**setThetaResolution**(*resolution*)
Set the theta resolution of the sphere.

**setPhiResolution**(*resolution*)
Set the phi resolution of the sphere.

### `StreamLineModule` class

Class that defines the streamline module.

The following are some of the methods available:

**setMaximumPropagationTime**(*time*)
Set the maximum length of the streamline expressed in elapsed time.

**setIntegrationToBothDirections**()
Set the integration to occur both sides: forward (where the streamline goes) and backward (where the streamline came from).

### `Transform` class

Class that defines the orientation of planes.

The following are some of the methods available:

**translate**(*x_offset, y_offset, z_offset*)
Translate the rendered object along the x, y and z-axes.

**rotateX**(*angle*)
Rotate the plane around the x-axis.

**rotateY**(*angle*)
Rotate the plane around the y-axis.

**rotateZ**(*angle*)
Rotate the plane around the z-axis.

**setPlaneToXY**(*offset = 0*)
Set the plane orthogonal to the z-axis.

**setPlaneToYZ**(*offset = 0*)
Set the plane orthogonal to the x-axis.

**setPlaneToXZ**(*offset = 0*)
Set the plane orthogonal to the y-axis.

### `Tube` class

Class that defines the tube wrapped around the streamlines.

The following are some of the methods available:

**setTubeRadius**(*radius*)
Set the radius of the tube.

**setTubeRadiusToVaryByVector**()
Set the radius of the tube to vary by vector data.

**setTubeRadiusToVaryByScalar**()
Set the radius of the tube to vary by scalar data.

### `Warp` class

Class that defines the deformation of a scalar field.

The following are some of the methods available:

**setScaleFactor**(*scale_factor*)
Set the displacement scale factor.

### `MaskPoints` class

Class that defines masking of points. This is useful to prevent the rendered object from being cluttered with arrows or ellipsoids.

The following are some of the methods available:

**setRatio**(*ratio*)
Mask every n'th point.

**randomOn**()
Enables randomization of the points selected for masking.

### `ScalarBar` class

Class that defines a scalar bar.

The following are some of the methods available:

**setTitle**(*title*)
Set the title of the scalar bar.

**setPosition**(*position*)
Set the local position of the scalar bar.

**setOrientationToHorizontal**()
Set the orientation of the scalar bar to horizontal.

**setOrientationToVertical**()
Set the orientation of the scalar bar to vertical.

**setHeight**(*height*)
Set the height of the scalar bar.

**setWidth**(*width*)
Set the width of the scalar bar.

**setLabelColor**(*color*)
Set the color of the scalar bar's label.

**setTitleColor**(*color*)
Set the color of the scalar bar's title.

### `ImageReslice` class

Class that defines an image reslice which is used to resize static (no interaction capability) images (i.e. logo).

The following are some of the methods available:

**setSize**(*size*)
Set the size factor of the image. The value must be between 0 and 2. Size 1 (one) keeps the image in its original size (which is the default).

### `DataSetMapper` class

Class that defines a data set mapper.

The following are some of the methods available:

**setScalarRange**(*lower_range, upper_range*)
Set the minimum and maximum scalar range for the data set mapper. This method is called when the range has been specified by the user. Therefore, the scalar range read from the source will be ignored.

### `CubeSource` class

Class that defines a cube source. The center of the cube source defines the point from which the cube is to be generated and the X, Y and Z lengths define the length of the cube from the center point. If X length is 3, then the X length to the left and right of the center point is 1.5 respectively.

The following are some of the methods available:

**setCenter**(*center*)
Set the cube source center.

**setXLength**(*length*)
Set the cube source length along the x-axis.

**setYLength**(*length*)
Set the cube source length along the y-axis.

**setZLength** (*length*)
Set the cube source length along the z-axis.

`Rotation` class

Class that sweeps 2D data around the z-axis to create a 3D looking effect.

The following are some of the methods available:

**setResolution** (*resolution*)
Set the resolution of the sweep for the rotation, which controls the number of intermediate points.

**setAngle** (*angle*)
Set the angle of rotation.

## B.3 More Examples

This section provides examples for some common tasks.

### B.3.1 Reading a Series of Files

The following script shows how to generate images from a time series using two data sources.

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

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

PYVISI_EXAMPLE_MESHES_PATH = "data_meshes"
PYVISI_EXAMPLE_IMAGES_PATH = "data_sample_images"
X_SIZE = 400
Y_SIZE = 300

SCALAR_FIELD_POINT_DATA_1 = "lava"
SCALAR_FIELD_POINT_DATA_2 = "talus"
FILE_2D = "phi_talus_lava."

IMAGE_NAME = "seriesofreads"
JPG_RENDERER = Renderer.ONLINE_JPG

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

# Create a DataCollector reading from an XML file.
dc1 = DataCollector(source = Source.XML)
dc1.setActiveScalar(scalar = SCALAR_FIELD_POINT_DATA_1)

# Create a Contour.
mosc1 = Contour(scene = s, data_collector = dc1,
        viewport = Viewport.SOUTH_WEST, lut = Lut.COLOR, cell_to_point = False,
        outline = True)
mosc1.generateContours(0)

# Create a second DataCollector reading from the same XML file
```

```
# but specifying a different scalar field.
dc2 = DataCollector(source = Source.XML)
dc2.setActiveScalar(scalar = SCALAR_FIELD_POINT_DATA_2)

# Create a second Contour.
mosc2 = Contour(scene = s, data_collector = dc2,
        viewport = Viewport.SOUTH_WEST, lut = Lut.COLOR, cell_to_point = False,
        outline = True)
mosc2.generateContours(0)

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

# Read in one file after another and render the object.
for i in range(99, 104):
    dc1.setFileName(file_name =  os.path.join(PYVISI_EXAMPLE_MESHES_PATH, \
                FILE_2D + "%04d.vtu") % i)
    dc2.setFileName(file_name =  os.path.join(PYVISI_EXAMPLE_MESHES_PATH, \
            FILE_2D + "%04d.vtu") % i)

    s.render(image_name = os.path.join(PYVISI_EXAMPLE_IMAGES_PATH, \
            IMAGE_NAME + "%04d.jpg") % i)
```

### B.3.2 Creating Slices of a Data Source

The following script shows how to save a series of images that slice the data at different points by gradually translating the cut plane.

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

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

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

SCALAR_FIELD_POINT_DATA = "temperature"
FILE_3D = "interior_3D.xml"
IMAGE_NAME = "seriesofcuts"
JPG_RENDERER = Renderer.ONLINE_JPG

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

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

# Create a MapOnPlaneCut.
mopc1 = MapOnPlaneCut(scene = s, data_collector = dc1,
        viewport = Viewport.SOUTH_WEST, lut = Lut.COLOR, cell_to_point = False,
        outline = True)
mopc1.setPlaneToYZ(offset = 0.1)
```

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

# Render the object with multiple cuts using a series of translations.
for i in range(0, 5):
    s.render(image_name = os.path.join(PYVISI_EXAMPLE_IMAGES_PATH, IMAGE_NAME +
                        "%02d.jpg") % i)
    mopc1.translate(0.6,0,0)
```

### B.3.3 Reading Data Directly from escript Objects

The following script shows how to combine Pyvisi code with escript code to generate visualizations on the fly.

```
"""
Author: Lutz Gross, l.gross@uq.edu.au
Author: John Ngui, john.ngui@uq.edu.au
"""

# Import the necessary modules.
from esys.escript import *
from esys.escript.linearPDEs import LinearPDE
from esys.finley import Rectangle
from esys.pyvisi import Scene, DataCollector, Map, Camera
from esys.pyvisi.constant import *
import os

PYVISI_EXAMPLE_IMAGES_PATH = "data_sample_images"
X_SIZE = 400
Y_SIZE = 400
JPG_RENDERER = Renderer.ONLINE_JPG

#... set some parameters ...
xc = [0.02,0.002]
r = 0.001
qc = 50.e6
Tref = 0.
rhocp = 2.6e6
eta = 75.
kappa = 240.
tend = 5.
# initialize time, time step size and counter ...
t=0
h=0.1
i=0

# generate domain ...
mydomain = Rectangle(l0=0.05, l1=0.01, n0=250, n1=50)
# open PDE ...
mypde = LinearPDE(mydomain)
mypde.setSymmetryOn()
mypde.setValue(A=kappa*kronecker(mydomain), D=rhocp/h, d=eta, y=eta*Tref)
# set heat source: ...
x = mydomain.getX()
qH = qc*whereNegative(length(x-xc)-r)

# set initial temperature ....
T=Tref

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

```
# Create a DataCollector reading directly from escript objects.
dc = DataCollector(source = Source.ESCRIPT)

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

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

# start iteration
while t < 0.4:
      i += 1
      t += h
      mypde.setValue(Y=qH+rhocp/h*T)
      T = mypde.getSolution()

      dc.setData(temp = T)

      # Render the object.
      s.render(image_name = os.path.join(PYVISI_EXAMPLE_IMAGES_PATH, \
              "diffusion%02d.jpg") % i)
```

## B.4 Useful Keys

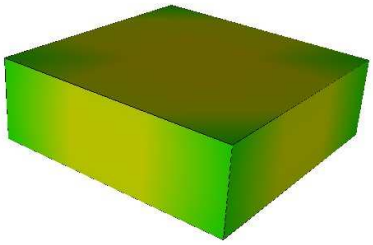
This section lists keyboard shortcuts available when interacting with rendered objects using the Online approach.

| Key | Description |
|---|---|
| Keypress 'c' / 'a' | Toggle between the camera ('c') and object ('a') mode. In camera mode, mouse events affect the camera position and focal point. In object mode, mouse events affect the rendered object's element (i.e. cut surface map, clipped velocity field, streamline, etc) that is under the mouse pointer. |
| Mouse button 1 | Rotate the camera around its focal point (if in camera mode) or rotate the rendered object's element (if in object mode). |
| Mouse button 2 | Pan the camera (if in camera mode) or translate the rendered object's element (if in object mode). |
| Mouse button 3 | Zoom the camera (if in camera mode) or scale the rendered object's element (if in object mode). |
| Keypress 3 | Toggle the render window in and out of stereo mode. By default, red-blue stereo pairs are created. |
| Keypress 'e' / 'q' | Exit the application if only one file is to be read, or read and display the next file if multiple files are to be read. |
| Keypress 's' | Modify the representation of the rendered object to surfaces. |
| Keypress 'w' | Modify the representation of the rendered object to wireframe. |
| Keypress 'r' | Reset the position of the rendered object to the center. |

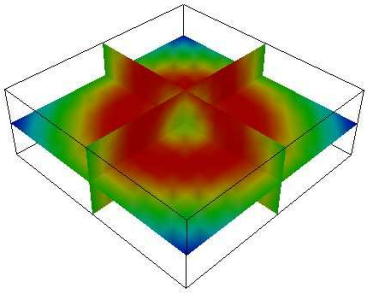
Table B.1: Useful keys in Online render mode

## B.5 Sample Output

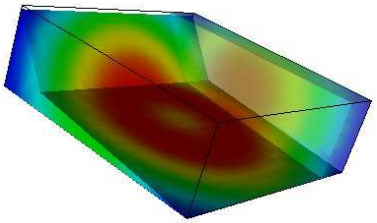
This section shows sample images produced with the various classes of Pyvisi. The source code to produce these images is included in the Pyvisi distribution.



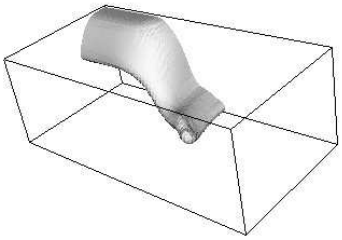
Map



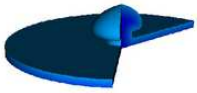
MapOnPlaneCut



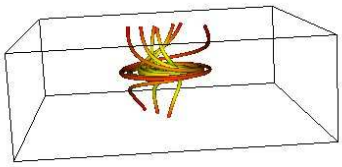
MapOnPlaneClip



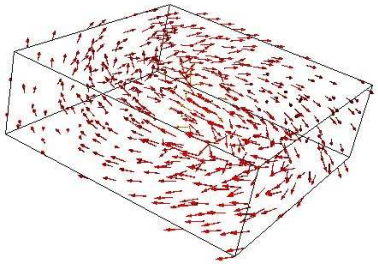
MapOnScalarClip



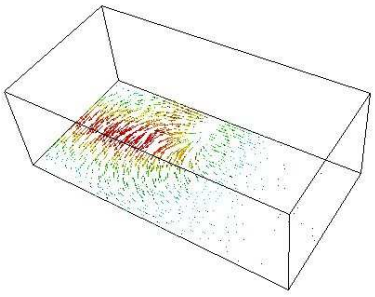
MapOnScalarClipWithRotation



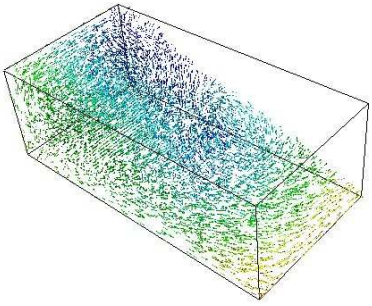
Streamline



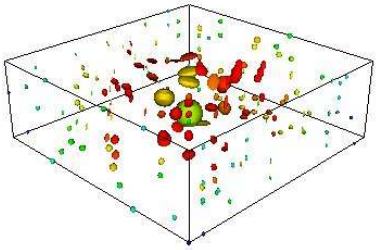
Velocity



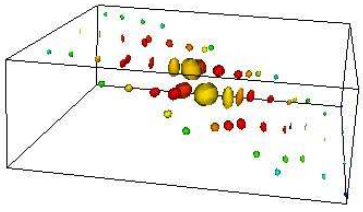
VelocityOnPlaneCut



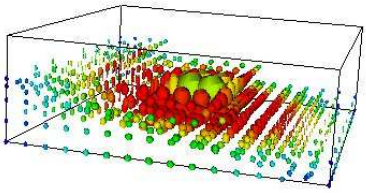
VelocityOnPlaneClip



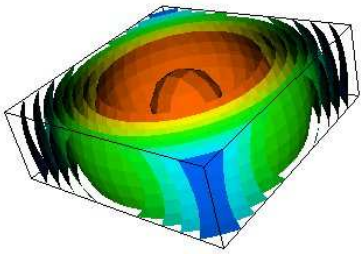
Ellipsoid



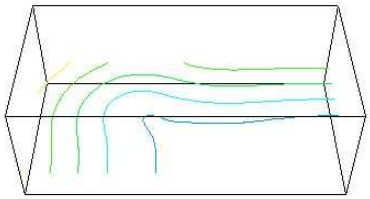
EllipsoidOnPlaneCut



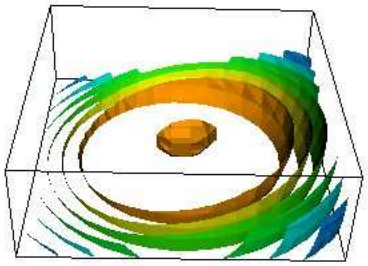
EllipsoidOnPlaneClip



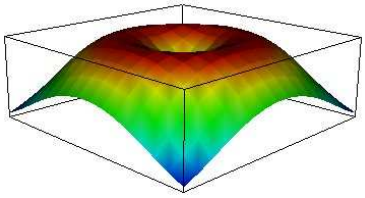
Contour



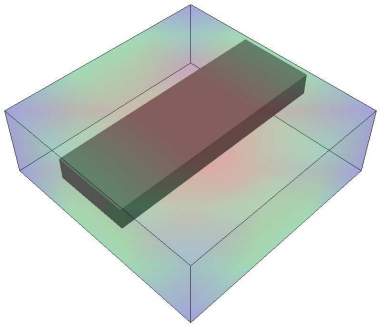
ContourOnPlaneCut



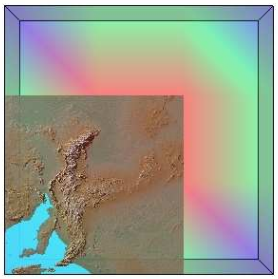
ContourOnPlaneClip



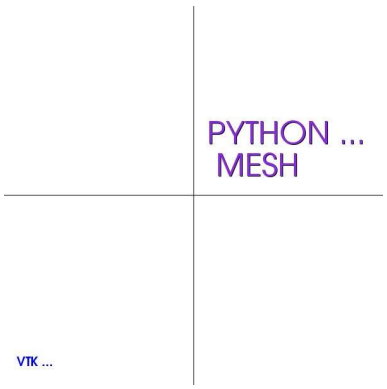
Carpet



Rectangle

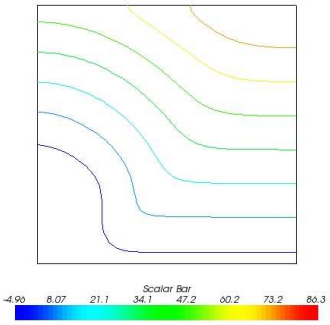


Image




Text


Logo




Legend

# INDEX

# BIBLIOGRAPHY


[1] *Right-hand rule*.

[2] *Mayavi2: The next generation scientific data visualization*, 2009.

[3] A. Amirberkyan and L. Gross. Efficient solvers for incompressible fluid flows in geosciences. *ANZIAM Journal*, 50:C189–C203, 2008.

[4] M. Benzi, G. H. Golub, and J. Liesen. Numerical solution of saddle point problems. *Acta Numerica*, 14:1–137, 2005.

[5] P. G. Ciarlet and J. L. Lions. *Handbook of Numerical Analysis*, volume 2. North Holland, Amsterdam, 1991.

[6] Scipy Community. *Numpy and Scipy Documentation*.

[7] The Scipy community. *Numpy and Scipy Documentation*.

[8] Howard Elman David Kay David Silvester and Andrew Wathen. Efficient preconditioning of the linearized navier-stokes equations. *Journal of Computational and Applied Mathematics*, (128):261–279, 2001.

[9] Tim Edwards. *Netgen 1.4*.

[10] Christophe Geuzaine and Jean-Francois Remacle. *Gmsh Reference Manual*, 1.12 edition, Aug 2003.

[11] V. Girault and P. A. Raviart. *Finite Element Methods for Navier-Stokes Equations- Theory and Algorithms*. Springer Verlag, Berlin, 1986.

[12] John Hunter, Michael Droettboom, and Darren Dale. *matplotlib*, July 2009.

[13] Intel's math kernel library.

[14] MPI. http://www.mpi-forum.org.

[15] Hans-Bernd; Regenauer-Lieb Klaus Muhlhaus. Towards a self-consistent plate mantle model that includes elasticity: simple benchmarks and application to basic modes of convection. *Geophysical Journal International*, 163(2):788–800(13), November 2005.

[16] netCDF. http://www.unidata.ucar.edu/software/netcdf.

[17] OpenDX. http://www.opendx.org/.

[18] OpenMP. http://openmp.org.

[19] A. I. Pehlivanov, G. F. Carey, and R. D. Lazarov. Least-squares mixed finite elements for second-order elliptic problems. *SIAM Journal on Numerical Analysis*, 31(5):1368–1377, October 1994.

[20] Y. Saad. *Iterative Methods for Sparse Linear Systems*. PWS Publishing Company, 20 Park Plaza, Boston, MA 02116, USA, 1996.

[21] Y. Shapira. *Matrix-Based Multigrid*. Springer, 2008.

[22] Hang Si. *TetGen: A Quality Tetrahedral Mesh Generator and Three-Dimensional Delaunay Triangulator*, Jan 2008.

[23] http://www.cise.ufl.edu/research/sparse/umfpack/.

[24] VisIt. https://wci.llnl.gov/codes/visit/home.html.

[25] R. Weiss. *Parameter-Free Iterative Linear Solvers*. Mathematical Research, vol. 97. Akademie Verlag, Berlin, 1996.

[26] Thomas Williams and Colin Kelley. *gnuplot homepage*, March 2009.

[27] B. Suchomel Y. Saad. Arms: an algebraic recursive multilevel solver for general sparse linear systems. *Numerical Linear Algebra with Applications*, 9(5):1099–1506, 2002.

[28] O. C. Zienkiewicz. *The Finite Element Method in Engineering Science*. McGraw-Hill, London, second edition, 1971.