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

Release - 6
(r1766120800)

Lutz Gross *et al.* (Editor)

December 19, 2025

School of Earth and Environmental Sciences
The University of Queensland
Brisbane, Australia

This work was supported by the AuScope National Collaborative Research Infrastructure Strategy, the Queensland State Government and The University of Queensland.

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
- PDE solver interface
- finite element solvers `esys.finley`, `esys.ripley`, `esys.speckley` and `esys.oxley`
- interface to visualization tools

All `esys.escript` modules require *python* 3 (version 3.7 or later). The current version supports parallelization through *MPI* for distributed memory, *OpenMP* for shared memory on CPUs, as well as *CUDA* for some GPU-based solvers.

This release comes with some significant changes and new features. Please see the RELEASE_NOTES file for a detailed list. In particular, if you wish to solve PDEs with complex coefficients, please consult the install guide regarding Trilinos support.

If you use this software in your research, then we would appreciate (but do not require) a citation. Please see the README.md file for relevant references.

Contents

CHAPTER
ONE

Tutorial: Solving PDEs

1.1 Installation

To download *escript* and friends, please visit `https://launchpad.net/escript-finley`. The web site offers binary distributions for some platforms, source code, documentation including information about the installation process, as well as a way to ask questions.

1.2 The First Steps

This chapter is an introduction on how to use `esys.escript` to solve a partial differential equation (PDE). We assume you are at least a little familiar with *python*. The knowledge presented in the *python* tutorial at `https://docs.python.org/2/tutorial/` 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 Ω, is the unit square

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

The domain is shown in Figure 1.1.

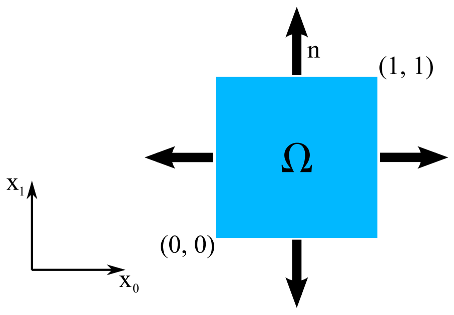


FIGURE 1.1: Domain $\Omega = [0,1]^2$ with outer normal field n.

Δ 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 right 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 compact Einstein summation convention. This drops the $\sum$ sign and assumes that a summation is performed over any repeated index. For instance,

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

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

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

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

$$\tag{1.9}$$

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 Ω the normal derivative $n_i u_{,i}$ of the solution u shall be zero, i.e. u shall fulfill the homogeneous Neumann boundary condition

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

$n = (n_i)$ denotes the outer normal field of the domain, see Figure 1.1. Remember that we apply 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 Γ^N, 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, 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 identical to zero:

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

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

[1]You may be more familiar with the Laplace operator being written as ∇^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 be more familiar with the notation $\frac{\partial u}{\partial n} = n_i u_{,i} = \mathbf{n} \cdot \nabla u$.

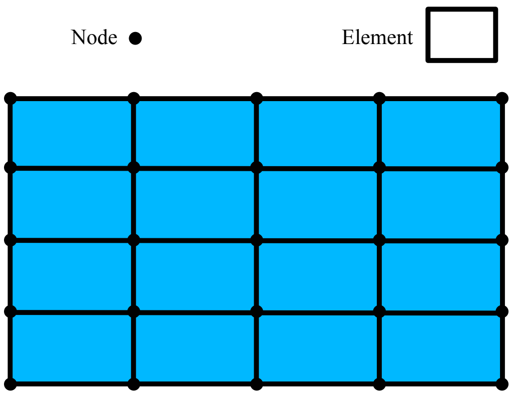


FIGURE 1.2: Mesh of 4×4 elements on a rectangular domain. Here each element is a quadrilateral and described by four nodes, namely the corner points. The solution is interpolated by a bi-linear polynomial.

In general the BVP cannot be solved analytically and numerical methods are used to construct an approximation of the solution u. Here we will use the finite element method (FEM). The basic idea is to fill the domain with a set of points called nodes. The solution is approximated by its values on the nodes. Moreover, the domain is subdivided into smaller sub-domains called elements. On each element the solution is represented by a polynomial of a certain degree through its values at the nodes located in the element. The nodes and their connection through elements is called a mesh. Figure 1.2 shows an example of a FEM mesh with four elements in the x_0 and four elements in the x_1 direction over the unit square. For more details we refer the reader to the literature, for instance Reference [29, 2].

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

```
run-escript
```

which will pass you on to the *python* prompt

```
Python 2.7.6 (default, Mar 22 2014, 15:40:47)
[GCC 4.8.2] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

Here you can use all available *python* commands and language features[4], for instance

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

We refer to the *python* user's guide if you are 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 Ω. In `esys.escript` `Domain` class objects are used to describe the geometry of a domain but it also contains information about the discretization methods and the solver used to solve the PDE. Here we use the FEM library `esys.finley`. The following statements create the `Domain` object `mydomain` from the `esys.finley` function `Rectangle`:

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

[3]**`run-escript`** is not available under Windows. 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.

[4]Throughout our examples, we use the python 3 form of print. That is, print(1) instead of print 1.

In this case the domain is a rectangle with the lower left corner at point $(0,0)$ and the upper right corner at $(\mathtt{l0},\mathtt{l1}) = (1,1)$. 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 see Chapter 5, Chapter 6, and 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 Γ^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

```
Finley_Nodes [ContinuousFunction(domain)] on FinleyMesh
```

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

```
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 () Finley_Nodes [ContinuousFunction(domain)] 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 a function which equals 1 where `x[0]` is (almost) equal to zero and 0 elsewhere. Similarly, `whereZero(x[1])` creates a function which equals 1 where `x[1]` is equal to zero and 0 elsewhere. The sum of the results of `whereZero(x[0])` and `whereZero(x[1])` gives a function on the domain `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` as `x0` used to define it. So from

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

one gets

```
0 () Finley_Nodes [ContinuousFunction(domain)] 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 the homogeneous Dirichlet boundary condition is set. The complete definition of our example is now:

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

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

Now we can write the script to solve our Poisson problem

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

The question is what we do with the calculated solution `u`. Besides postprocessing, e.g. calculating the gradient or the average value, which will be discussed later, plotting the solution is one of the things you might want to do. `esys.escript` offers two ways to do this, both based on external modules or packages. The first option uses the `matplotlib` module which allows plotting of 2D results relatively quickly from within the *python* script, see [10]. However, there are limitations when using this tool, especially for large problems and when solving three-dimensional problems. Therefore, `esys.escript` provides functionality to export data as files which can subsequently be read by third-party software packages such as *Mayavi2* [12] or *VisIt* [25].

1.2.1 Plotting Using `matplotlib`

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

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

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

`x_grid` is an array defining the x coordinates of the grid 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 *SciPy*[21] `interpolate.griddata` function to do this. Spatial coordinates are easily extracted as a `list` by

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

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

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

The values in `z` are the values at the points with the coordinates given by `x` and `y`. These values are interpolated to the grid defined by `x_grid` and `y_grid` by using

```
import scipy.interpolate
z_grid = scipy.interpolate.griddata((x,y),z,(x_grid[None,:],y_grid[:,None]),'linear')
```

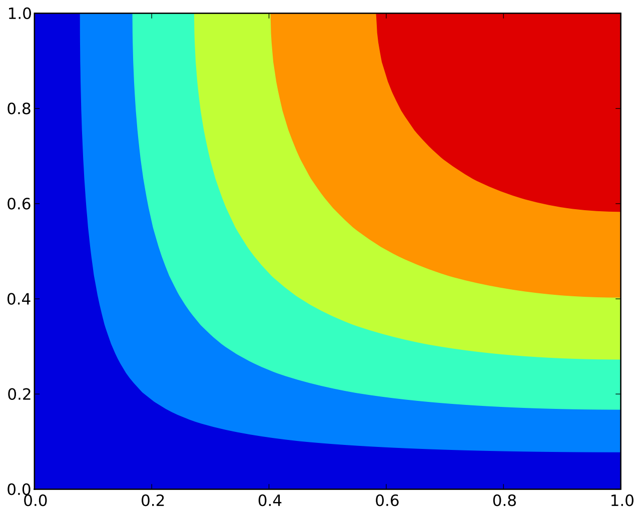


FIGURE 1.3: Visualization of the Poisson Equation Solution for $f = 1$ using `matplotlib`

Now `z_grid` gives the values of the PDE solution `u` at the grid which can be plotted using `contourf`:

```
import matplotlib
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 scipy.interpolate
import numpy
import matplotlib

import pylab
# generate domain:
mydomain = Rectangle(l0=1.,l1=1.,n0=40, n1=20)
# define characteristic function of Gamma^D
x = mydomain.getX()
gammaD = whereZero(x[0])+whereZero(x[1])
# define PDE and get its solution u
mypde = Poisson(domain=mydomain)
mypde.setValue(f=1,q=gammaD)
u = mypde.getSolution()
# interpolate u to a matplotlib grid:
x_grid = numpy.linspace(0.,1.,50)
y_grid = numpy.linspace(0.,1.,50)
x=mydomain.getX()[0].toListOfTuples()
y=mydomain.getX()[1].toListOfTuples()
z=interpolate(u,mydomain.getX().getFunctionSpace()).toListOfTuples()
z_grid = scipy.interpolate.griddata((x,y),z,(x_grid[None,:],y_grid[:,None]),'linear')
```

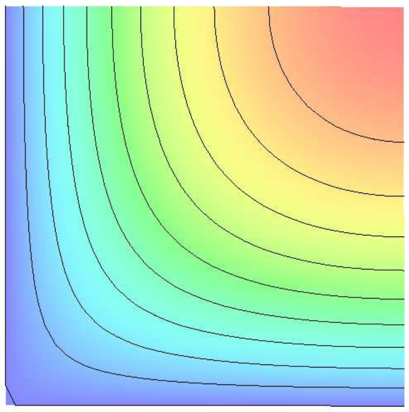


FIGURE 1.4: Visualization of the Poisson Equation Solution for $f = 1$

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

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

```
run-escript poisson_matplotlib.py
```

This will create a file called `u.png`, see Figure 1.3. For details on the usage of the `matplotlib` module we refer to the documentation [10].

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

1.2.2 Visualization using export files

As an alternative to `matplotlib`, *escript* supports exporting data to *VTK* and *SILO* files which can be read by visualization tools such as *Mayavi2* [12] and *VisIt* [25]. This method is *MPI* safe and works with large 2D and 3D problems.

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

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

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

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

will write `u` to a *SILO* file if escript was compiled with support for LLNL's *SILO* library.

The Poisson problem script is now

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

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

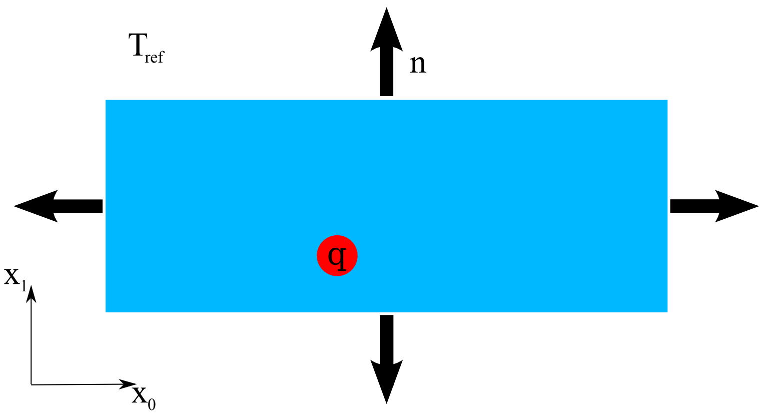


FIGURE 1.5: Temperature Diffusion Problem with Circular Heat Source

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

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

You can run the script using the *escript* environment and visualize the solution using *Mayavi2*:

```
run-escript poisson_vtk.py
mayavi2 -d u.vtu -m Surface
```

The result is shown in Figure 1.4.

1.3 The Diffusion Problem

1.3.1 Outline

In this section we 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 temperature diffusion. On the surface, energy can radiate into the surrounding environment. Figure 1.5 shows the configuration.

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

1.3.2 Temperature Diffusion

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

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

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

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

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

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

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

η 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. Note that the initial conditions satisfy the boundary condition defined by Equation (1.17).

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

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

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 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 the Helmholtz equation.

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

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

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

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

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

$$u = r \text{ where } q > 0 \tag{1.27}$$

Both r and q are a 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:

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

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

The coefficients can be set by several calls to `setValue` in arbitrary order. 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()
```

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

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

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

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

[6]There is a difference in `esys.escript` for a coefficient to be not present and set to zero. Since in the former case the coefficient is not processed, it is more efficient to leave it undefined instead of assigning zero to it.

[7]Note that this is not a complete code. The full source code can be found in "helmholtz.py".

```
print(mypde.checkSymmetry()) # returns False
mypde.setValue(C=kronecker(mydomain)[0])
print(mypde.checkSymmetry()) # returns True
```

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

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

Now we want to see how we 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. We take $T = x_0$ (here $q_H = 0$) and then calculate right hand side terms f and g such that the test solution becomes the solution of the problem. If we assume κ is 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 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
from esys.weipa import saveVTK
# set some parameters
kappa=1.
omega=0.1
eta=10.
# generate domain
mydomain = Rectangle(l0=5., l1=1., n0=50, n1=10)
# open PDE and set coefficients
mypde=LinearPDE(mydomain)
mypde.setSymmetryOn()
n=mydomain.getNormal()
x=mydomain.getX()
mypde.setValue(A=kappa*kronecker(mydomain), D=omega,Y=omega*x[0], \
               d=eta, y=kappa*n[0]+eta*x[0])
# calculate error of the PDE solution
u=mypde.getSolution()
print("error is ",Lsup(u-x[0]))
saveVTK("x0.vtu", sol=u)
```

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

```
mayavi2 -d x0.vtu -m Surface
```

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

The script is similar to the script `poisson.py` discussed in Chapter 1.2. `mydomain.getNormal()` returns the outer normal field on the surface of the domain. The function `Lsup` is imported by the `from esys.escript import *` statement and returns the maximum absolute value of its argument. The error shown by the print statement should be in the order of 10^{-7}. As piecewise bi-linear interpolation is used by `esys.finley` to approximate the solution, and our solution is a linear function of the spatial coordinates, one might expect that the error would be zero or in the order of machine precision (typically $\approx 10^{-15}$). However most PDE packages use an iterative solver which is terminated when a given tolerance has been reached. The default tolerance is 10^{-8}. This value can be altered by using the `setTolerance` method 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 (1.22). The statement `T=mypde.getSolution()` overwrites `T` with the temperature of the new time step `t` + `h`. To get this iterative process going we need to specify the initial temperature distribution, which is equal to T_{ref}. The `LinearPDE` object `mypde` and the coefficients that do not change over time are set up before the loop is entered. In each time step only the coefficient Y is reset as it depends on the temperature of the previous time step. This allows the PDE solver to reuse information from previous time steps as much as possible.

The heat source q_H which is defined in Equation (1.16) is `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 an `esys.escript.Data` object which contains locations in the `Domain` `mydomain`. The `length()` function (also 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
from esys.weipa import saveVTK
#... set some parameters ...
xc=[0.02, 0.002]
r=0.001
qc=50.e6
Tref=0.
rhocp=2.6e6
eta=75.
kappa=240.
tend=5.
# ... time, time step size and counter ...
t=0
h=0.1
i=0
#... generate domain ...
mydomain = Rectangle(l0=0.05, l1=0.01, n0=250, n1=50)
#... open PDE ...
mypde=LinearPDE(mydomain)
mypde.setSymmetryOn()
mypde.setValue(A=kappa*kronecker(mydomain), D=rhocp/h, d=eta, y=eta*Tref)
# ... set heat source: ....
x=mydomain.getX()
qH=qc*whereNegative(length(x-xc)-r)
```

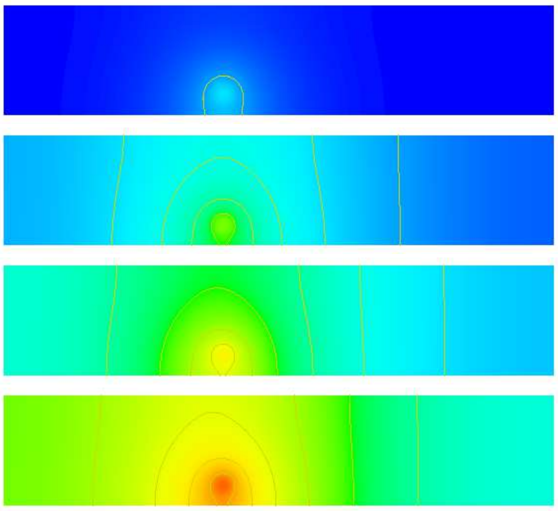


FIGURE 1.6: Results of the Temperature Diffusion Problem for Time Steps 1, 16, 32 and 48 (top to bottom)

```
# ... 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.vtu"%i, temp=T)
```

The script will create the files `T.1.vtu`, `T.2.vtu`, ..., `T.50.vtu` in the directory where the script has been started. The files contain the temperature distributions at time steps $1, 2, i, \ldots, 50$ 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

```
mayavi2 -d T.1.vtu -m Surface
```

Use the *Configure Data* window in *Mayavi2* to move forward and backward in time.

1.4 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:

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

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

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

where λ and μ are the Lamé coefficients and δ_{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

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

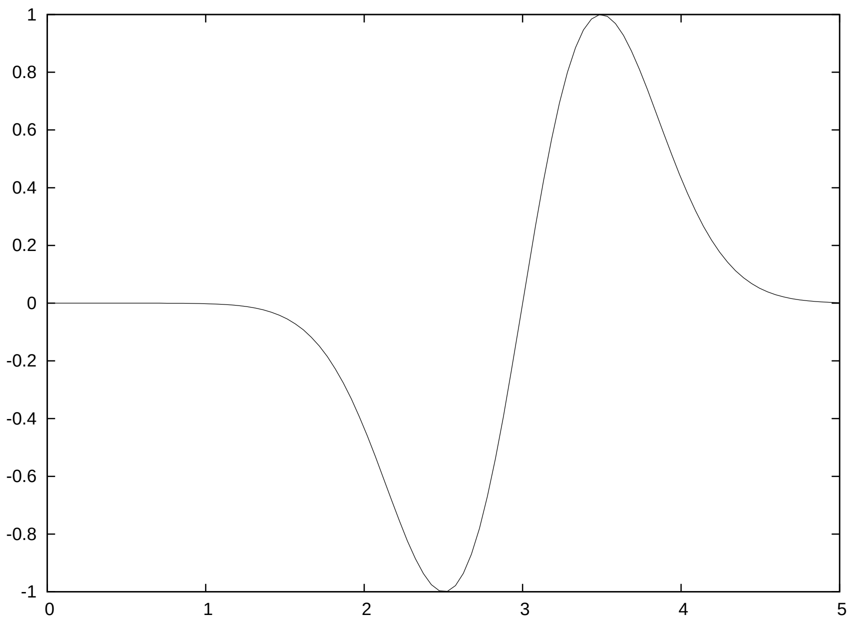


FIGURE 1.7: Input Displacement at Source Point ($\alpha = 0.7$, $t_0 = 3$, $U_0 = 1$).

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

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

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

$$\tag{1.36}$$

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

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

We also need to apply the constraint

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

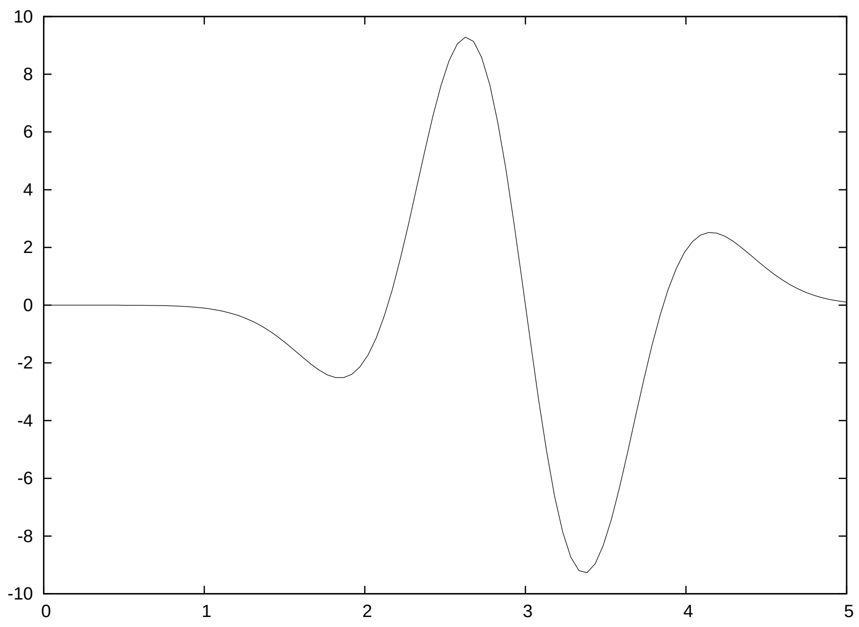


FIGURE 1.8: Input Acceleration at Source Point ($\alpha = 0.7$, $t_0 = 3$, $U_0 = 1$).

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

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

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

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:

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

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 3% of the width of the domain):

$$q_i(x) = \begin{cases} 1 & \text{where } \|x - x_c\| \le R \\ 0 & \text{otherwise.} \end{cases} \tag{1.45}$$

The following script defines the function `wavePropagation` which implements the Verlet scheme to solve our wave propagation problem. The argument `domain` which is a `Domain` class object defines the domain of the problem. `h` and `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(SolverOptions.HRZ_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)
    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 for efficiency reasons since it allows the `LinearPDE` class to reuse information as much as possible when iterating over time.

The statement

```
mypde.getSolverOptions().setSolverMethod(SolverOptions.HRZ_LUMPING)
```

enables the use of an aggressive approximation of the PDE operator as a diagonal matrix formed from the coefficient D. The approximation allows, at the cost of additional error, very fast solution of the PDE, see also Section 4.4.

There are a few new 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 (i.e. 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 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 3.2.3.

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

```
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. These values are handed back to the caller. Later we will show ways to access these data.

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

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

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

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

The following script uses the wavePropagation function to solve the wave equation for a point source located at the bottom face of a block. The width of the block in each direction on the bottom face is 10km (x_0 and x_1 directions, i.e. l0 and l1). The variable 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 depth is chosen such that the elements become cubic. We chose 10 for the number of elements in the x_2-direction so that the computation is faster. 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
```

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

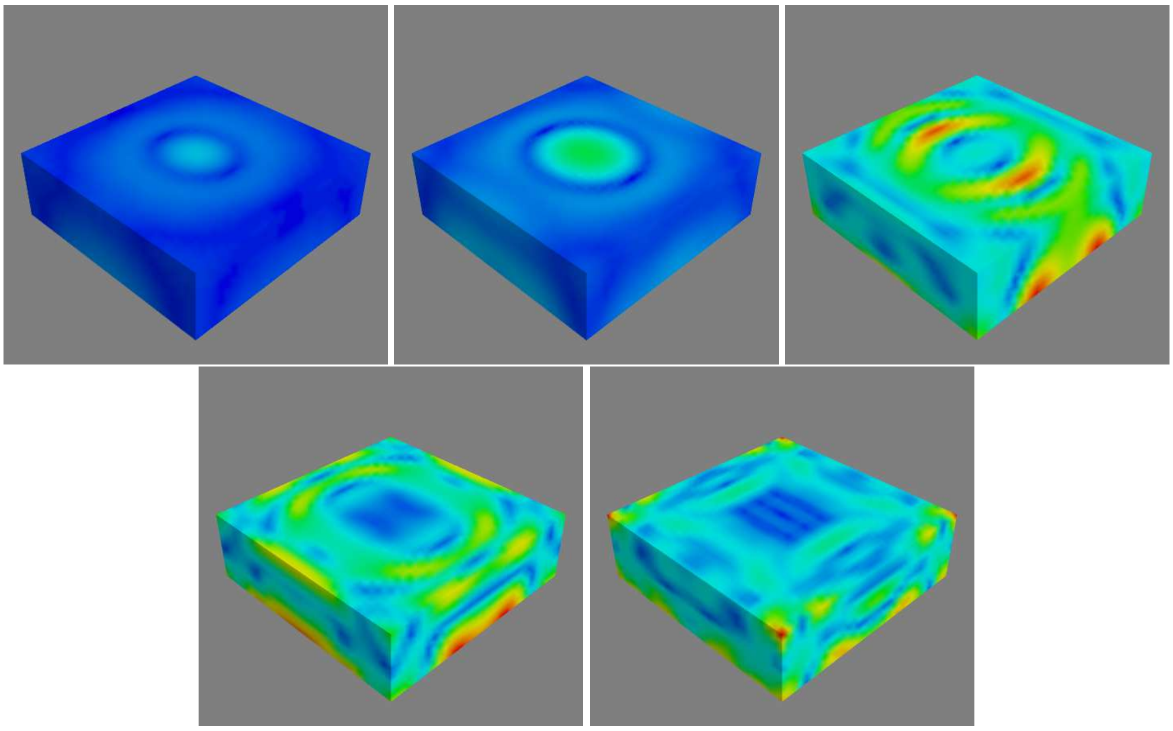


FIGURE 1.9: Selected time steps ($n = 11, 22, 32, 36$) of a wave propagation over a 10km $\times$ 10km $\times$ 3.125km block from a point source initially at (5km, 5km, 0) with time step size $h = 0.02083$. Color represents the displacement. Here the view is oriented onto the bottom face.

```
# spherical source at middle of bottom face
xc=[width/2.,width/2.,0.]
# define small radius around point xc
src_radius = 0.03*width
print("src_radius =",src_radius)
mydomain=Brick(ne, ne, 10, l0=width, l1=width, l2=10.*width/32.)
h=(1./5.)*inf(sqrt(rho/(lam+2*mu)))*inf(domain.getSize())
print("time step size =",h)
ts, u_pc0, u_pc1, u_pc2 =  \
        wavePropagation(mydomain, h, tend, lam, mu, rho, xc, src_radius, U0)
```

The `domain.getSize()` function returns the local element size Δx. Using `inf` ensures that the CFL condition 1.46 holds everywhere in the domain.

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

```
mayavi2 -d usoln.1.vtu -m Surface
```

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

It remains to show some possibilities to inspect the collected data `u_pc0`, `u_pc1` and `u_pc2`. One way is to write the data to a file and then use an external package such as *gnuplot*[28], LibreOffice Calc or Excel to read the data for further analysis. The following code shows one possible way to write the data to the file `./data/U_pc.csv`:

```
u_pc_data=FileWriter('./data/U_pc.csv')
for i in range(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()
```

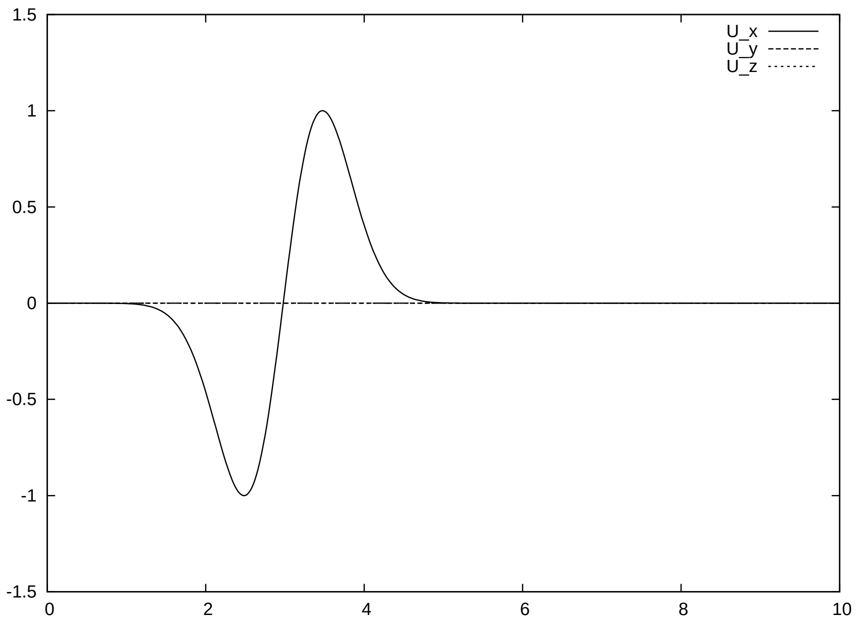


FIGURE 1.10: Amplitude at Point source from the Simulation

The file `U_pc.csv` stores 4 columns of data: t, u_x, u_y, u_z respectively, where u_x, u_y, u_z are the x_0, x_1, x_2 components of the displacement vector u at the point source. These can be plotted easily using any plotting package. In *gnuplot*[28] the command:

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

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

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

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

You can add `plt.show()` to create an interactive browser window. Notice that by checking the condition `getMPIRankWorld()==0` the plot is generated on one processor only (in this case the rank 0 processor) when run under *MPI*.

Both options for processing the point source data are included in the example file `wave.py`. There are other options available to process these data in particular through the *SciPy*[21] package, e.g. Fourier transformations. It is beyond the scope of this user's guide to document the usage of *SciPy*[21] for time series analysis but it is highly recommended to look in relevant readily available documentation.

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 σ 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 λ and μ are the Lamé coefficients, α is the temperature expansion coefficient, T is the temperature distribution and T_{ref} a reference temperature. Note that Equation (1.47) is similar to Equation (1.31) introduced in Section 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 σ in Equation (1.48) an extra term is introduced to bring in stress due to volume changes through temperature dependent expansion.

Our domain is the unit cube

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

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 β 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 Lamé equation. We want to solve this using the `LinearPDE` class. For a system of PDEs and a solution with several components the `LinearPDE` class takes PDEs of the form

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

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

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

while constraints take the form

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

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

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

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

$$\tag{1.58}$$

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 λ, μ, β 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.

Analogous to the 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 Lamé 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 Lamé 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 stress and write it to a file for visualization.

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

```
from esys.escript import *
from esys.escript.linearPDEs import LinearPDE
from esys.finley import Brick
from esys.weipa import saveVTK
#... set some parameters ...
lam=1.
mu=0.1
alpha=1.e-6
xc=[0.3, 0.3, 1.]
beta=8.
T_ref=0.
T_0=1.
#... generate domain ...
mydomain = Brick(l0=1., l1=1., l2=1., n0=10, n1=10, n2=10)
x=mydomain.getX()
#... set temperature ...
T=T_0*exp(-beta*length(x-xc))
#... open symmetric PDE ...
mypde=LinearPDE(mydomain)
mypde.setSymmetryOn()
#... set coefficients ...
C=Tensor4(0., Function(mydomain))
for i in range(mydomain.getDim()):
  for j in range(mydomain.getDim()):
     C[i,i,j,j]+=lam
     C[i,j,i,j]+=mu
     C[i,j,j,i]+=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-Mises 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. \
```

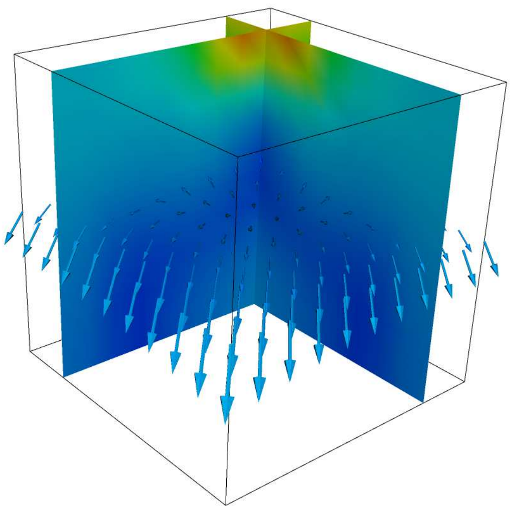


FIGURE 1.11: von-Mises Stress and Displacement Vectors

```
                +3*(sigma[0,1]**2 + sigma[1,2]**2 + sigma[2,0]**2))
  #... output ...
  saveVTK("deform.vtu", disp=u, stress=sigma_mises)
```

Finally, the results can be visualized by calling

```
mayavi2 -d deform.vtu -f CellToPointData -m Vectors -m Surface
```

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

1.6 Point Sources

In the chapter we will show the usage of point sources and sinks. A simple example is a block of material with heat source at a location p and heat sink at a location q. Under the assumption of a constant conductivity the steady heat diffusion equation for the temperature u is given as

$$-u_{,ii} = s(p_{in})\;\delta_{p_{in}} + s(p_1)\;\delta_{p_1} \tag{1.63}$$

where $\delta_{p_{in}}$ and $\delta_{p_{out}}$ refer to the Dirac δ-function and $s(p_{in})$ and $s(p_{out})$ define the heat production and heat extraction rates at locations p_{in} and p_{out}, respectively.

First the locations of point sources and sinks need to be added to the domain. This is done at generation time:

```
mydomain=Rectangle(30,30, l0=3, l1=2,
                diracPoints=[(1.,1.), (2.,1.)], diracTags=['in', 'out'])
```

In this case the points are located at $p_{in} = (1., 1.)$ and $p_{out} = (2., 1.)$. For easier reference the points are tagged with the name `in` and `out`.

The values at the point source locations are defined using a `Data` object. One possible way to define the values at the locations defined through the `diracPoint` list is using tagging:

```
s=Scalar(0., DiracDeltaFunctions(mydomain))
s.setTaggedValue('in', +1.)
s.setTaggedValue('out', -1.)
```

Here we set value 1 at locations tagged with in (in this case this is just point $p_{in} = (1., 1.)$) and value -1 at locations tagged with out (in this case this is just point $p_{out} = (2., 1.)$). The point source in the right hande side of the PDE Equation (1.63) is then set as

```
mypde = LinearSinglePDE(domain=mydomain)
mypde.setValue(y_dirac=s)
```

Under the assumption that we fix the temperature to zero on the entire boundary the script to solve the PDE is given as follows:

```
from esys.escript import *
from esys.weipa import *
from esys.finley import Rectangle
from esys.escript.linearPDEs import LinearSinglePDE

mydomain=Rectangle(30,30, l0=3, l1=2,
                diracPoints=[(1.,1.), (2.,1.)],  diracTags=['in', 'out'])
x = mydomain.getX()
gammaD = whereZero(x[0])+whereZero(x[1])+whereZero(x[0]-3.)+whereZero(x[1]-2.)

s=Scalar(0., DiracDeltaFunctions(mydomain))
s.setTaggedValue('in', +1.)
s.setTaggedValue('out', -1.)

mypde = LinearSinglePDE(domain=mydomain)
mypde.setValue(q=gammaD, A=kronecker(2), y_dirac=s)
u = mypde.getSolution()
saveVTK("u.vtu",sol=u)
```

Result is shown in Figure 1.12.

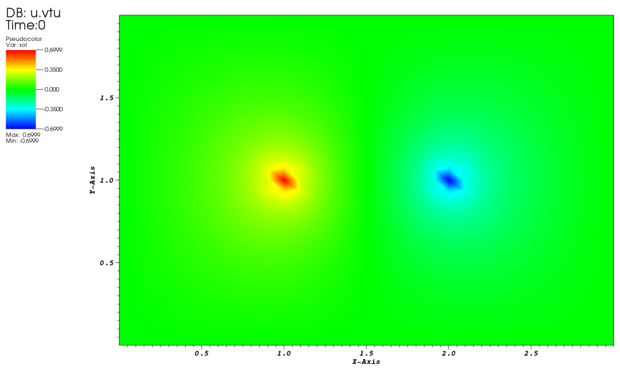


FIGURE 1.12: Results diffusion problem with nodal source and sink.

CHAPTER
TWO

Execution of an *escript* Script

2.1 Overview

A typical way of starting your *escript* script `myscript.py` is with the **`run-escript`** command[1]. This command was renamed from **`escript`** (used in previous releases) to avoid clashing with an unrelated program installed by default on some systems. To run your script, issue[2]

```
run-escript myscript.py
```

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

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

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

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

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

This requires *escript* to be compiled with *OpenMP* [15] support. To run the script using *MPI* [14] with 8 processes use

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

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

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

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

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

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

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

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

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

2.2 Options

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

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

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

The options have the following meaning:

-n `nn`	the number of compute nodes `nn` to be used. The total number of processes being used is $\texttt{nn} \cdot \texttt{np}$. This option overrides the value of the **ESCRIPT_NUM_NODES** environment variable. If a `hostfile` is given (see below), the number of nodes needs to match the number of hosts given in that file. If $\texttt{nn} > 1$ but *escript* is not compiled for *MPI*, a warning is printed but execution is continued with $\texttt{nn} = 1$. 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 $\texttt{nn} \cdot \texttt{np}$. This option overwrites the value of the **ESCRIPT_NUM_PROCS** environment variable. If $\texttt{np} > 1$ but *escript* is not compiled for *MPI*, a warning is printed but execution is continued with $\texttt{np} = 1$. The default value is 1.
-t `nt`	the number of threads used per process. The option overwrites the value of the *OpenMP* environment variable **ESCRIPT_NUM_THREADS**. If $\texttt{nt} > 1$ but *escript* is not compiled for *OpenMP* , a warning is printed but execution is continued with $\texttt{nt} = 1$. The default value is 1.
-f `hostfile`	the name of a file with a list of host names. Some systems require to specify the addresses or names of the compute nodes where *MPI* processes should be spawned. These addresses or names of the compute nodes are listed in the file with the name `hostfile`. If -n is set, the number of different hosts defined in `hostfile` must be equal to the number of requested compute nodes `nn`. The option overwrites the value of the **ESCRIPT_HOSTFILE** environment variable. By default no host file is used.
-c	prints information about the settings used to compile *escript* and stops execution.
-V	prints the version of *escript* and stops execution.
-h	prints a help message and stops execution.
-i	executes the script `file` and switches to interactive mode after the execution is finished or an exception has occurred. This option is useful for debugging a script. The option cannot be used if more than one process ($\texttt{nn} \cdot \texttt{np} > 1$) is used.
-b	do not invoke python. This is used to run non-python programs within an environment set for *escript*.
-e	shows additional environment variables and commands used to set up the *escript* environment. This option is useful if users wish to execute scripts without using the **`run-escript`** command.
-o	enables the redirection of messages printed by processors with *MPI* rank greater than zero to the files `stdout_r.out` and `stderr_r.out` where `r` is the rank of the processor. The option overwrites the value of the **ESCRIPT_STDFILES** environment variable.
-x	runs everything within a new *xterm* instance.
-v	prints some diagnostic information.
-m `tool`	runs under *valgrind*. The argument `tool` must be one of `m` (for memcheck), `c` (for callgrind), or `h` (for cachegrind). Valgrind output is written to a file under `valgrind_logs` as reported when *escript* terminates.

2.2.1 Notes

The **`run-escript`** script is generated at build time taking into account the `prelaunch`, `launcher`, and `postlaunch` settings passed to **`scons`**. This makes it possible to easily customize the script for different environments, such as batch systems (PBS, SLURM) and different implementations of MPI (Intel, SGI, OpenMPI, etc.).

2.3 Input and Output

When *MPI* is used on more than one process ($\texttt{nn} \cdot \texttt{np} > 1$) no input from the standard input is accepted. Standard output on any process other than the master process (`rank` = 0) will be silently discarded by default. Error output from any processor will be redirected to the node where **`run-escript`** has been invoked. If the -o option or **ESCRIPT_STDFILES** is set[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 ($\texttt{nn} \cdot \texttt{np} > 1$), the *MPI* process rank is appended to the file names. This is to avoid problems if processes are using a shared file system. Files which collect data that are global for all *MPI* processors are created by the process with *MPI* rank 0 only. Users should keep in mind that if the file system is not shared among the processes, then a file containing global information which is read by all processors needs to be copied to the local file system(s) before **`run-escript`** is invoked.

2.4 Hints for MPI Programming

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

When *MPI* is used on more than one process ($\texttt{nn} \cdot \texttt{np} > 1$) 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.

This has three main implications:

1. most arguments (`Data` excluded) should have the same values on all processors, e.g. `int`, `float`, `str` and `numpy` parameters.

2. the same operations will be called on all processors.

3. different processors may store different amounts of information.

With a few exceptions[6], values of types `int`, `float`, `str` and `numpy` returned by `esys.escript` will have the same value on all processors. If values produced by other modules are used as arguments, the user has to make sure that the argument values are identical on all processors. For instance, the usage of a random number generator to create argument values bears the risk that the value may depend on the processor.

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

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

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

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

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

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

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

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

CHAPTER
THREE

The `esys.escript` Module

3.1 Concepts

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

3.1.1 Function spaces

In order to understand what we mean by the term 'function space', consider that the solution of a partial differential equation (PDE) is a function on a domain Ω. 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 commonly used:

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

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

- `ReducedFunction(mydomain)`

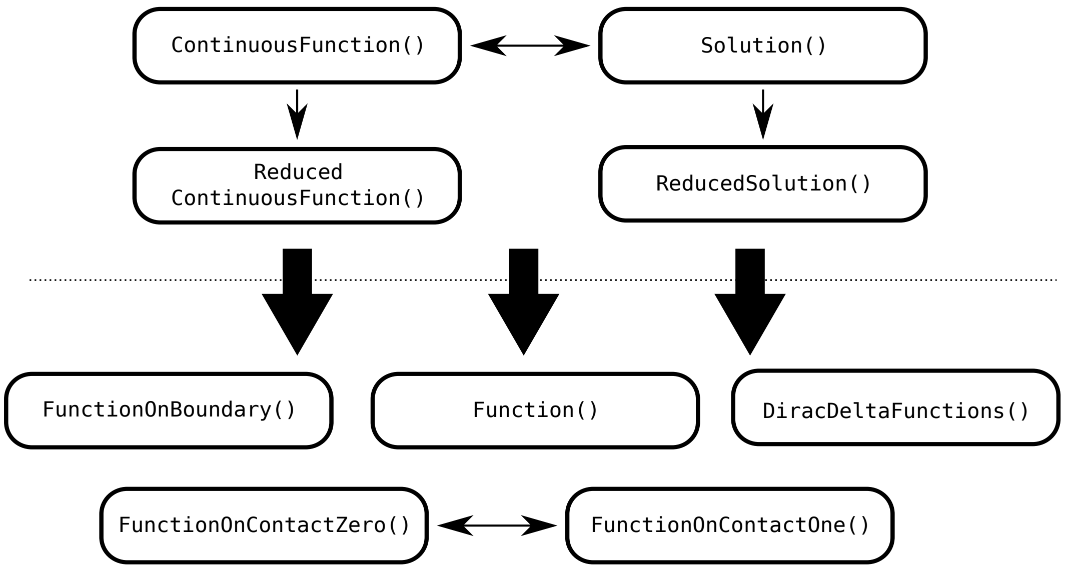


FIGURE 3.1: Dependency of function spaces in `esys.finley`. An arrow indicates that a function in the `FunctionSpace` at the starting point can be interpolated to the `FunctionSpace` of the arrow target. All function spaces above the dotted line can be interpolated to any of the function spaces below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

- `ReducedFunctionOnBoundary(mydomain)`
- `ReducedFunctionOnContact0(mydomain)`
- `ReducedFunctionOnContact1(mydomain)`

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

The reduced smoothness for a PDE solution is often used to fulfill the Ladyzhenskaya-Babuska-Brezzi condition [7] when solving saddle point problems, e.g. the Stokes equation. A discontinuity is a region within the domain across which functions may be discontinuous. The location of a discontinuity is defined in the `Domain` object. Figure 3.1 shows the dependency between the types of function spaces in `esys.finley` (other libraries may have different relationships).

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 a 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, for example when periodic boundary conditions are used in `esys.finley`, a solution fulfills periodic boundary conditions while a continuous function does not have to be periodic.

The concept of function spaces describes the properties of functions and allows abstraction from the actual representation of the function in the context of a particular application. For instance, in the FEM context a function of the general `FunctionSpace` type (written as *Function()* in Figure 3.1) is usually represented by its values at the element center, but in a finite difference scheme the edge midpoint of cells is preferred. By changing its function space you can use the same function in a Finite Difference scheme instead of Finite Element scheme. Changing the function space of a particular function will typically lead to a change of its representation. So, when seen as a general function, a continuous function which is typically represented by its values on the nodes of the FEM mesh or finite difference grid must be interpolated to the element centers or the cell edges, respectively. Interpolation happens automatically in `esys.escript` whenever it is required. The user needs to be aware that an interpolation is not always possible, see Figure 3.1 for `esys.finley`. An alternative approach to change the representation (=`FunctionSpace`) is projection, see Section 4.2.

3.1.2 `Data` Objects

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

The values of the function have a rank which gives the number of indices, and a shape defining the range of each index. The rank in `esys.escript` is limited to the range 0 through 4 and it is assumed that the rank and shape is the same for all data sample points. The shape of a `Data` object is a tuple (list) `s` of integers. The length of `s` is the rank of the `Data` object and the `i`-th index ranges between 0 and `s[i]` -1. For instance, a stress field has rank 2 and shape (d, d) where d is the number of spatial dimensions. 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` [3]. 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 2×3 matrix as an instance of `numpy.ndarray` filled with ones. The shape of the created `Data` object is taken from the shape of the array. In the second case, the creator converts the initial value, which is a list of lists, into a `numpy.ndarray` before creating the actual `Data` object.

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

- `Scalar(0, Function(mydomain))` is the same as `Data(0, Function(myDomain),(,))` (each value is a scalar), e.g. a temperature field
- `Vector(0, Function(mydomain))` is the same as `Data(0, Function(myDomain),(d,))` (each value is a vector), e.g. a velocity field
- `Tensor(0, Function(mydomain))` equals `Data(0, Function(myDomain), (d,d))`, e.g. a stress field
- `Tensor4(0,Function(mydomain))` equals `Data(0,Function(myDomain), (d,d,d,d))`, e.g. a Hook tensor field
- `ComplexScalar(0+0j, Function(mydomain))` is the same as `ComplexData(0+0j, Function(myDomain),(,))` (each value is a complex scalar), e.g. a temperature field
- `ComplexVector(0+0j, Function(mydomain))` is the same as `ComplexData(0+0j, Function(myDomain),(d,))` (each value is a complex vector), e.g. a velocity field
- `ComplexTensor(0+0j, Function(mydomain))` is the same as `ComplexData(0+0j, Function(myDomain), (d,d))`, e.g. a stress field
- `ComplexTensor4(0+0j,Function(mydomain))` is the same as `ComplexData(0+0j,Function(myDomain), (d,d,d,d))`, e.g. a Hook tensor field

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

`Data` objects can be manipulated by applying unary operations (e.g. cos, sin, log), and they can be combined point-wise by applying arithmetic operations (e.g. +, - ,* , /). We emphasize that `esys.escript` itself does not handle any spatial dependencies as it does not know how values are interpreted by the processing PDE solver library. However `esys.escript` invokes interpolation if this is needed during data manipulations. Typically, this occurs in binary operations when the 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 σ 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 δ_{ij} is the Kronecker symbol and λ and μ are the Lamé coefficients. The following function takes the displacement `u` and the Lamé 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 dimensionality of the domain on which the displacements are defined. The `kronecker(d)` call, returns the Kronecker symbol with indices i and j running from 0 to `d`-1. The `grad(u)` call, requires the displacement field `u` to be in the `Solution` or continuous `FunctionSpace`. The result `g` as well as the returned stress will be in the general `FunctionSpace`. If, for example, `u` is the solution of a PDE then `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` but with `g` the function `trace(g)` is in the general `FunctionSpace`. In the evaluation of the product `lam*trace(g)` we have different function spaces (on the nodes versus in the centers) and at first glance we have incompatible data. `esys.escript` converts the arguments into an appropriate function space according to Figure 3.1. In this example that means `esys.escript` sees `lam` as a function of the general `FunctionSpace`. In the context of FEM this means the nodal values of `lam` are interpolated to the element centers. The interpolation is automatic and requires no special handling.

3.1.3 Tagged, Expanded and Constant Data

Material parameters such as the Lamé 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 predominantly 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 to use tagged values for `lam` as shown in Figure 3.2 for the stress calculation discussed above:

```
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 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 Lamé coefficients, and then later switch to tagged Lamé coefficients without otherwise changing your *python* script. In short, you can use the same script for models with different domains and different types of input data.

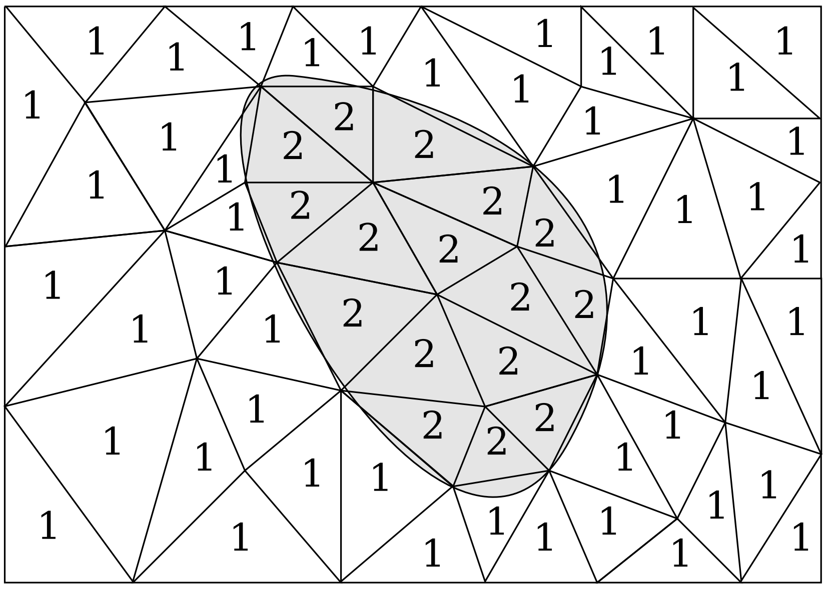


FIGURE 3.2: Element Tagging. A rectangular mesh over a region with two rock types *white* and *gray* is shown. The number in each cell refers to the major rock type present in the cell (1 for *white* and 2 for *gray*).

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

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

3.1.4 Saving and Restoring Simulation Data

`Data` objects can be written to disk files with the `dump` method and read back using the `load` method, both of which use the HDF5[11] file format. Use these to save data for checkpoint/restart or simply to save and reuse data that was expensive to compute. For instance, to save the coordinates of the data points of a continuous `FunctionSpace` to the file `x.nc` use

```
x=ContinuousFunction(mydomain).getX()
x.dump("x.h5")
mydomain.dump("dom.h5")
```

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

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

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

The function space of the `Data` is stored in `x.nc`. If the `Data` object is expanded, the number of data points in the file and of the `Domain` for the particular `FunctionSpace` must match. Moreover, the ordering of the values is checked using the reference identifiers provided by the `FunctionSpace` on the `Domain`. In some cases, data points will be reordered so be aware and confirm that you get what you wanted.

A more flexible way of saving and restoring `esys.escript` simulation data is through an instance of the `DataManager` class. It has the advantage of allowing to save and load not only a `Domain` and `Data` objects but also other values[1] you compute in your simulation script. Further, `DataManager` objects can simultaneously create files for visualization so no extra calls to `saveVTK` etc. are needed.

The following example shows how the `DataManager` class can be used. For an explanation of all member functions and options see the class reference Section 3.2.9.

```
from esys.escript import DataManager, Scalar, Function
from esys.finley import Rectangle

dm = DataManager(formats=[DataManager.RESTART, DataManager.VTK])
if dm.hasData():
  mydomain=dm.getDomain()
  val=dm.getValue("val")
  t=dm.getValue("t")
  t_max=dm.getValue("t_max")
else:
  mydomain=Rectangle()
  val=Function(mydomain).getX()
  t=0.
  t_max=2.5

while t<t_max:
  t+=.01
  val=val+t/2
  dm.addData(val=val, t=t, t_max=t_max)
  dm.export()
```

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

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

3.2 `esys.escript` Classes

3.2.1 The `Domain` class

class Domain()

[1]The *python pickle* module is used for other types.

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 number of spatial dimensions of the `Domain`.

dump(filename)
writes the `Domain` to the file `filename` using the *netCDF* file format.

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

getNumpyX()
returns the locations in the `Domain` as a `numpy` ndarray. The `FunctionSpace` of the returned `Data` object is chosen by the `Domain` implementation. Typically it will be in the continuous `FunctionSpace`.
Note that it is necessary to load `numpy` first in the escript.

setX(newX)
assigns new locations to the `Domain`. `newX` has to have shape $(d,)$ where d is the spatial dimensionality of the domain. Typically `newX` must be in the continuous `FunctionSpace` but the space actually to be used depends on the `Domain` implementation. Not all domain families support setting locations.

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

getSize()
returns the local sample size, i.e. the element diameter, as a `Data` object.

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

getTag(tag_name)
returns the tag associated with the tag name `tag_name`.

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

__eq__(arg)
(*python* == operator) returns *True* if the `Domain` `arg` describes the same domain, *False* otherwise.

__ne__(arg)
(*python* != operator) returns *True* if the `Domain` `arg` does not describe the same domain, *False* otherwise.

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

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

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

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

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

3.2.2 The `FunctionSpace` class

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

The following methods are available:

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

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

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

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

getDomain()
: returns the `Domain` of the `FunctionSpace`.

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

__eq__(arg)
: (*python* == operator) returns *True* if the `FunctionSpace` `arg` describes the same function space, *False* otherwise.

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

__str__()
: (*python* `str()` function) returns a string representation of the `FunctionSpace`.

The following functions provide generators for `FunctionSpace` objects:

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

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

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

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

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

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

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

3.2.3 The `Data` Class

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

Expression	Description
`+arg`	identical to `arg`
`-arg`	negation of `arg`
`arg0+arg1`	adds `arg0` and `arg1`
`arg0*arg1`	multiplies `arg0` and `arg1`
`arg0-arg1`	subtracts `arg1` from `arg0`
`arg0/arg1`	divides `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 `arg0`'s `FunctionSpace`. 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+=arg1`	adds `arg1` to `arg0`
`arg0*=arg1`	multiplies `arg0` by `arg1`
`arg0-=arg1`	subtracts `arg1` from`arg0`
`arg0/=arg1`	divides `arg0` by `arg1`
`arg0**=arg1`	raises `arg0` to the power of `arg1`

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

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

Rank of `arg`	Slicing expression	shape of returned and assigned object
0	no slicing	N/A
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)

Let `s` be the shape of `arg`, then

$$\begin{aligned}0 \le \texttt{l0} \le \texttt{u0} \le \texttt{s[0]},\\ 0 \le \texttt{l1} \le \texttt{u1} \le \texttt{s[1]},\\ 0 \le \texttt{l2} \le \texttt{u2} \le \texttt{s[2]},\\ 0 \le \texttt{l3} \le \texttt{u3} \le \texttt{s[3]}.\end{aligned}$$

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]
```

3.2.4 Generation of `Data` objects

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

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

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

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

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

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

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

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

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

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

ComplexData(value, what=FunctionSpace(), expanded=*False*)

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

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

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

ComplexData()

creates an empty `Data` object with complex values (i.e. with memory allocated to store a complex number). The empty `Data` object is used to indicate that an argument is not present where a `Data` object is required.

ComplexScalar(value=0.+0.j, what=FunctionSpace(), expanded=*False*)

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

ComplexData(value=0.+0.j, what=FunctionSpace(), expanded=*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 initialized with complex `value`, a double precision quantity. If `expanded` is *True* the `Data` object is represented in expanded form.

ComplexTensor(value=0.+0.j, what=FunctionSpace(), expanded=*False*)

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

ComplexTensor3(value=0.+0.j, what=FunctionSpace(), expanded=*False*)

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

ComplexTensor4(value=0.+0.j, what=FunctionSpace(), expanded=*False*)

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

load(filename, domain)

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

3.2.5 Generating random `Data` objects

A `Data` object filled with random values can be produced using the `RandomData` function. By default values are drawn uniformly at random from the interval $[0, 1]$ (i.e. including end points). The function takes a shape for the data points and a `FunctionSpace` for the new `Data` as arguments. For example:

```
from esys.finley import *
from esys.escript import *

domain=Rectangle(11,11)
fs=ContinuousFunction(domain)
d=RandomData((), fs)
```

would result in `d` being filled with scalar random data since `()` is an empty tuple.

```
from esys.finley import *
from esys.escript import *

domain=Rectangle(11,11)
fs=ContinuousFunction(domain)
d=RandomData((2,2), fs)
```

would give `d` the same number of data points, but each point would be a 2×2 matrix instead of a scalar.

By default, the seed used to generate the random values will be different each time. If required, you can specify a seed to ensure the same sequence is produced.

```
from esys.finley import *
from esys.escript import *

seed=-17171717
domain=Brick(10,10,10)
fs=Function(domain)
d=RandomData((2,2), fs, seed)
```

The `seed` can be any integer value[2] but 0 is special. A seed of zero will cause `esys.escript` to use a different seed each time. Also, note that the mechanism used to produce the random values could be different in different releases.

Note for MPI users: *Even if you specify a seed, you will only get the same results if you are running with the same number of ranks. If you change the number of ranks, you will get different values for the same seed.*

3.2.5.1 Smoothed randoms

The `esys.ripley` domains (see Chapter 6) support generating random scalars which are smoothed using Gaussian blur. To use this, you need to supply the radius of the filter kernel (in elements) and the `sigma` value used in the filter. For example:

```
from esys.ripley import *
from esys.escript import *

fs=ContinuousFunction(Rectangle(11,11, d1=2,d0=2))
d=RandomData((), fs, 0, ('gaussian', 1, 0.5))
```

will use a filter that uses the immediate neighbours of each point with a sigma value of 0.5. The random values will be different each time this code is executed due to the seed of 0.

Ripley's Gaussian smoothing has the following requirements:

1. If *MPI* is in use, then each rank must have at least 5 elements in it *in each dimension*. This value increases as the radius of the blur increases.

2. The data being generated must be scalar. (You can generate random data objects for `esys.ripley` domains with whatever shape you require, you just can't smooth them unless that shape is scalar).

[2] which can be converted to a C++ long

An exception will be raised if either of these requirements is not met.

The components of the matrix used in the kernal for the 2D case are defined[27] by:

$$G(x,y) = \frac{1}{2\pi\sigma^2} e^{-\frac{x^2+y^2}{2\sigma^2}}$$

For the 3D case, we use:

$$G(x,y) = \frac{1}{(\sqrt{2\pi\sigma^2})^3} e^{-\frac{x^2+y^2+z^2}{2\sigma^2}}$$

All distances (x,y,z) refer to the number of points from the centre point. That is, the closest neighbours have at least one distance of 1, the next “ring” of neighbours have at least one 2 and so on. The matrix is normalised before use.

3.2.6 `Data` methods

These are the most frequently used methods of the `Data` class. A complete list of methods can be found in the reference guide, see `http://esys.geocomp.uq.edu.au/docs.html`.

getFunctionSpace()
returns the `FunctionSpace` of the object.

getDomain()
returns the `Domain` of the object.

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

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

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

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

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

__str__()
returns a string representation of the object.

3.2.7 Functions of `Data` objects

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

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

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

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

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

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

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

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

unitVector(i,d)
returns a rank-1 Data object 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 point the minimum value over all components.

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

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

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

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 this is

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

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

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

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. For instance, in the case of a rank-2 Data object this is

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

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

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

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. For instance, if `a` is a rank-4 `Data` object, `axis0=1` and `axis1=2`, the result is

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

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` so that

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

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

eigenvalues(a)
returns the eigenvalues of `a` so that

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

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

eigenvalues_and_eigenvectors(a)
returns the eigenvalues and eigenvectors of `a`.

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

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

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

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

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

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

at all data sample points.

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

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

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.

cross(a0, a1)

returns the cross product of `a0` and `a1`.

inner(a0, a1)

returns the inner product of `a0` and `a1`. For instance in the case of a rank-2 `Data` object:

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

and for a rank-4 `Data` object:

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

matrix_mult(a0, a1)

returns the matrix product of `a0` and `a1`. If `a1` is a rank-1 `Data` object this is

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

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

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

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 a rank-1 `Data` object this is

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

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

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

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 a rank-2 `Data` object this is

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

outer(a0, a1)

returns the outer product of `a0` and `a1`. For instance, if both, `a0` and `a1` is a rank-1 `Data` object then

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

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

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

tensor_mult(a0, a1)

returns the tensor product of `a0` and `a1`. If `a1` is a rank-2 `Data` object this is

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

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

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

transposed_tensor_mult(a0, a1)

returns the tensor product of the transposed of `a0` and `a1`. The function is equivalent to `tensor_mult(transpose(a0),a1)`. If `a1` is a rank-2 `Data` object this is

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

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

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

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 a rank-2 `Data` object this is

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

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

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

grad(a[, where=None])

returns the gradient of `a`. If `where` is present the gradient will be calculated in the `FunctionSpace` `where`, otherwise a default `FunctionSpace` is used. In case that `a` is a rank-2 `Data` object one has

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

curl(a[, where=None])

returns the curl of `a`. If `where` is present the curl will be calculated in the `FunctionSpace` `where`, otherwise a default `FunctionSpace` is used.

div(a[, where=None])

returns the divergence of `a`:

$$\texttt{div(a)=trace(grad(a),where)} \tag{3.33}$$

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 in continuous `FunctionSpace` it is

$$\texttt{integrate(a)}\,[i,j] = \int_\Omega \texttt{a}\,[i,j]\; d\Omega \tag{3.34}$$

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

$$\texttt{integrate(a,where=FunctionOnBoundary(a.getDomain()))}\,[i,j] = \int_{\partial\Omega} a\,[i,j]\;ds \quad (3.35)$$

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`.

jump(a[, domain=None])

returns the jump of `a` over the discontinuity in its domain or if `Domain` `domain` is present in `domain`.

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

L2(a)

returns the L^2-norm of `a` in its `FunctionSpace`. This is

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

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

sin(a)

applies the sine function to `a`.

cos(a)

applies the cosine function to `a`.

tan(a)

applies the tangent function to `a`.

asin(a)

applies the arc (inverse) sine function to `a`.

acos(a)

applies the arc (inverse) cosine function to `a`.

atan(a)

applies the arc (inverse) tangent function to `a`.

sinh(a)

applies the hyperbolic sine function to `a`.

cosh(a)

applies the hyperbolic cosine function to `a`.

tanh(a)

applies the hyperbolic tangent function to `a`.

asinh(a)

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

acosh(a)

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

atanh(a)

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

exp(a)
applies the exponential function to a.

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

log(a)
takes the natural logarithm of a.

log10(a)
takes the base-10 logarithm of a.

sign(a)
applies the sign function to a. The result 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 a times rtol is used.

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

3.2.8 Interpolating Data

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

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

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

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

First we produce an interpolation table:

```
sine_table=[0, 0.70710678118654746, 1, 0.70710678118654746, 0,
           -0.70710678118654746, -1, -0.70710678118654746, 0]
```

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

```
  numslices=len(sine_table)-1
  minval=0.
  maxval=1.
  step=sup(maxval-minval)/numslices
```

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

Now we produce our new `Data` object:

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

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

Now consider a 2D example. We will interpolate from a plane where $\forall x, y \in [0,9] : (x,y) = x + y \cdot 10$.

```
from esys.escript import whereZero
table2=[]
for y in range(0,10):
      r=[]
      for x in range(0,10):
         r.append(x+y*10)
      table2.append(r)
xstep=(maxval-minval)/(10-1)
ystep=(maxval-minval)/(10-1)

xmin=minval
ymin=minval

result2=interpolateTable(table2, x2, (xmin, ymin), (xstep, ystep), toobig)
```

We can check the values using `whereZero`. For example, for $x = 0$:

```
print(result2*whereZero(x[0]))
```

Finally let us look at a 3D example. Note that the parameter tuples should be (x, y, z) but that in the interpolation table, x is the innermost dimension.

```
b=Brick(n,n,n)
x3=b.getX()
toobig=1000000

table3=[]
for z in range(0,10):
   face=[]
   for y in range(0,10):
      r=[]
      for x in range(0,10):
         r.append(x+y*10+z*100)
      face.append(r)
   table3.append(face);

zstep=(maxval-minval)/(10-1)

zmin=minval

result3=interpolateTable(table3, x3, (xmin, ymin, zmin),
     (xstep, ystep, zstep), toobig)
```

3.2.8.1 Non-uniform Interpolation

Non-uniform interpolation is also supported for the one dimensional case.

```
Data.nonuniformInterpolate(in, out, check_boundaries)
Data.nonuniformSlope(in, out, check_boundaries)
```

Will produce a new `Data` object by mapping the given `Data` object through the user-defined function specified by `in` and `out`. The ...Interpolate version gives the value of the function at the specified point and the ...Slope version gives the slope at those points. The check_boundaries boolean argument specifies what the function should do if the `Data` object contains values outside the range specified by the `in` parameter. If the argument is `False`, then those datapoints will be interpolated to the value of the edge they are closest to (or assigned a slope of zero). If the argument is `True`, then an exception will be thrown if out of bounds values are detected. Note that the values given by the `in` parameter must be monotonically increasing.
For example:
If `d` contains the values `{1,2,3,4,5}`, then

```
d.nonuniformInterpolate([1.5, 2, 2.8, 4.6], [4, 5, -1, 1], False)
```

would produce a `Data` object containing `{4, 5, -0.7777, 0.3333, 1}`.
A similar call to `nonuniformSlope` would produce a `Data` object containing `{0, 2, 1.1111, 1.1111, 0}`.

3.2.9 The `DataManager` Class

The `DataManager` class can be used to conveniently add checkpoint/restart functionality to `esys.escript` simulations. Once an instance is created `Data` objects and other values can be added and dumped to disk by a single method call. If required the object can be set up to also save the data in a format suitable for visualization. Internally the `DataManager` interfaces with `esys.weipa` for this.

class DataManager(formats=[RESTART], work_dir=".", restart_prefix="restart", do_restart=*True*)
initializes a new `DataManager` object which can be used to save, restore and export simulation data in a number of formats. All files and directories saved or restored by this object are located under the directory specified by `work_dir`. If `RESTART` is specified in `formats`, the `DataManager` will look for directories whose name starts with `restart_prefix`. In case `do_restart` is *True*, the last of these directories is used to restore simulation data while all others are deleted. If `do_restart` is *False*, then all of those directories are deleted. The `restart_prefix` and `do_restart` parameters are ignored if `RESTART` is not specified in `formats`.
Valid values for the `formats` parameter are:

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

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

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

VTK
exports simulation data in the *VTK* file format.

The `DataManager` class has the following methods:

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

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

hasData()
returns *True* if the manager has loaded simulation data for a restart.

getDomain()
returns the domain as recovered from a restart.

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

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

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

setTime(time)
sets the simulation time stamp. This floating point number is stored in the metadata of exported data but not used by `RESTART`.

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

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

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

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

3.2.10 Saving Data as CSV

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

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

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

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

The names of the keyword parameters form the names of columns in the output. If the data objects are over different function spaces, then `saveDataCSV` will attempt to interpolate to a common function space. If this is not possible, then an exception is raised.

Output can be restricted using a scalar mask as follows:

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

This command will only output those rows which correspond to to positive values of `myscalar`. Some aspects of the output can be tuned using additional parameters:

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

- `refid` – specifies that the output should include the reference IDs of the elements or nodes
- `append` – specifies that the output should be written to the end of an existing file
- `sep` – defines the separator between fields
- `csep` – defines the separator between components in the header line. For example between the components of a matrix.

The above command would produce output like this:

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

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

3.2.11 Converting `Data` to a Numpy Array

`Data` objects can be converted into a numpy structured array using the commands `getNumpy` and `convertNumpy`.

3.2.11.1 getNumpy

If `mydata1` and `mydata2` are scalar `Data`, then the command

```
a = getNumpy(U=mydata1, V=mydata2)
```

will return a dictionary containing two structured ndarrays with the names '*U*' and '*V*'.

```
a.get('U') = [1.0000000e+0, 2.0000000e-1, ...
a.get('V') = [2.0000000e+0, 3.0000000e-1, ...
```

Any number of `Data` objects can be passed to `getNumpy`. These objects can be scalar, vector or tensor `Data` objects. The names of the keyword parameters form the names of the returned arrays. If the data objects are over different function spaces, then `getNumpy` will attempt to interpolate to a common function space. If this is not possible, then an exception is raised.

Output can be restricted using a scalar mask as follows:

```
a = getNumpy(U=mydata1, V=mydata2, W=mydata3, mask=myscalar)
```

This command will only output those rows which correspond to to positive values of `myscalar`.

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

3.2.11.2 convertNumpy

`Data` objects can also be converted into a numpy structured array using the command `convertNumpy`. If `mydata1` is a `Data` object, then the command

```
a = convertNumpy(mydata1)
```

will return a structured ndarray containing all of the data in `mydata1`. Unlike `getNumpy`, this function does not support the use of masks and does not use MPI.

3.2.12 The `Operator` Class

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

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

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

resetValues()
: resets all entries in the operator.

solve(rhs)
: returns the solution `u` of: operator * `u` = `rhs`.

of(u)
: applies the operator to the `Data` object `u`, i.e. performs a matrix-vector multiplication.

saveMM(fileName)
: saves the object to a Matrix Market format file with name `fileName`, see
`http://math.nist.gov/MatrixMarket`

3.3 Physical Units

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

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

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

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

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

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

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

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

can be used print the pressure in units of atmosphere.

The following is an incomplete list of supported physical units:

km
: unit of kilometer

m
: unit of meter

cm
unit of centimeter

mm
unit of millimeter

sec
unit of second

minute
unit of minute

h
unit of hour

day
unit of day

yr
unit of year

gram
unit of gram

kg
unit of kilogram

lb
unit of pound

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 degrees Kelvin

Celsius
: unit of degrees Celsius

Fahrenheit
: unit of degrees Fahrenheit

Supported unit prefixes:

Yotta
: prefix yotta = 10^{24}

Zetta
: prefix zetta = 10^{21}

Exa
: prefix exa = 10^{18}

Peta
: prefix peta = 10^{15}

Tera
: prefix tera = 10^{12}

Giga
: prefix giga = 10^{9}

Mega
: prefix mega = 10^{6}

Kilo
: prefix kilo = 10^{3}

Hecto
: prefix hecto = 10^{2}

Deca
: prefix deca = 10^{1}

Deci
: prefix deci = 10^{-1}

Centi
: prefix centi = 10^{-2}

Milli
: prefix milli = 10^{-3}

Micro
: prefix micro = 10^{-6}

Nano
: prefix nano = 10^{-9}

Pico
prefix pico = 10^{-12}

Femto
prefix femto = 10^{-15}

Atto
prefix atto = 10^{-18}

Zepto
prefix zepto = 10^{-21}

Yocto
prefix yocto = 10^{-24}

3.4 Utilities

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

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

The following methods are available:

close()
closes the file.

flush()
flushes the internal buffer to disk.

write(txt)
writes string `txt` to the file. Note that a newline is not added.

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

closed
this member is *True* if the file is closed.

mode
holds the access mode.

name
holds the file name.

newlines
holds the line separator.

The following additional functions are available in the `esys.escript` module:

setEscriptParamInt(name,value)

assigns the integer value `value` to the internal Escript parameter `name`. This should be considered an advanced feature and it is generally not required to call this function. One parameter worth mentioning is `name` ="TOO_MANY_LINES" which affects the conversion of `Data` objects to a string. If more than `value` lines would be created, a condensed format is used instead which reports the minimum and maximum values and general information about the `Data` object rather than all values.

getEscriptParamInt(name)
returns the current value of internal Escript parameter `name`.

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

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

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

MPIBarrierWorld()
performs a barrier synchronization across all processes within the **MPI_COMM_WORLD** process group.

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

3.5 Lazy Evaluation of Data

Constant and Tagged representations of Data are relatively small but Expanded[3] are larger and will not entirely fit in CPU cache.

Escript's lazy evaluation features record operations performed on Data objects but do not actually carry them out until the Data is "resolved".

Consider the following code:

```
from esys.escript import *
from esys.finley import Rectangle
x=Rectangle(3,3)
x=Rectangle(3,3).getX()
c=Data((1.5, 1), x.getFunctionSpace())
t=Data(((1,1),(0,1)), x.getFunctionSpace())
t.tag()
```

The variables `c`, `t`, `x` are stored as `constant`, `tagged` and `expanded` Data respectively. Printing those variables will show the values stored (or if we were to use a larger Rectangle, a summary).

```
v = matrix_mult(t,x) + c
print(v.isExpanded())
print(v)
```

Will output `True` followed by all of the values for `v`. Now we'll introduce lazy evaluation:

```
xx = x.delay()
print(xx.isExpanded(), xx.isLazy())
print(x.isExpanded(), x.isLazy())
print(xx)
```

The first print will show that `xx` is not considered to be "expanded", while the second print shows that `x` is unaffected. The last print will produce something like:

```
Lazy Data: [depth=0] E@0x55ed512ad760
```

[3]Separate values stored for each point of the FunctionSpace.

The `E` before the `@` shows that this lazy Data is wrapping "expanded" Data. Calling `.delay()` on constant or tagged Data results in `C@...` and `T@...` respectively.

If an input to an operation is lazy, then the result will be lazy as well[4]:

```
res = matrix_mult(t,-xx) + c
print(res)
```

Will produce:

```
Lazy Data: [depth=3] (prod(T@0x..., neg(E@...)) + C@0x...)
```

Depth indicates the largest number of operators from the top of the expression to the bottom.

To actually find the value of this lazy Data object, we need to resolve it:

```
res.resolve()
```

Note that `resolve()` doesn't return a new object, but transforms the object it is called on. Printing, `res` now will show the values at each point.

3.5.1 Lazyness and non-expanded Data

While it is possible to call delay on constant or tagged Data, escript will not build expressions consisting solely of such Data.

```
cx=c.delay()
res=cx+cx
print(res)
```

would output:

```
Lazy Data: [depth=0] C@0x55ed512cc7c0
# Not
Lazy Data: [depth=1] (C@0x... + C@0x...)
```

3.5.2 When to resolve

You are never *required* to manually resolve lazy Data in `escript`. Any operations which need the actual values of an expression will either

- compute the values without resolving the whole Data object at once (solvers assembling FEM matrices)
- resolve the data automatically (everthing else)

Escript will automatically resolve lazy Data:

1. If a matrix inversion operation is applied to the Data.
2. If the expression tree becomes too deep[5].

Note, the second point is important when writing loops like this:

```
# x is initial guess
while err > tol:
  construct PDE coefficients involving x
  solve PDE
  calculate err
  update x
```

After a few iterations of the loop, `x` may be something like `x=F(F(F(F(originalX))))`. So it will probably be better to `resolve` `x` at the end of each loop iteration. Alternatively, if `x` is included in many expressions in the loop, it may be better to resolve it earlier.

[4]Matrix inverse is an exception to this.

[5]At time of writing, this threshold is somewhat arbitrarily set at `depth>9`, but this is configurable.

3.5.3 Options for using lazy evaluation

There are two ways to enable lazy evaluation:

1. Any escript script can make use of lazy evaluation by `delay()`-ing one of its expanded Data variables. Any expressions including that delayed variable (directly or indirectly) will be lazy until resolved.

2. Setting the `AUTOLAZY` parameter for `escript` to `1`. In this case, most escript operation which would normally produce extended Data, will produce lazy Data instead. In general, this option is not recommended for two reasons:

   - AUTOLAZY uses the `setEscriptParamInt()` which is not guaranteed to have continued support.
   - Making everything lazy instead of just more complex objects is not likely to give significant efficiency improvements.

3.5.4 When to use lazy evaluation?

Exactly when using lazy evaluation will be more efficient is still an open question. When the objects being manipulated are large (eg 4-Tensors in Drucker-Prager), significant memory and runtime improvements can be achieved. See [5].

Our best advice is to experiment with it.

CHAPTER

FOUR

The `esys.escript.linearPDEs` Module

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 Ω defined through a `Domain` object. In the following Γ denotes the boundary of the domain Ω and n denotes the outer normal field on Γ.

For a single PDE with a solution that has a single component the linear PDE is defined in the following form [1]

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

$u_{,j}$ denotes the derivative of u with respect to the j-th spatial direction. Einstein's summation convention, i.e. summation over indexes appearing twice in a term of a sum, is used in this chapter. $y^{dirac}(p)$ represent a nodal source term at point p, cf. $y^{dirac}(p)$ and similar $d^{dirac}(p)$ define Dirac delta-function terms. 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. d^{dirac} A is a rank-2 `Data` object, B, C and X are each a rank-1 `Data` object and D and Y are scalars. y^{dirac} and d^{dirac} are each scalars in the Dirac delta-function `FunctionSpace`. The following natural boundary conditions are considered on Γ:

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

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

[1]This PDE system can be written in the equivalent form,

$$-\nabla\cdot(\mathbf{A}\nabla u) - \nabla\cdot(\mathbf{B}\,u) + \mathbf{C}\cdot\nabla u + \mathbf{D}\,u + \sum_p \delta(p)u(p) = -\nabla\cdot\mathbf{X} + \mathbf{Y} + \sum_p \delta(p),$$
$$\hat{\mathbf{n}}\cdot(\mathbf{A}\nabla u + \mathbf{B}\,u) + \mathbf{d}\,u = \hat{\mathbf{n}}\cdot\mathbf{X} + \mathbf{y},$$

where bold font denotes a data object and $\hat{\mathbf{n}}$ denotes a unit vector normal. The weak form for this is

$$\int_\Omega (\nabla v)\cdot(\mathbf{A}\nabla u) + (\nabla v)\cdot(\mathbf{B}\,u) + v\mathbf{C}\cdot\nabla u + \mathbf{D}\,vu + \sum_p \delta(p)u(p)v\; d\Omega = \int_\Omega (\nabla v)\cdot\mathbf{X} + \mathbf{Y}v + \sum_p \delta(p)v\; d\Omega.$$

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

$$-(A_{ijkl}u_{k,l})_{,j} - (B_{ijk}u_k)_{,j} + C_{ikl}u_{k,l} + D_{ik}u_k + \sum_p d^{dirac}_{ik}(p)\; u_i(p) = -X_{ij,j} + Y_i + \sum_p\; y^{dirac}_i(p)\; . \tag{4.4}$$

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

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

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 a rank-1 `Data` object. Notice that not necessarily all components must have a constraint at all locations. An example for a system of PDEs is shown in the elastic deformation example in section 1.5

`LinearPDE` also supports solution discontinuities over a contact region $\Gamma^{contact}$ in the domain Ω. To specify the conditions across the discontinuity we are using the generalised flux J[2] which in the case of a system of PDEs and several components of the solution, is defined as

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

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

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

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

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

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 the case of a single PDE and a single component solution the contact condition takes the form

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

In this case the coefficient $d^{contact}$ and $y^{contact}$ are each scalar `Data` objects both in either the contact `FunctionSpace` on side 0 or the 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 to the scalar case Equation (4.11) now the coefficients D, d and $d^{contact}$ have to be inspected.

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

[2]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 $q_{,i} = -\kappa T_{,i}$ (κ is the diffusivity) which differs from the definition used here by the sign. This needs to be kept in mind when defining natural boundary conditions.

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

We refer to Chapter 1 for more details.

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

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

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

Typically, a negative value for a diagnostic variable indicates that it is undefined. For more details on `SolverOptions` and Trilinos see chapter Reference [9].

4.1.1 Classes

The module `esys.escript.linearPDEs` provides an interface to define and solve linear partial differential equations within `esys.escript`. The module `esys.escript.linearPDEs` does not provide any solver capabilities in itself but hands the PDE over to the PDE solver library defined through the `Domain` of the PDE, e.g. `esys.finley`. The general interface is provided through the `LinearPDE` class. The `Poisson` class which is also derived form the `LinearPDE` class can 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.geocomp.uq.edu.au/docs.html`.

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

4.1.2.1 `LinearPDE` methods

setValue([A][, B], [, C][, D] [, X][, Y] [, d][, y] [, d_contact][, y_contact] [, d_dirac][, y_dirac] [, q][, r],)

assigns new values to coefficients. By default all values are assumed to be zero[3]. If the new coefficient value is not a `Data` object, it is converted into a `Data` object in the appropriate `FunctionSpace`.

getCoefficient(name)

returns the value assigned to coefficient `name`. If `name` is not a valid name an exception is raised.

getShapeOfCoefficient(name)

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

getFunctionSpaceForCoefficient(name)

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

setDebugOn()

switches on debug mode so more diagnostic messages will be printed.

setDebugOff()

switches off debug mode.

getSolverOptions()

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

setSolverOptions([options=None])

sets the solver options for solving the PDE. If argument `options` is present it must be a `SolverOptions` class object, see Section 4.3 for details. Otherwise the solver options are reset to the default.

isUsingLumping()

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

getDomain()

returns the `Domain` of the PDE.

getDim()

returns the number of spatial dimensions of the PDE.

getNumEquations()

returns the number of equations.

getNumSolutions()

returns the number of components of the solution.

checkSymmetry(verbose=*False*)

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

getFlux(u)

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

isSymmetric()

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

returns *True* if the PDE has been indicated to be symmetric, *False* otherwise.

setSymmetryOn()
indicates that the PDE is symmetric which enables the use of certain solvers and can potentially speed up the solver.

setSymmetryOff()
indicates that the PDE is not symmetric.

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

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

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

getRightHandSide()
returns the right hand side of the PDE as a `Data` object.

getSystem()
returns the `Operator` and right hand side of the PDE as a tuple.

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

4.1.3 The `Poisson` Class

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

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

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

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

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

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

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

4.1.5 The `Lame` Class

The `Lame` class defines a Lamé equation problem

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

with natural boundary conditions

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

and constraint

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

μ, λ have to be a scalar `Data` object in the general `FunctionSpace`, F has to be a vector `Data` object in the general `FunctionSpace`, σ 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 Lamé equation on the `Domain` domain. `Lame` is derived from `LinearPDE`.

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

4.2 Projection

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

$$u = Y \tag{4.26}$$

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

As projection is often used in simulations `esys.escript` provides an easy to use class `Projector` which is part of the `esys.escript.pdetools` module. The following script demonstrates the usage of the class to project the piecewise constant function ($= 1$ for $x_0 \geq 0.5$ and $= -1$ for $x_0 < 0.5$) to a function with the reduced solution `FunctionSpace` attribute (default target):

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

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

The following script illustrates how to evaluate the normal stress on the boundary from a given displacement field `u`:

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

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

This class defines a projector on the domain `domain`. If `reduce` is set to *True* the projection will be returned as a reduced solution `FunctionSpace Data` object. Otherwise the solution `FunctionSpace` representation is returned. If `reduce` is set to *True* lumping is used when the Equation (4.26) is solved, otherwise the standard PDE solver is used. Notice, that lumping requires significantly less computation time and memory. The class is callable.

getSolverOptions()

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

getValue(input_data)

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

4.3 Solver Options

class SolverOptions()

This class defines the solver options for a linear or non-linear solver. The option also supports the handling of diagnostic information.

getSummary()

returns a string reporting the current settings.

getName(key)

returns the name as a string of a given key.

setSolverMethod(method)

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. Note that SolverOptions needs to be

imported from linearPDEs and is not the same as the object returned by pde.getSolverOptions(). The value of `method` must be one of the constants:
`SolverOptions.DEFAULT` – use default solver depending on other options
`SolverOptions.BICGSTAB` – Biconjugate Gradient Stabilized iterative method
`SolverOptions.CGLS` – Conjugate Gradient with Least Squares method
`SolverOptions.CGS` – Conjugate Gradient Square method
`SolverOptions.CHOLEVSKY` – Direct solver based on LDLT factorization
`SolverOptions.CR` – Conjugate Residual method
`SolverOptions.DIRECT` – use a direct solver if available
`SolverOptions.DIRECT_SUPERLU` – use a direct SUPERLU solver
`SolverOptions.DIRECT_TRILINOS` – use default TRILINOS default solver (KLU2) (chapter Reference [9])
`SolverOptions.GMRES` – Gram-Schmidt minimum residual method
`SolverOptions.HRZ_LUMPING` – Matrix lumping using the HRZ approach (section Reference [1.4])
`SolverOptions.ITERATIVE` – use a suitable iterative solver
`SolverOptions.LSQR` – Least squares based LSQR solver
`SolverOptions.LUMPING` – Matrix lumping
`SolverOptions.MINRES` – Minimum Residual method
`SolverOptions.NONLINEAR_GMRES` – restarted GMRES for nonlinear systems
`SolverOptions.PCG` – Preconditioned Conjugate Gradient method
`SolverOptions.PRES20` – GMRES with restart after 20 steps and truncations after 5 residuals
`SolverOptions.ROWSUM_LUMPING` – Matrix lumping using row sum
`SolverOptions.TFQMR` – Transpose Free Quasi Minimum Residual method.
Not all packages support all solvers. It can be assumed that a package makes a reasonable choice if it encounters an unknown solver. See Table Reference [5.2] for the solvers supported by `esys.finley`.

getSolverMethod()

returns the key of the solver method to be used.

setPreconditioner(preconditioner)

sets the preconditioner to be used. The value of `preconditioner` must be one of the constants:
`SolverOptions.AMG` – Algebraic Multi Grid
`SolverOptions.GAUSS_SEIDEL` – Gauss-Seidel
`SolverOptions.ILU0` – Incomplete LU-factorization with no fill-in
`SolverOptions.ILUT` – Incomplete LU-factorization with fill-in
`SolverOptions.JACOBI` – Jacobi preconditioner
`SolverOptions.NO_PRECONDITIONER` – do not apply a preconditioner
`SolverOptions.RILU` – relaxed ILU0.
Not all packages support all preconditioners. It can be assumed that a package makes a reasonable choice if it encounters an unknown preconditioner. See Table Reference [5.3] for the preconditioners supported by `esys.finley`.

getPreconditioner()

returns the key of the preconditioner to be used.

setPackage(package)

sets the solver package to be used as a solver. The value of `package` must be one of the constants:
`SolverOptions.DEFAULT` – choose a default depending on other options
`SolverOptions.CUSP` – CUDA sparse linear algebra package
`SolverOptions.MKL` – Intel MKL direct solver
`SolverOptions.PASO` – built-in PASO solver library
`SolverOptions.TRILINOS` – Trilinos solver package (chapter Reference [9])
`SolverOptions.UMFPACK` – direct solver from the UMFPACK library.
Not all packages are supported on all implementations. An exception may be thrown on some platforms if a particular package is requested. Currently `esys.finley` supports `SolverOptions.PASO` (as default) and,

if available, `SolverOptions.MKL`[4] and `SolverOptions.UMFPACK`.

getPackage()
returns the solver package key.

setTrilinosParameter([name , value])
sets Trilinos parameters for use by MueLu and Trilinos iterative and direct solvers. It is important that the exact space between words is included in the names. The following keywords are supported:
`"verbosity"`: controls level of AMG output, `"none"`, `"low"`, `"medium"`, `"high"`, and `"extreme"`
`"problem:type"`: `"unknown"`, `"Poisson-2D"`, `"Poisson-3D"`, `"Elasticity-2D"`, `"Elasticity-3D"`, `"Poisson-2D-complex"`, `"Poisson-3D-complex"`, `"Elasticity-2D-complex"`, `"Elasticity-3D-complex"`, `"ConvectionDiffusion"` and `"MHD"`.
`"number of equations"`: number of equations
`"multigrid algorithm"`: `"sa"`, `"unsmoothed"`, `"pg"`, `"interp"`, `"emin"`, `"semicoarsen"`
`"sa: damping factor"`: for smoothed aggregation AMG alorithm, default =1.3
`"sa: use filtered matrix"`: Booleon
`"filtered matrix: use lumping"`: Boolean
`"filtered matrix: reuse eigenvalue"`: Boolean
`"interp: interpolation order"`: 0, 1
`"interp: build coarse coordinates`: Boolean
`"emin: iterative method"`: `"cg"`, `"gmres"` or `"sd"`
`"emin: num iterations"`: integer
`"emin: num reuse iterations"`: integer
`"cycle type"`: `"V"` or `"W"`
`"xml parameter file"`: name of XML file containing chosen XML parameters

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"`: number of iteration steps
`"cum_num_iter"`: cumulative number of iteration steps
`"num_level"`: number of levels in the multi level solver
`"num_inner_iter"`: number of inner iteration steps
`"cum_num_inner_iter"`: cumulative number of inner iteration steps
`"time"`: execution time
`"cum_time"`: cumulative execution time
`"set_up_time"`: time to set up the solver, typically this includes factorization and reordering
`"cum_set_up_time"`: cumulative time to set up the solver
`"net_time"`: net execution time, excluding setup time for the solver and execution time for preconditioner
`"cum_net_time"`: cumulative net execution time
`"residual_norm"`: norm of the final residual
`"converged"`: status of convergence
`"preconditioner_size"`: size of preconditioner in MBytes
`"time_step_backtracking_used"`: whether the time step size was reduced after convergence failure
`"coarse_level_sparsity"`: the sparsity at coarse level (AMG only)
`"num_coarse_unknowns"`: number of unknowns at coarse level (AMG only) .

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.

[4]If the stiffness matrix is non-regular `MKL` may return without returning a proper error code. If you observe suspicious solutions when using MKL, this may be caused by a non-invertible operator.

setReordering(ordering)
: 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.DEFAULT_REORDERING` – as recommended by the solver
`SolverOptions.MINIMUM_FILL_IN` – reorder matrix to reduce fill-in during factorization
`SolverOptions.NESTED_DISSECTION` – reorder matrix to improve load balancing during factorization
`SolverOptions.NO_REORDERING` – no matrix reordering applied.

getReordering()
: returns the key of the reordering method to be applied if supported by the solver.

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

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

setTruncation([truncation=20])
: sets the number of residuals in `SolverOptions.GMRES` to be stored for orthogonalization. The more residuals are stored the faster `SolverOptions.GMRES` converges but the higher the storage needs are and the more expensive a single iteration step becomes.

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.

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

getTolerance()
: returns the relative tolerance for the solver.

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

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.

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

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

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

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

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

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

setVerbosityOn()
switches the verbosity of the solver on.

setVerbosityOff()
switches the verbosity of the solver off.

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

setInnerToleranceAdaptionOn()
switches the automatic selection of inner tolerance on.

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

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

getInnerIterMax()
returns the maximum number of inner iteration steps.

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

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

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

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

MKL
the `MKL` library by Intel, Reference [13][5].

[5]The `MKL` library will only be available when the Intel compilation environment was used to build `esys.escript`.

UMFPACK
the `UMFPACK` library, Reference [4]. Note that `UMFPACK` is not parallelized.

PASO
`PASO` is the default solver library of `esys.finley`, see Section 5.

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

DIRECT
the default direct linear solver.

CHOLEVSKY
direct solver based on Cholevsky factorization (or similar), see Reference [16]. The solver requires a symmetric PDE.

PCG
preconditioned conjugate gradient method, see Reference [26]. The solver requires a symmetric PDE.

TFQMR
transpose-free quasi-minimal residual method, see Reference [26].

GMRES
the GMRES method, see Reference [26]. Truncation and restart are controlled by the `truncation` and `restart` parameters of `getSolution`.

MINRES
minimal residual method

ROWSUM_LUMPING
row sum lumping of the stiffness matrix, see Section Reference [4.4] for details. Lumping does not use the linear system solver library.

HRZ_LUMPING
HRZ lumping of the stiffness matrix, see Section Reference [4.4] for details. Lumping does not use the linear system solver library.

PRES20
the GMRES method with truncation after five residuals and restart after 20 steps, see Reference [26].

CGS
conjugate gradient squared method, see Reference [26].

BICGSTAB
stabilized bi-conjugate gradients methods, see Reference [26].

SSOR
symmetric successive over-relaxation method, see Reference [26]. 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 [16].

JACOBI
the Jacobi preconditioner, see Reference [16].

AMG
the algebraic multi grid method, see Reference [17]. This method can be used as linear solver method but is more robust when used as a preconditioner.

GAUSS_SEIDEL
the symmetric Gauss-Seidel preconditioner, see Reference [16]. `getNumSweeps()` is the number of sweeps used.

REC_ILU
recursive incomplete LU factorization preconditioner, see Reference [19]. This method is similar to the one used for `SolverOptions.ILU0` but applies reordering during the factorization.

NO_REORDERING
no reordering is used during factorization.

DEFAULT_REORDERING
the default reordering method during factorization.

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

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

TRILINOS
the Trilinos library [9] is used as a solver.

NO_PRECONDITIONER
no preconditioner is applied.

DIRECT_INTERPOLATION
direct interpolation in `SolverOptions.AMG` , see [17]

CLASSIC_INTERPOLATION
classic interpolation in `SolverOptions.AMG` , see [17]

CLASSIC_INTERPOLATION_WITH_FF_COUPLING
classic interpolation with enforced FF coupling in `SolverOptions.AMG` , see [17]

4.4 Some Remarks on Lumping

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

4.4.1 Scalar wave equation

One example where lumping can be applied to a hyperbolic problem, is the scalar wave equation

$$u_{,tt} = c^2 u_{,ii} \; . \tag{4.27}$$

In this example, both of the lumping schemes are tested against the reference solution

$$u = sin(5\pi(x_0 - c * t)) \tag{4.28}$$

over the 2D unit square. Note that $u_{,i}n_i = 0$ on faces $x_1 = 0$ and $x_1 = 1$. Thus, on the faces $x_0 = 0$ and $x_0 = 1$ the solution is constrained.

To solve this problem the explicit Verlet scheme was used with a constant time step size dt given by

$$u^{(n)} = 2u^{(n-1)} - u^{(n-2)} + dt^2 a^{(n)} \tag{4.29}$$

for all $n = 2, 3, \ldots$ where the upper index (n) refers to values at time $t^{(n)} = t^{(n-1)} + h$ and $a^{(n)}$ is the solution of

$$a^{(n)} = c^2 u^{(n-1)}_{,ii} \; . \tag{4.30}$$

This equation can be interpreted as a PDE for the unknown value $a^{(n)}$, which must be solved at each time-step. In the notation of equation 4.1 we thus set $D = 1$ and $X = -c^2 u^{(n-1)}_{,i}$. Furthermore, in order to maintain stability, the time step size needs to fulfill the Courant-Friedrichs-Lewy condition (CFL condition). For this example, the CFL condition takes the form

$$dt = f \cdot \frac{dx}{c}. \tag{4.31}$$

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

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

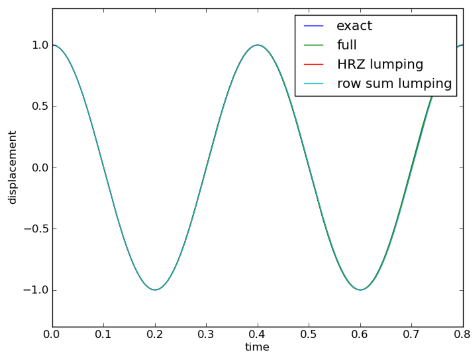


FIGURE 4.1: Amplitude at point $(\frac{1}{2}, \frac{1}{2})$ using the acceleration formulation 4.30 of the Velet scheme with order 1 elements, element size $dx = 0.01$, and $c = 1$.

Alternatively, one can directly solve for $u^{(n)}$ by inserting equation 4.29 into equation 4.30:

$$u^{(n)} = 2u^{(n-1)} - u^{(n-2)} + (dt \cdot c)^2 u^{(n-1)}_{,ii} \; . \tag{4.32}$$

This can also be interpreted as a PDE that must be solved at each time-step, but for the unknown $u^{(n)}$. As per equation 4.1 we set the general form coefficients to: $D = 1$; $Y = 2u^{(n-1)} - u^{(n-2)}$; and $X = -(h \cdot c)^2 u^{(n-1)}_{,i}$. For the full mass matrix, the acceleration 4.30 and displacement formulations 4.32 are identical.

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

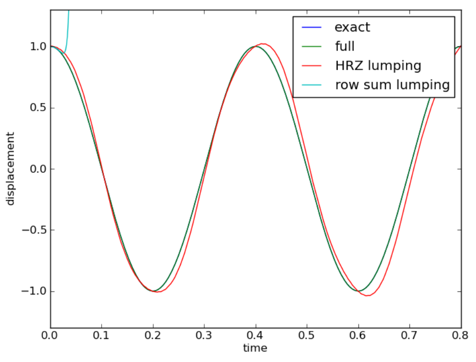


FIGURE 4.2: Amplitude at point $(\frac{1}{2}, \frac{1}{2})$ using the acceleration formulation 4.30 of the Velet scheme with order 2 elements, element size 0.01, and $c = 1$.

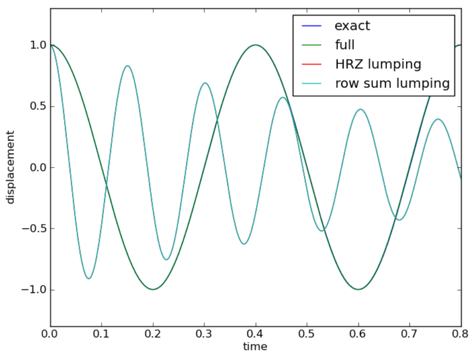


FIGURE 4.3: Amplitude at point $(\frac{1}{2}, \frac{1}{2})$ using the displacement formulation 4.32 of the Velet scheme with order 1 elements, element size 0.01 and $c = 1$.

4.4.2 Advection equation

Consider now, a second example that demonstrates the advection equation

$$u_{,t} = (v_i u)_i \ . \tag{4.33}$$

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

$$u(x,t) = \left\{ \begin{array}{cc} 1 & x_0 < t \\ 0 & x_0 \geq t \end{array} \right\}. \tag{4.34}$$

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

$$du^{(n-\frac{1}{2})} = \frac{dt}{2}(v_i u^{(n-1)})_i \tag{4.35}$$

$$du^{(n)} = dt(v_i(u^{(n-1)} + du^{(n-\frac{1}{2})}))_i \tag{4.36}$$

$$u^{(n)} = u^{(n)} + du^{(n)} \tag{4.37}$$

This can be reformulated to calculate $u^{(n)}$ directly:

$$u^{(n-\frac{1}{2})} = u^{(n-1)} + \frac{dt}{2}(v_i u^{(n-1)})_i \tag{4.38}$$

$$u^{(n)} = u^{(n-1)} + dt(v_i u^{(n-\frac{1}{2})})_i \tag{4.39}$$

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

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

$$dt = f \cdot \frac{dx}{\|v\|}. \tag{4.40}$$

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

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

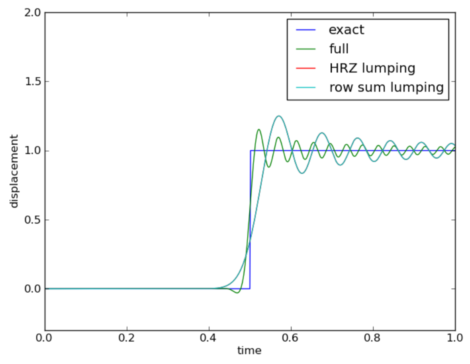


FIGURE 4.4: Amplitude at point $(\frac{1}{2}, \frac{1}{2})$ using the incremental formulation 4.35 of the Taylor-Galerkin scheme with order 1 elements, element size $dx = 0.01$, $v = (1, 0)$.

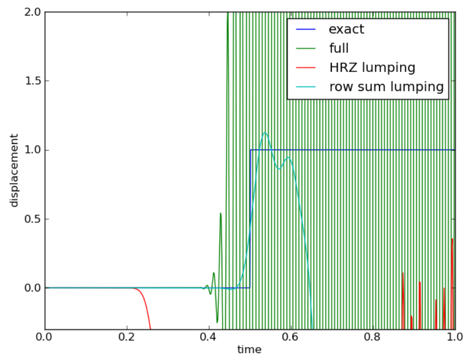


FIGURE 4.5: Amplitude at point $(\frac{1}{2}, \frac{1}{2})$ using the incremental formulation 4.35 of the Taylor-Galerkin scheme with order 2 elements, element size 0.01, $v = (1, 0)$.

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

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

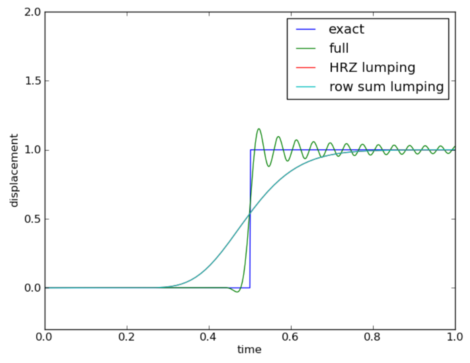


FIGURE 4.6: Amplitude at point $(\frac{1}{2}, \frac{1}{2})$ using the direct formulation 4.38 of the Taylor-Galerkin scheme using order 1 elements, element size $dx = 0.01$, $v = (1, 0)$.

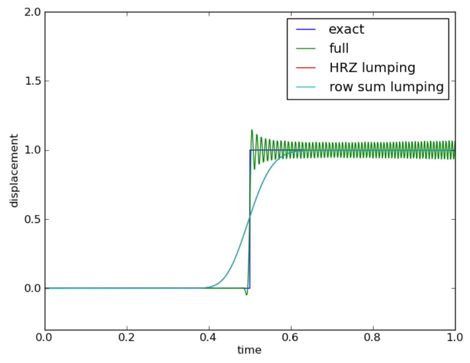


FIGURE 4.7: Amplitude at point $(\frac{1}{2}, \frac{1}{2})$ using the direct formulation 4.38 of the Taylor-Galerkin scheme using order 1 elements, element size $dx = 0.002$, $v = (1, 0)$.

4.4.3 Summary

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

In transport type problems, it is essential that row sum lumping is used to achieve a smooth solution. Additionally, it is not recommended that second order elements be used in advection type problems.

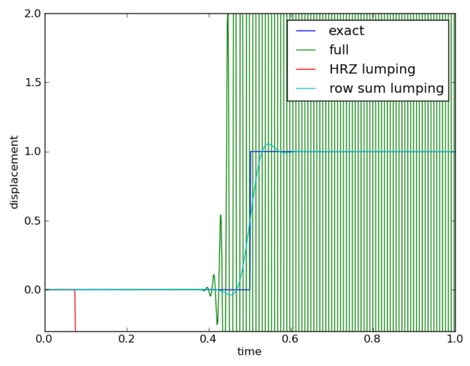


FIGURE 4.8: Amplitude at point $(\frac{1}{2}, \frac{1}{2})$ using the direct formulation 4.38 of the Taylor-Galerkin scheme using order 2 elements, element size 0.01, $v = (1, 0)$.

CHAPTER
FIVE

The `esys.finley` Module

The `esys.finley` library allows the creation of domains for solving linear, steady partial differential equations (PDEs) or systems of PDEs using isoparametrical finite elements. It supports unstructured 1D, 2D and 3D meshes. The PDEs themselves are represented by the `LinearPDE` class of `esys.escript`. `esys.finley` is parallelized under both *OpenMP* and *MPI*.

5.1 Formulation

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

$$
\begin{aligned}
& \int_{\Omega} A_{jl} \cdot v_{,j} u_{,l} + B_{j} \cdot v_{,j} u + C_{l} \cdot v u_{,l} + D \cdot v u \; d\Omega \\
+ & \int_{\Gamma} d \cdot v u \; d\Gamma + \int_{\Gamma^{contact}} d^{contact} \cdot [v][u] \; d\Gamma \\
= & \int_{\Omega} X_{j} \cdot v_{,j} + Y \cdot v \; d\Omega \\
+ & \int_{\Gamma} y \cdot v \; d\Gamma + \int_{\Gamma^{contact}} y^{contact} \cdot [v] \; d\Gamma
\end{aligned} \tag{5.1}
$$

5.2 Meshes

To understand the usage of `esys.finley` one needs to have an understanding of how the finite element meshes are defined. Figure 5.1 shows an example of the subdivision of an ellipse into so-called elements. In this case, triangles have been used but other forms of subdivisions can be constructed, e.g. quadrilaterals or, in the three-dimensional case, into tetrahedra and hexahedra. The idea of the finite element method is to approximate the solution by a function which is a polynomial of a certain order and is continuous across its boundary to neighbour elements. In the example of Figure 5.1 a linear polynomial is used on each triangle. As one can see, the triangulation is quite a poor approximation of the ellipse. It can be improved by introducing a midpoint on each element edge then positioning those nodes located on an edge expected to describe the boundary, onto the boundary. In this case the triangle gets a curved edge which requires a parameterization of the triangle using a quadratic polynomial. For this case, the solution is also approximated by a piecewise quadratic polynomial (which explains the name isoparametrical elements), see Reference [29, 2] for more details. `esys.finley` also supports macro elements. For these elements a piecewise linear approximation is used on an element which is further subdivided (in the case of `esys.finley` halved). As such, these elements do not provide more than a further mesh refinement but should be used in the case of incompressible flows, see `StokesProblemCartesian`. For these problems a linear approximation of the pressure across the element is used (use the reduced solution `FunctionSpace`) while the refined element is used to approximate velocity. So a macro element provides a continuous pressure approximation together with a velocity approximation on a refined mesh. This approach is necessary to make sure that the incompressible flow has a unique solution.

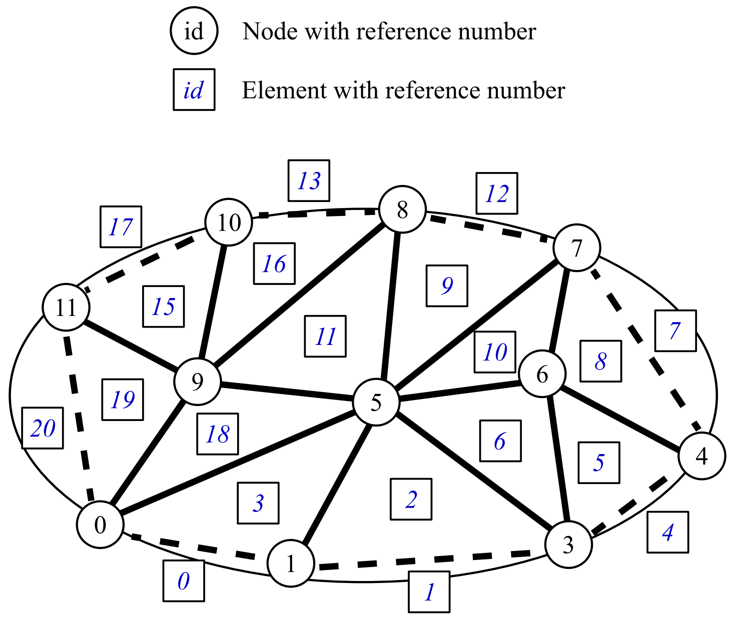


FIGURE 5.1: Subdivision of an Ellipse into triangles order 1 (*Tri3*)

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 5.1 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 5.1 line elements with two nodes are used. The elements are also defined by their describing nodes, e.g. the face element with reference number 20, which has type *Line2*, is defined by the nodes with the reference numbers 11 and 0. Again the order is crucial, if moving from the first to second node the domain has to lie on the left hand side (in the case of a two-dimensional surface element the domain has to lie on the left hand side when moving counterclockwise). If the gradient on the surface of the domain is to be calculated rich face elements need to be used. Rich elements on a face are identical to interior elements but with a modified order of nodes such that the 'first' face of the element aligns with the surface of the domain. In Figure 5.1 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 5.2 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 and 5 define contact element 4, where the coordinates of nodes 12 and 5 and nodes 4 and 6 are identical, with the idea that nodes 12 and 9 are located above and nodes 5 and 6 below the contact region. Again, the order of the nodes within an element is crucial. There is also the option of using rich elements if the gradient is to be calculated on the contact region. Similarly to the rich face elements these are constructed from two interior elements by reordering the nodes such that the 'first' face of the element above and the 'first' face of the element below the contact regions line up. The rich version of element 4 is of type *Rec4Face_Contact* and is defined by the nodes 9, 12, 16, 18, 6, 5, 0 and 2. Table 5.1 shows the interior element types and the corresponding element types to be used on the face and contacts. Figure 5.3, Figure 5.4 and Figure 5.5 show the ordering of the nodes within an element.

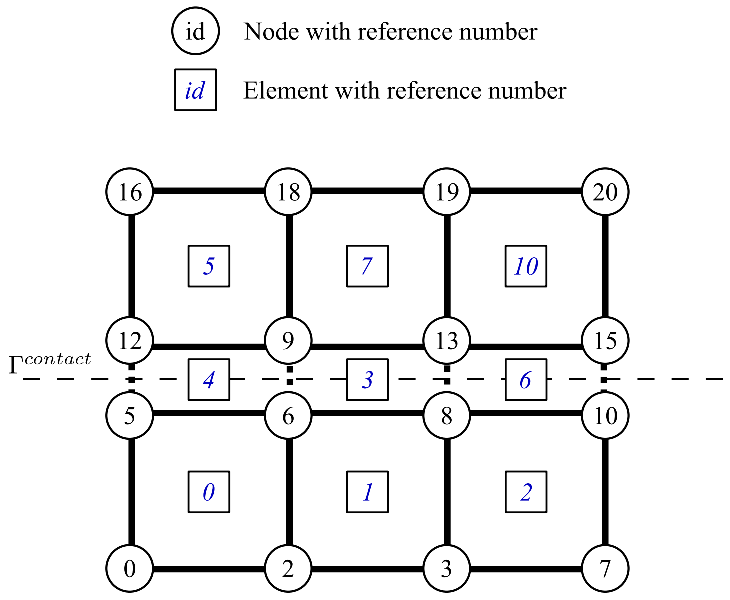


FIGURE 5.2: Mesh around a contact region (*Rec4*)

The native `esys.finley` file format is defined as follows. Each node `i` has `dim` spatial coordinates `Node[i]`, a reference number `Node_ref[i]`, a degree of freedom `Node_DOF[i]` and a tag `Node_tag[i]`. In most cases `Node_DOF[i]` =`Node_ref[i]` however, for periodic boundary conditions, `Node_DOF[i]` is chosen differently, see example below. The tag can be used to mark nodes sharing the same properties. Element `i` is defined by the `Element_numNodes` nodes `Element_Nodes[i]` which is a list of node reference numbers. The order of these is crucial. Each element has a reference number `Element_ref[i]` and a tag `Element_tag[i]`. The tag can be used to mark elements sharing the same properties. For instance elements above a contact region are marked with tag 2 and elements below a contact region are marked with tag 1. `Element_Type` and `Element_Num` give the element type and the number of elements in the mesh. Analogue notations are used for face and contact elements.

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*
Hex27	*Rec9*	N/A	N/A	N/A
Hex27Macro	*Rec9Macro*	N/A	N/A	N/A
Tet10Macro	*Tri6Macro*	N/A	N/A	N/A
Rec9Macro	*Line3Macro*	N/A	N/A	N/A
Tri6Macro	*Line3Macro*	N/A	N/A	N/A

Table 5.1: Finley elements and corresponding elements to be used on domain faces and contacts. The rich types have to be used if the gradient of the function is to be calculated on faces and contacts, respectively.

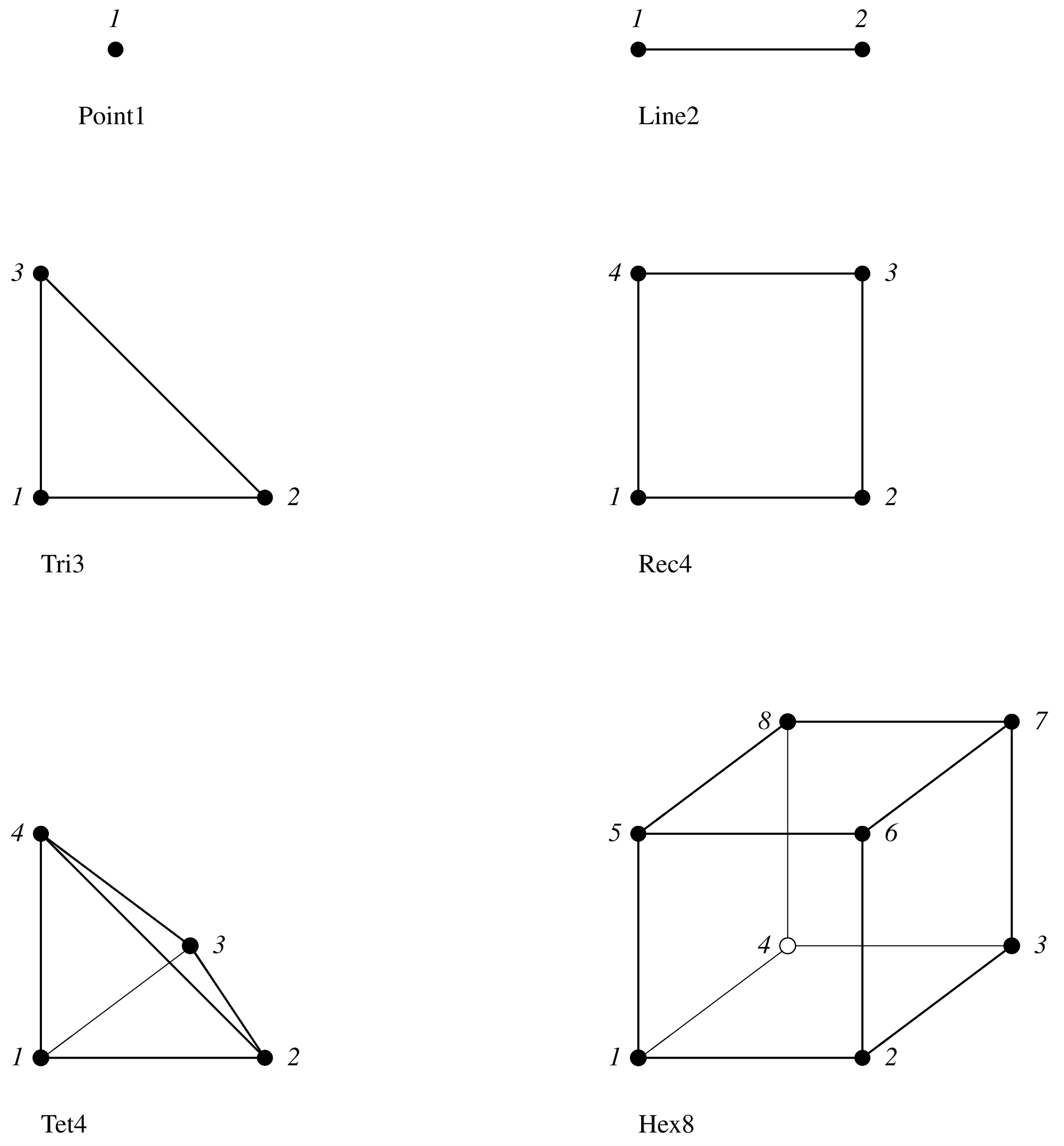


FIGURE 5.3: Elements of order 1

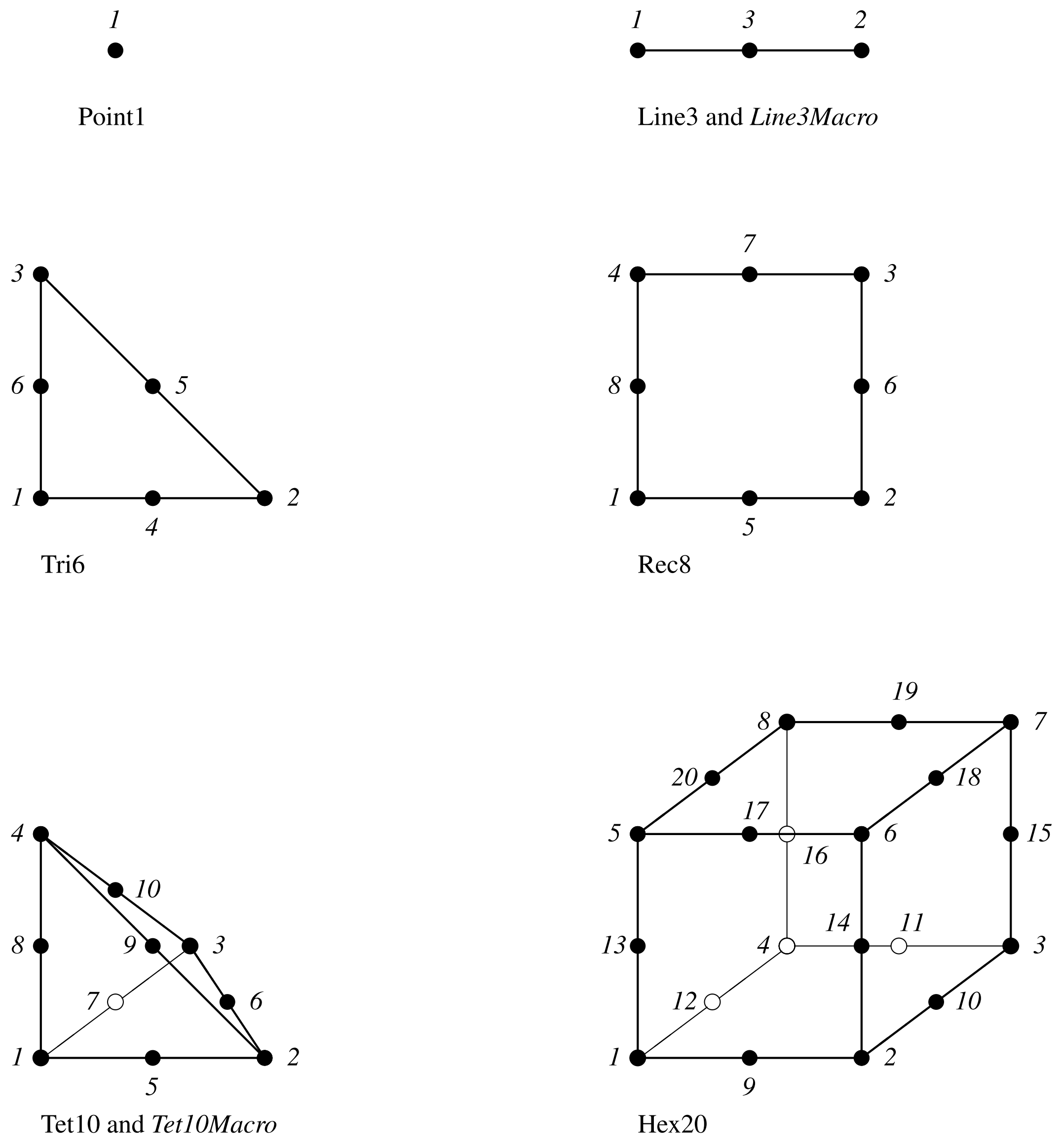


FIGURE 5.4: Elements of order 2 and macro elements

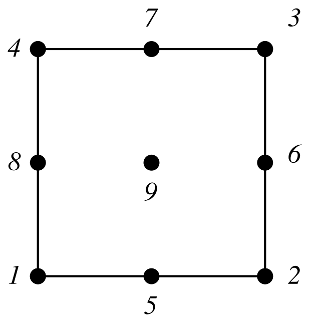
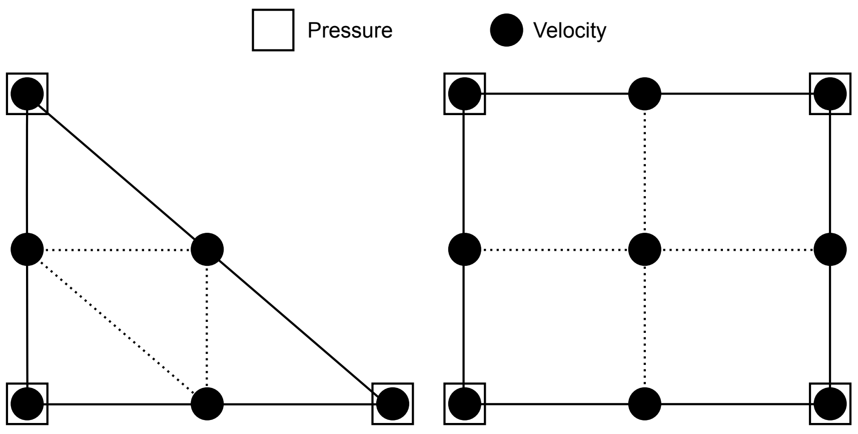


FIGURE 5.5: *Rec9* and *Rec9Macro*

setSolverMethod	DIRECT	PCG	GMRES	TFQMR	MINRES	PRES20	BICGSTAB
setReordering	✓						
setRestart			✓			20	
setTruncation			✓			5	
setIterMax		✓	✓	✓	✓	✓ ✓	
setTolerance		✓	✓	✓	✓	✓ ✓	
setAbsoluteTolerance		✓	✓	✓	✓	✓	
setReordering	✓						

Table 5.2: Solvers available for `esys.finley` and the `PASO` package and the relevant options in `SolverOptions`. `MKL` supports `MINIMUM_FILL_IN` and `NESTED_DISSECTION` reordering. Currently the `UMFPACK` interface does not support any reordering.

5.3 Macro Elements



(a) Triangle (b) Quadrilateral

FIGURE 5.6: Macro elements in `esys.finley`

`esys.finley` supports the usage of macro elements which can be used to achieve LBB compliance when solving incompressible fluid flow problems. LBB compliance is required to get a problem which has a unique solution for pressure and velocity. For macro elements the pressure and velocity are approximated by a polynomial of order 1 but the velocity approximation bases on a refinement of the elements. The nodes of a triangle and quadrilateral element are shown in Figures 5.6(a) and 5.6(b), respectively. In essence, the velocity uses the same nodes like a quadratic polynomial approximation but replaces the quadratic polynomial by piecewise linear polynomials. In fact, this is the way `esys.finley` defines the macro elements. In particular `esys.finley` uses the same local ordering of the nodes for the macro element as for the corresponding quadratic element. Another interpretation is that one uses a linear approximation of the velocity together with a linear approximation of the pressure but on elements created by combining elements to macro elements. Notice that the macro elements still use quadratic interpolation to represent the element and domain boundary. However, if elements have linear boundaries a macro element approximation for the velocity is equivalent to using a linear approximation on a mesh which is created through a one-step global refinement. Typically macro elements are only required to use when an incompressible fluid flow problem is solved, e.g. the Stokes problem in Section **??**. Please see Section 5.2 for more details on the supported macro elements.

5.4 Linear Solvers in `SolverOptions`

If available, Trilinos sovers are used by default, see Chapter 9. Table 5.2 and Table 5.3 show the solvers supported by `esys.finley` through the `PASO` library. `esys.finley` uses the iterative solvers `SolverOptions.PCG` for symmetric and `SolverOptions.BICGSTAB` for non-symmetric problems.

NO_PRECONDITIONER	AMG	JACOBI	GAUSS_SEIDEL	REC_ILU	RILU	ILU0
status:	✓	✓	✓	✓		✓
setLevelMax	✓					
setCoarseningThreshold	✓					
setMinCoarseMatrixSize	✓					
setMinCoarseMatrixSparsity	✓					
setNumSweeps		✓	✓			
setNumPreSweeps	✓					
setNumPostSweeps	✓					
setDiagonalDominanceThreshold	✓					
setAMGInterpolation	✓					
setRelaxationFactor					✓	

Table 5.3: Preconditioners available for `esys.finley` and the `PASO` package and the relevant options in `SolverOptions`.

5.5 Functions

ReadMesh(fileName [, [integrationOrder=-1], optimize=True])

creates a `Domain` object from the FEM mesh defined in file `fileName`. The file must be in the `esys.finley` file format. If `integrationOrder` is positive, a numerical integration scheme is chosen which is accurate on each element up to a polynomial of degree `integrationOrder` . Otherwise an appropriate integration order is chosen independently. By default the labeling of mesh nodes and element distribution is optimized. Set `optimize=False` to switch off relabeling and redistribution.

ReadGmsh(fileName, numDim, [, [integrationOrder=-1], optimize=True[, useMacroElements=False]])

creates a `Domain` object from the FEM mesh defined in file `fileName` for a domain of dimension `numDim`. The file must be in the *Gmsh*[6] file format. If `integrationOrder` is positive, a numerical integration scheme is chosen which is accurate on each element up to a polynomial of degree `integrationOrder` . Otherwise an appropriate integration order is chosen independently. By default the labeling of mesh nodes and element distribution is optimized. Set `optimize=False` to switch off relabeling and redistribution. If `useMacroElements` is set, second order elements are interpreted as macro elements.

load(fileName)

recovers a `Domain` object from a dump file `fileName` created by the `dump` method of a `Domain` object.

Rectangle(n0,n1,order=1,l0=1.,l1=1., integrationOrder=-1, periodic0=*False*, periodic1=*False*, useElementsOnFace=*False*, 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 x_0-axis and `n1` elements along the x_1-axis. For `order` =1 and `order` =2, elements of type *Rec4* and *Rec8* are used, respectively. In the case of `useElementsOnFace` =*False*, *Line2* and *Line3* are used to subdivide the edges of the rectangle, respectively. If `order` =-1, *Rec8Macro* and *Line3Macro* are used. This option should be used when solving incompressible fluid flow problems, e.g. `StokesProblemCartesian`. In the case of `useElementsOnFace` =*True* (this option should be used if gradients are calculated on domain faces), *Rec4Face* and *Rec8Face* are used on the edges, respectively. If `integrationOrder` is positive, a numerical integration scheme is chosen which is accurate on each element up to a polynomial of degree `integrationOrder` . Otherwise an appropriate integration order is chosen independently. If `periodic0` =*True*, periodic boundary conditions along the x_0-direction are enforced. That means for any solution of a PDE solved by `esys.finley` the values on the line $x_0 = 0$ will be identical to the values on $x_0 =$ `l0`. Correspondingly, `periodic1` =*True* sets periodic boundary conditions in the x_1-direction. If `optimize` =*True* mesh node relabeling will be attempted to reduce the computation and also ParMETIS will be used to improve the mesh partition if running on multiple CPUs with *MPI*.

Brick(n0,n1,n2,order=1,l0=1.,l1=1.,l2=1., integrationOrder=-1, periodic0=*False*, periodic1=*False*, periodic2=*False*, useElementsOnFace=*False*,useFullElementOrder=*False*, optimize=*False*)

generates a `Domain` object representing a three-dimensional brick between $(0,0,0)$ and $(l0,l1,l2)$ with orthogonal faces. The brick is filled with `n0` elements along the x_0-axis, `n1` elements along the x_1-axis and `n2` elements along the x_2-axis. For `order` =1 and `order` =2, elements of type *Hex8* and *Hex20* are used, respectively. In the case of `useElementsOnFace` =*False*, *Rec4* and *Rec8* are used to subdivide the faces of the brick, respectively. In the case of `useElementsOnFace` =*True* (this option should be used if gradients are calculated on domain faces), *Hex8Face* and *Hex20Face* are used on the brick faces, respectively. If `order` =-1, *Hex20Macro* and *Rec8Macro* are used. This option should be used when solving incompressible fluid flow problems, e.g. `StokesProblemCartesian`. If `integrationOrder` is positive, a numerical integration scheme is chosen which is accurate on each element up to a polynomial of degree `integrationOrder` . Otherwise an appropriate integration order is chosen independently. If `periodic0` =*True*, periodic boundary conditions along the x_0-direction are enforced. That means for any solution of a PDE solved by `esys.finley` the values on the plane $x_0 = 0$ will be identical to the values on $x_0 = \mathtt{l0}$. Correspondingly, `periodic1` =*True* and `periodic2` =*True* sets periodic boundary conditions in the x_1-direction and x_2-direction, respectively. If `optimize` =*True* mesh node relabeling will be attempted to reduce the computation and also ParMETIS will be used to improve the mesh partition if running on multiple CPUs with *MPI*.

GlueFaces(meshList, tolerance=1.e-13)
generates a new `Domain` object from the list `meshList` of `esys.finley` meshes. Nodes in face elements whose difference of coordinates is less than `tolerance` times the diameter of the domain are merged. The corresponding face elements are removed from the mesh. `GlueFaces` is not supported under *MPI* with more than one rank.

JoinFaces(meshList, tolerance=1.e-13)
generates a new `Domain` object from the list `meshList` of `esys.finley` meshes. Face elements whose node coordinates differ by less than `tolerance` times the diameter of the domain are combined to form a contact element. The corresponding face elements are removed from the mesh. `JoinFaces` is not supported under *MPI* with more than one rank.

CHAPTER
SIX

The `esys.ripley` Module

`esys.ripley` is an alternative domain library to `esys.finley`; it supports structured, uniform meshes with rectangular elements in 2D and hexahedral elements in 3D. Uniform meshes allow a straightforward division of elements among processes with *MPI* and allow for a number of optimizations when solving PDEs. `esys.ripley` also supports fast assemblers for certain types of PDE (specifically Lamé and Wave PDEs). These assemblers make use of the regular nature of the domain to optimize the stiffness matrix assembly process for these specific problems. Finally, `esys.ripley` is currently the only domain family that supports GPU-based solvers.

As a result, `esys.ripley` domains cannot be created by reading from a mesh file since only one element type is supported and all elements need to be equally sized. For the same reasons, `esys.ripley` does not allow assigning coordinates via `setX`.

While `esys.ripley` cannot be used with mesh files, it can be used to read in *GOCAD* data. A script with an example of a voxet reader is included in the examples as `voxet_reader.py`.

Other than use of meshfiles, `esys.ripley` and `esys.finley` are generally interchangeable in a script with both modules having the `Rectangle` or `Brick` functions available. Consider the following example which creates a 2D `esys.ripley` domain:

```
from esys.ripley import Rectangle, Brick
dom = Rectangle(9, 9)
```

Multi-resolution domains are supported in `esys.ripley` via `MultiBrick` and `MultiRectangle`. Each level of one of these domains has twice the elements in each axis of the next lower resolution. The `MultiBrick` is not currently supported when running `esys.escript` with multiple processes using *MPI*. Interpolation between these multi-resolution domains is possible providing they have matching dimensions and subdivisions, along with a compatible number of elements.

To simplify these conditions the use of `MultiResolutionDomain` is highly recommended. The following example creates two 2D domains of different resolutions and interpolates between them:

```
from esys.ripley import MultiResolutionDomain
mrd = MultiResolutionDomain(2, n0=10, n1=10)
ten_by_ten = mrd.getLevel(0)
data10 = Vector(..., Function(ten_by_ten))
...
forty_by_forty = mrd.getLevel(2)
data40 = interpolate(data10, Function(forty_by_forty))
```

6.1 Formulation

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

$$\begin{aligned}
& \int_{\Omega} A_{jl} \cdot v_{,j} u_{,l} + B_j \cdot v_{,j} u + C_l \cdot v u_{,l} + D \cdot vu \; d\Omega + \int_{\Gamma} d \cdot vu \; d\Gamma \\
= & \int_{\Omega} X_j \cdot v_{,j} + Y \cdot v \; d\Omega + \int_{\Gamma} y \cdot v \; d\Gamma
\end{aligned} \tag{6.1}$$

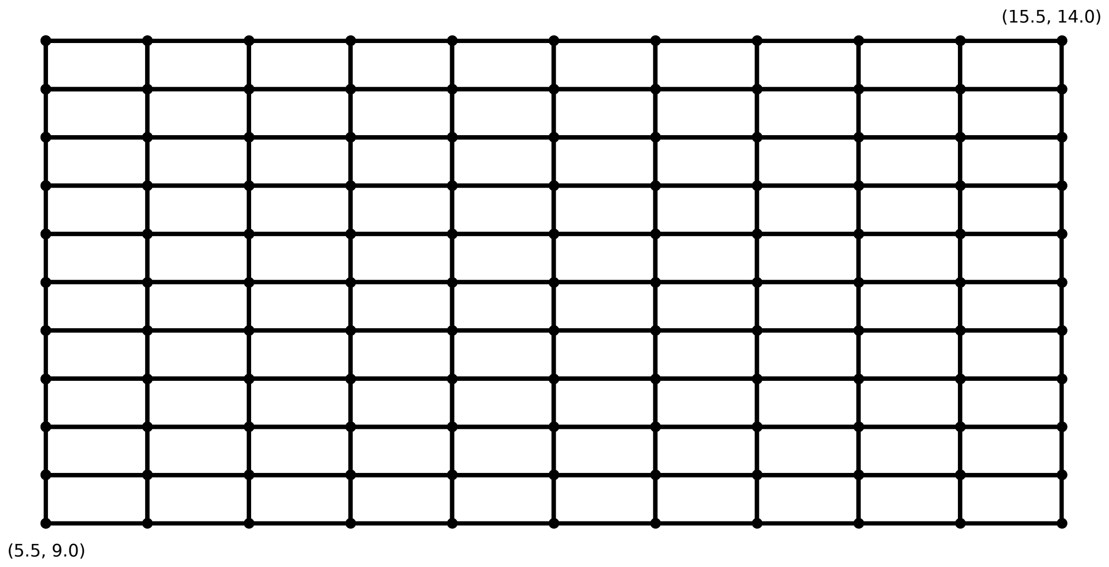


FIGURE 6.1: 10x10 `esys.ripley` Rectangle created with `l0=(5.5, 15.5)` and `l1=(9.0, 14.0)`

6.2 Meshes

An example 2D mesh from `esys.ripley` is shown in Figure 6.1. Mesh files cannot be used to generate `esys.ripley` domains, i.e. `esys.ripley` does not have `ReadGmsh` or `ReadMesh` functions. Instead, `esys.ripley` domains are always created using a call to `Brick` or `Rectangle`, see Section 6.3.

6.3 Functions

Brick(n0,n1,n2,l0=1.,l1=1.,l2=1.,d0=-1,d1=-1,d2=-1, diracPoints=list(), diracTags=list())

generates a `Domain` object representing a three-dimensional brick between $(0,0,0)$ and $(l0,l1,l2)$ with orthogonal faces. All elements will be regular. The brick is filled with `n0` elements along the x_0-axis, `n1` elements along the x_1-axis and `n2` elements along the x_2-axis. If built with *MPI* support, the domain will be subdivided `d0` times along the x_0-axis, `d1` times along the x_1-axis, and `d2` times along the x_2-axis. `d0`, `d1`, and `d2` must be factors of the number of *MPI* processes requested. If axial subdivisions are not specified, automatic domain subdivision will take place. This may not be the most efficient construction and will likely result in extra elements being added to ensure proper distribution of work. Any extra elements added in this way will change the length of the domain proportionately. `diracPoints` is a list of coordinate-tuples of points within the mesh, each point tagged with the respective string within `diracTags`.

Rectangle(n0,n1,l0=1.,l1=1.,d0=-1,d1=-1, diracPoints=list(), diracTags=list())

generates a `Domain` object representing a two-dimensional rectangle between $(0,0)$ and $(l0,l1)$ with orthogonal faces. All elements will be regular. The rectangle is filled with `n0` elements along the x_0-axis and `n1` elements along the x_1-axis. If built with *MPI* support, the domain will be subdivided `d0` times along the x_0-axis and `d1` times along the x_1-axis. `d0` and `d1` must be factors of the number of *MPI* processes requested. If axial subdivisions are not specified, automatic domain subdivision will take place. This may not be the most efficient construction and will likely result in extra elements being added to ensure proper distribution of work. Any extra elements added in this way will change the length of the domain proportionately. `diracPoints` is a list of coordinate-tuples of points within the mesh, each point tagged with the respective string within `diracTags`.

The arguments `l0`, `l1` and `l2` for `Brick` and `Rectangle` may also be given as tuples `(x0,x1)` in which case the coordinates will range between `x0` and `x1`. For example:

```
from esys.ripley import Rectangle
```

```
dom = Rectangle(10, 10, l0=(5.5, 15.5), l1=(9.0, 14.0))
```

This will create a rectangle with 10 by 10 elements where the bottom-left node is located at $(5.5, 9.0)$ and the top-right node has coordinates $(15.5, 14.0)$, see Figure 6.1.

The `MultiResolutionDomain` class is available as a wrapper, taking the dimension of the domain followed by the same arguments as `Brick` (if a two-dimensional domain is requested, any extra arguments over those used by `Rectangle` are ignored). All of these standard arguments to `MultiResolutionDomain` must be supplied as keyword arguments (e.g. `d0` =...). The `MultiResolutionDomain` can then generate compatible domains for interpolation.

6.4 Linear Solvers in `SolverOptions`

Currently direct solvers and GPU-based solvers are not supported under *MPI* when running with more than one rank. By default, `esys.ripley` uses the iterative solvers `SolverOptions.PCG` for symmetric and `SolverOptions.BICGSTAB` for non-symmetric problems. A GPU will not be used unless explicitly requested via the `setSolverTarget` method of the solver options. These solvers are only available if `esys.ripley` was built with *CUDA* support. If the direct solver is selected, which can be useful when solving very ill-posed equations, `esys.ripley` uses the `MKL` [1] solver package. If `MKL` is not available `UMFPACK` is used. If `UMFPACK` is not available a suitable iterative solver from `PASO` is used, but if a direct solver was requested via `SolverOptions` an exception will be raised.

[1]If the stiffness matrix is non-regular `MKL` may return without a proper error code. If you observe suspicious solutions when using `MKL` , this may be caused by a non-invertible operator.

CHAPTER
SEVEN

The `esys.speckley` Module

speckley is a high-order form of *ripley*, supporting structured, uniform meshes in two and three dimensions. Uniform meshes allow a more regular division of elements among compute nodes. Possible orders range from 2 to 10, inclusive.

speckley domains cannot be created by reading from a mesh file.

The family of domain that will result from a `Rectangle` or `Brick` call depends on which module is imported in the specific script. The following line is an example of importing `esys.speckley` domains:

```
from esys.speckley import Rectangle, Brick
```

7.1 Formulation

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

$$\int_{\Omega} D \cdot vu \; d\Omega + \int_{\Gamma} d \cdot vu \; d\Gamma = \int_{\Omega} X_j \cdot v_{,j} + Y \cdot v \; d\Omega + \int_{\Gamma} y \cdot v \; d\Gamma \tag{7.1}$$

7.2 Meshes

`esys.speckley` meshes are formed of regular elements using Gauss-Labatto-Legendre quadrature points. The number of quadrature points in each axis is dependent on the order of the domain. Examples of small Rectangle domains of different orders are shown in Figure 7.1.

Meshfiles cannot be used to generate `esys.speckley` domains.

7.3 Linear Solvers in `SolverOptions`

While `esys.speckley` has the same defaults as `esys.ripley`, the `SolverOptions.HRZ_LUMPING` must be set. `PASO` is not used in `esys.speckley`.

7.4 Cross-domain Interpolation

Data on a `esys.speckley` domain can be interpolated to a matching `esys.ripley` domain provided the two domains have identical dimension, length, and, in multi-process situations, domain sub-divisions.

A utility class, `SpeckleyToRipley` is available to simplify meeting these conditions. To gain access to the class, the following will be required in the script:

```
from esys.escript.domainCouplers import SpeckleyToRipley
```

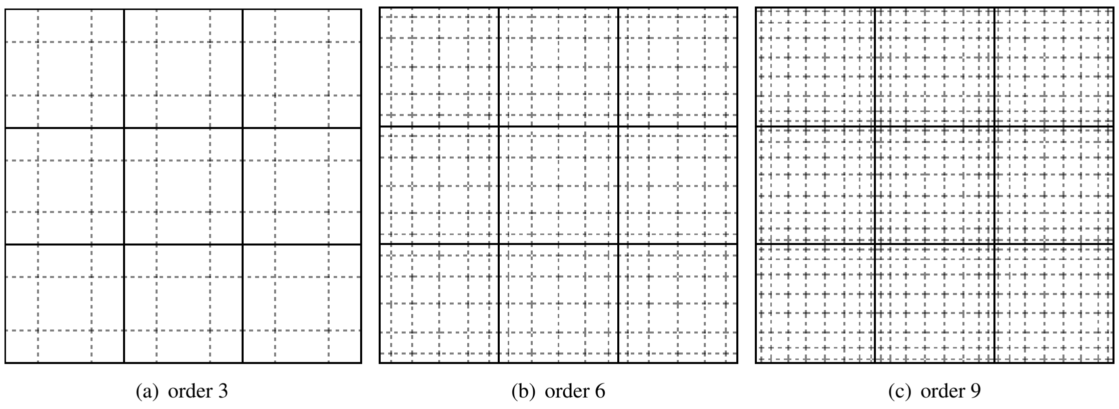

(a) order 3 (b) order 6 (c) order 9

FIGURE 7.1: 3x3 *speckley* Rectangle domains of different orders

7.5 Functions

Brick(order,n0,n1,n2,l0=1.,l1=1.,l2=1.,d0=-1,d1=-1,d2=-1, diracPoints=list(), diracTags=list())

generates a `Domain` object representing a three-dimensional brick between $(0,0,0)$ and $(l0,l1,l2)$ with orthogonal faces. All elements will be regular and of order `order`. The brick is filled with `n0` elements along the x_0-axis, `n1` elements along the x_1-axis and `n2` elements along the x_2-axis. If built with *MPI* support, the domain will be subdivided `d0` times along the x_0-axis, `d1` times along the x_1-axis, and `d2` times along the x_2-axis. `d0`, `d1`, and `d2` must be factors of the number of *MPI* processes requested. If axial subdivisions are not specified, automatic domain subdivision will take place. This may not be the most efficient construction and will likely result in extra elements being added to ensure proper distribution of work. Any extra elements added in this way will change the length of the domain proportionately. `diracPoints` is a list of coordinate-tuples of points within the mesh, each point tagged with the respective string within `diracTags`.

Rectangle(order,n0,n1,n2,l0=1.,l1=1.,l2=1.,d0=-1,d1=-1,d2=-1, diracPoints=list(), diracTags=list())

generates a `Domain` object representing a two-dimensional rectangle between $(0,0)$ and $(l0,l1)$ with orthogonal faces. All elements will be regular and of order `order`. The rectangle is filled with `n0` elements along the x_0-axis and `n1` elements along the x_1-axis. If built with *MPI* support, the domain will be subdivided `d0` times along the x_0-axis and `d1` times along the x_1-axis. `d0` and `d1` must be factors of the number of *MPI* processes requested. If axial subdivisions are not specified, automatic domain subdivision will take place. This may not be the most efficient construction and will likely result in extra elements being added to ensure proper distribution of work. Any extra elements added in this way will change the length of the domain proportionately. `diracPoints` is a list of coordinate-tuples of points within the mesh, each point tagged with the respective string within `diracTags`.

CHAPTER
EIGHT

The `esys.weipa` Module and Data Visualization

The *weipa* C++ library and accompanying *python* module allow exporting `esys.escript` `Data` objects and their domain in a format suitable for visualization. Besides creating output files, *weipa* can also interface with the *VisIt* visualization software. This allows accessing the latest simulation data while the simulation is still running without the need to save any files.

8.1 The `EscriptDataset` class

class EscriptDataset()

holds an *escript* dataset including a domain and data variables for a single time step and offers methods to export the data in various formats. It is preferable to create a dataset object using the `createDataset` function from `esys.weipa` (see Section 8.2) rather than using the (non-exposed) *python* constructor for the class.

The following methods are available:

setDomain(domain)

sets the `Domain` for this dataset. Note that the domain can only be set once and all `Data` objects added to this dataset must be defined on the same domain.

addData(data, name [, units=""])

adds the `Data` object `data` to this dataset which will be exported by the given `name`. Some export formats support data units which can be set through the `units` parameter, e.g. `"km/h"`. Before calling this method a domain must be set with `setDomain` and all `Data` objects added must be defined on the same domain. There is no restriction, however, on the `FunctionSpace` used.

setCycleAndTime(cycle, time)

sets the cycle and time values for this dataset. The cycle is an integer value which usually corresponds with the loop counter of the simulation script. That is, every time a new data file is created this counter is incremented. The value of `time` on the other hand is a floating point number that encodes some form of simulation time. Both, cycle and time may be read by analysis tools and shown alongside other metadata to the user.

setMeshLabels(x, y [, z=""])

sets the labels of the X, Y, and Z axis. By default, visualization tools display default strings such as "X-Axis" or "X" along the axes. Some export formats allow overriding these with more specific strings such as "Width", "Horizontal Distance", etc.

setMeshUnits(x, y [, z=""])
: sets the units to be displayed along the X, Y, and Z axis in visualization tools (if supported). Not all export formats will use these values.

setMetadataSchemaString([schema="" [, metadata=""]])
: adds custom metadata and/or XML schema strings to VTK files. The content of `schema` is added to the top-level *VTKFile* element so care must be taken to keep the resulting file valid. As an example, `schema` may contain the string `xmlns:gml="http://www.opengis.net/gml"`. The content of `metadata` will be written enclosed in `<MetaData>` tags. Thus, a valid example would be `<dataSource>something</dataSource>`. Note that these values are ignored by other exporters.

saveSilo(filename)
: saves the dataset in the *SILO* file format to a file named `filename`. The file extension `.silo` will be automatically added if not present.

saveVTK(filename)
: saves the dataset in the *VTK* file format to a file named `filename`. The file extension `.vtu` will be automatically added if not present. Certain combinations of function spaces cannot be written to a single *VTK* file due to format restrictions. In these cases this method will save separate files where each file contains compatible data. The function space name is appended to the filename to distinguish them.

8.2 Functions

createDataset(domain, **data)
: creates an `EscriptDataset` object, sets its domain, populates it with the given `Data` objects and returns it. Note that it is not possible to set units for the data variables added with this function. If this is required, it is recommended to call this function with a domain only and use the `addData` method subsequently.

saveVTK(filename [, domain=None [, metadata="" [, metadata_schema=None]]], **data)
: convenience function that creates a dataset with the given domain and `Data` objects and saves it to a file in the *VTK* file format. If `domain` is `None` the domain will be determined by the `Data` objects. See the `setDomain`, `addData`, `saveVTK`, and `setMetadataSchemaString` methods of the `EscriptDataset` class for details. Unlike the class method, the `metadata_schema` parameter should be a dictionary that maps namespace name to URI, e.g.
`{"gml":"http://www.opengis.net/gml"}`.

saveSilo(filename [, domain=None], **data)
: convenience function that creates a dataset with the given domain and `Data` objects and saves it to a file in the *SILO* file format. If `domain` is `None` the domain will be determined by the `Data` objects. See the `setDomain`, `addData`, and `saveSilo` methods of the `EscriptDataset` class for details.

saveVoxet(filename, **data)
: saves `Data` objects defined on a *ripley* grid in the *Voxet* file format suitable for import into *GOCAD* [8]. A *Voxet* dataset consists of a header file (extension `.vo`) and one property file (with no file extension) for each `Data` object.

visitInitialize(simFile [, comment=""])
: initializes the *VisIt* simulation interface which is responsible for the communication with a *VisIt* client. This function will create a file by the name given via `simFile` (extension `.sim2`) which can be loaded by a compatible *VisIt* client in order to connect to the simulation. The optional `comment` string is forwarded to the client. Note that this function only succeeds if *escript* was compiled with support for *VisIt* and the appropriate libraries are found in the runtime environment. Clients wanting to connect can only do so if the version number matches the version number used to compile `esys.weipa`. Calling this function does not make any data available yet, see the `visitPublishData` function.

visitPublishData(dataset)

publishes an `EscriptDataset` object through the *VisIt* simulation interface, checks for client requests and handles any outstanding ones. Before publishing any data, the `visitInitialize` function must be called to set up the interface. Since this function not only publishes new data but polls for incoming connections and handles requests, it should be called as often as practical (even with the same dataset) to avoid timeout errors from clients. On the other hand it should be noted that the same process(es) deal with visualization requests that run your simulation. So a request for an expensive task by a *VisIt* client will pause the simulation code while it is being processed.

8.3 Visualizing *escript* Data

This section gives a very brief overview on how data exported through `esys.weipa` can be visualized. While there are many visualization packages available that are compatible with *VTK* and *SILO* files produced by *escript*, this discussion will refer to *VisIt* [25], an actively maintained open source package optimally suited to visualize and analyze large datasets both interactively and through *python* scripts. You can find a number of manuals, a wiki page and links to mailing lists on the *VisIt* website. It is assumed that you have a working *VisIt* installation that can be started by entering `visit` on the command line.

The examples that follow will use the output produced by the Elastic Deformation example from section 1.5 (`heatedblock.py` in the example directory) which produces the file `deform.vtu`. This *VTK* file contains a 3D scalar variable called `stress` and a vector variable called `disp`, among others.

8.3.1 Using the *VisIt* GUI

Start the VisIt graphical user interface and open the file `deform.vtu` via the 'File' menu. Alternatively, you can directly open the file on startup by issuing

```
visit -o deform.vtu
```

You should see the *VisIt* GUI on the left hand side and an empty visualization window on the right. Click on 'Add' under Plots in the GUI to bring up a menu of plot types, then click on 'Pseudocolor' and select 'stress'. This will add a plot to the list which maps values of the 'stress' variable to colors. Note, that the plot will not be generated until you click on the 'Draw' button in the GUI. You should now see a coloured box in the visualization window which you can rotate around and inspect from different angles using your mouse. The example uses a coarse mesh of 10 by 10 by 10 elements which are clearly visible in this plot.

We can improve the visual effect by enabling interpolation between the elements. To do so, bring up the plot attributes by double-clicking the 'Pseudocolor - stress' plot entry in the GUI. Next, select 'Nodal' under 'Centering', click on 'Apply' and dismiss the dialog. Notice how the colours now smoothly blend into each other and the element boundaries are no longer visible.

Now we will add arrows to visualize the displacement vectors. Click on 'Add' and under 'Vector' select 'disp'. Once again click on 'Draw' to execute the new plot. By default only few vectors are shown but since the mesh is very coarse we can tell *VisIt* to draw all available vectors. Bring up the Vector plot attributes (double-click on the plot as before) and under 'Vector amount' select 'Stride', leaving the parameter as 1. Click on 'Apply' and dismiss the dialog.

As a final step we would like to see inside the plot. One possibility to do so is slicing. However, we want to keep all vectors while slicing only the Pseudocolor plot. In *VisIt* slicing is one of the Operators that may be added to plots and by default, Operators are added to *all* plots. To change this behaviour, uncheck the 'Apply operators to all plots' box which is located underneath the plot list in the GUI. Then select the Pseudocolor plot, bring up the Operators menu by clicking on 'Operators' and select 'ThreeSlice' from the 'Slicing' submenu. Again, click on 'Draw' to update the plots and notice how the box has now been sliced. We can move the slices to more suitable positions by editing the operator attributes. Click on the little triangle to the left of the Pseudocolor plot to reveal the list of elements that have been applied to it. Next, double-click the 'ThreeSlice' element to bring up the attribute window. Change the values to $X = 0.3$ and $Y = 0.3$, leaving $Z = 0$. Apply the changes and dismiss the dialog to see the result.

You can now create an image of the plots as shown in the window. First, adjust the save options to your needs in the 'Set Save options' dialog which is accessible from the 'File' menu. Then select 'Save Window' and you should find an image file with the name and location as entered in the options dialog.

8.3.2 Using the *VisIt* CLI (command line interface)

We will now perform exactly the same steps as in the last section but using the *python* interface of *VisIt* instead of the GUI. Start up the CLI by issuing

```
visit -cli
```

You should now see an empty visualization window but unlike in the previous section there will be no graphical user interface but a *python* command line instead. Enter the following commands, one by one, noticing the changes in the visualization window after every block of commands:

```
OpenDatabase("deform.vtu")
AddPlot("Pseudocolor","stress")
DrawPlots()

p=PseudocolorAttributes()
p.centering=p.Nodal
SetPlotOptions(p)

AddOperator("ThreeSlice")
DrawPlots()

t=ThreeSliceAttributes()
t.x=0.3
t.y=0.3
SetOperatorOptions(t)

AddPlot("Vector", "disp")
DrawPlots()

v=VectorAttributes()
v.useStride=1
SetPlotOptions(v)

s=SaveWindowAttributes()
#change settings as required
SaveWindow()
exit()
```

All but the last call to `DrawPlots()` is not required and was only put there for demonstrating the effects of the commands. You can save these commands to a file, e.g. `deformVis.py` and let *VisIt* process them non-interactively like so:

```
visit -cli -nowin -s deformVis.py
```

The `-nowin` option prevents the visualization window from being shown which is not required since the purpose of the script is to save an image file.

Obviously, we have barely touched on the powerful features of *VisIt* and this section was only meant to give you a minimal introduction. The *VisIt* website has a reference manual for the *python* interface that explains how to perform other operations programmatically, such as changing the view.

CHAPTER

NINE

Using Trilinos

Trilinos has a number of packages and provides a large collection of both direct and indirect solvers. We refer the reader to [22] for details. Escript needs to be installed with Trilinos to be able to use the Trilinos solvers. See the install guide for details. We show a few examples for the Trilinos options with a simple example.

Consider Laplacian in domain ($\Omega \in \mathbb{R}^3$) with a simple right hand side,

$$-\nabla^t \, \nabla u = 1, \qquad \text{in } \Omega, \tag{9.1}$$

$$u = 0, \qquad \text{on } \Gamma_D, \tag{9.2}$$

$$\mathbf{n}^t \, \nabla u = 0, \qquad \text{on } \Gamma_N, \tag{9.3}$$

with $\mathbf{n}$ the outward normal, with Γ_D the left boundary and $\Gamma = \partial\Omega \backslash \Gamma_D$. Gravity forward weak form of PDE (9.1)-(9.3), where (,) is the standard L^2 inner product on Ω, is

$$(\nabla u \, , \, \nabla v) = -(f \, , \, v), \tag{9.4}$$

for all admissible potential functions v.

For this example, we just consider a simple, unstructured, 3D domain. The mesh is created with GMSH using simplemesh.geo (file 9.1) see /escript/doc/examples/usersguide/simplemesh.geo. To test AMG, a finer mesh is created using /escript/doc/examples/usersguide/simplemeshfine.geo.

Listing 9.1: simplemesh.geo

```
// dimensions and mesh size
xdim = 100.;
ydim = 200.;
zdim = 50.;
mtop = 2.;
mbase = 5.;

//Points
Point(1) = {0., 0., 0., mbase};
Point(2) = {xdim, 0., 0., mbase};
Point(3) = {0., ydim, 0., mbase};
Point(4) = {xdim, ydim, 0., mbase};
Point(5) = {0., 0., zdim, mtop};
Point(6) = {xdim, 0., zdim, mtop};
Point(7) = {0., ydim, zdim, mtop};
Point(8) = {xdim, ydim, zdim, mtop};

//Lines and surfaces
Line(1) = {1, 2};
Line(2) = {3, 4};
Line(3) = {1, 3};
Line(4) = {2, 4};
Line(5) = {5, 6};
Line(6) = {7, 8};
Line(7) = {5, 7};
Line(8) = {6, 8};
Line(9) = {1, 5};
Line(10) = {3, 7};
Line(11) = {2, 6};
Line(12) = {4, 8};
Line Loop(1) = {-1, 3, 2, -4};
Plane Surface(1) = {1};
Line Loop(2) = {5, 8, -6, -7};
Plane Surface(2) = {2};
Line Loop(3) = {1, 11, -5, -9};
Plane Surface(3) = {3};
Line Loop(4) = {-2, 10, 6, -12};
Plane Surface(4) = {4};
Line Loop(5) = {-3, 9, 7, -10};
Plane Surface(5) = {5};
Line Loop(6) = {4, 12, -8, -11};
Plane Surface(6) = {6};

// domain
Surface Loop(1) = {1:6};
Volume(1) = {1};
```

The most basic escript script to solve this pde using escript defaults is in program 9.2. The computed solution is saved in a silo file "asimple.silo" and can be visualized using *VisIt*.

Listing 9.2: basic solve using defaults only

```
from esys.escript import *
from esys.weipa import saveVTK, saveSilo
from esys.escript.linearPDEs import LinearSinglePDE
from esys.finley import ReadGmsh

domain=ReadGmsh("simplemesh.msh", 3,  optimize=True )

pde = LinearSinglePDE(domain, isComplex=False)
pde.setSymmetryOn()
x = domain.getX()
pde.setValue(A=kronecker(3), Y = 1., q = whereZero(x[0]-inf(x[0])))

u=pde.getSolution()
saveSilo("asimple", u=u)
```

This script uses default Trilinos solver PCG with Jacobi preconditioner. (It takes 271 iterations to reach the default tolerance of 10^{-8}). Tolerance and solver output can be controlled by adding to the basic listing.

Listing 9.3: tolerance

```
options = pde.getSolverOptions()
options.setTolerance(1e-8)
options.setVerbosityOn()
```

9.1 Direct Solvers

There are a number of options that can be set for solving the pde. The simplest way to choose a direct solver is to use the default Trilinos direct solver KLU2. Escript can only use this feature if it is available in Trilinos.

To choose the default Trilinos direct solver, set the tolerance for the PDE solver to 10^{-8}, use the Trilinos suite of solvers and output information about the solvers, we add code to listing 9.2.

Listing 9.4: default Direct

```
options = pde.getSolverOptions()
options.setSolverMethod(SolverOptions.DIRECT)
options.setPackage(SolverOptions.TRILINOS)
options.setVerbosityOn()
```

The default Trilinos Direct solver is the Amesos2 LU factorisation KLU. It is a serial, unsymmetric sparse, partial-pivoting, direct matrix solver. The last line above ensures that the output includes details of the methods used.

To use SUPERLU the code that needs to be added to listing 9.2 is

Listing 9.5: SuperLU

```
options = pde.getSolverOptions()
options.setTolerance(1e-8)
options.setPackage(SolverOptions.TRILINOS)
options.setSolverMethod(SolverOptions.DIRECT_SUPERLU)
options.setVerbosityOn()
```

Trilinos is the default solver package so the setPackage line is not really necessary.

9.2 Preconditioned Conjugate Gradient

If we want to use preconditioned conjugate gradient, then we remover the DIRECT solver line and add the line

Listing 9.6: Preconditioned conjugate gradient defaults

```
options.setSolverMethod(SolverOptions.PCG)
```

This takes 271 iterations to reach tolerance using a Jacobi preconditioner, BELOS Pseudo Block CG with Ifpack2. To change the preconditioner to Gauss-Siedel, we add the line

```
options.setPreconditioner(SolverOptions.GAUSS_SEIDEL)
```

This takes 120 iterations to reach tolerance.

9.3 Multigrid

Geometric multigrid methods were introduced for structured grids to maximise the advantages of iterative methods for solving matrix equations derived from discretised partial differential equations. Iterative methods for these problems are effective in reducing oscillatory error but stall for smooth error and smooth error appears oscillatory when restricted to a coarser grid. The idea is to have a succession of grids from fine to coarse and to remove error by iterating on each of the grids in turn reducing the oscillatory error on that grid. The coarsest grid is chosen small enough so that it can be quickly solved directly or iteratively. If n is the size of the problem then a multigrid algorithm is order n.

The matrix equation, derived from the PDE, is

$$\mathbf{A}\mathbf{u} = \mathbf{f}, \tag{9.5}$$

where $\mathbf{u}$ is the unknown $n \times 1$ vector, $\mathbf{A}$ is the $n \times n$ matrix and $\mathbf{f}$ is the nown right hand side $n \times 1$ vector. The residual $\mathbf{r}$ is defined

$$\mathbf{r} = \mathbf{f} - \mathbf{A}\mathbf{u} \tag{9.6}$$

and the residual equation is defined

$$\mathbf{A}\mathbf{e} = \mathbf{r}, \tag{9.7}$$

where $\mathbf{e} = \mathbf{u}^* - \mathbf{u}$ is the error and $\mathbf{u}^*$ is the exact solution. Relaxing on 9.7) with the residual as the right hand side is the same as relaxing on (9.5) with the original right and side. We use superscripts on these terms to indicate the discretization representing element length, h, for a fine grid and H for one level coarser discretization, $h < H$. For a structured mesh, $H = \frac{h}{2}$ and $\mathbf{A}^h$ is an $n \times n$ matrix and $\mathbf{A}^H$ is an $N \times N$ matrix obtained in the coarsening process with $N < n$. Prolongation operator, $\mathbf{P}^h_H$, an $n \times N$ matrix, is used to interpolate the error from the coarse grid to the fine grid and restriction operator $\mathbf{P}^H_h$, an $N \times n$ matrix, is used to restrict the residual from the fine grid to the coarse grid. It is not necessary for the restriction operator to be the transpose of the interpolation operator.

AMG($\mathbf{u}^h; \mathbf{A}^h, \mathbf{f}^h$)	
$\mathbf{A}^h\mathbf{u}^h = \mathbf{f}^h$	**p steps pre-smoothing iteration**
$\mathbf{r}^h = \mathbf{f}^h - \mathbf{A}^h\mathbf{u}^h$	**fine grid residual**
$\mathbf{r}^H = \mathbf{R}_h^H\mathbf{r}^h$	**restrict residual**
$\mathbf{A}^H = \mathbf{R}_h^H\mathbf{A}^h\mathbf{P}_H^h$	**restrict operator**
if not coarsest	
AMG($\mathbf{e}^H; \mathbf{A}^H, \mathbf{r}^H$)	**recursion**
else	
$\mathbf{A}^H\mathbf{e}^H = \mathbf{r}^H$	**coarsest solve**
$\mathbf{e}^h = \mathbf{P}_H^h\mathbf{e}^H$	**interpolate error**
$\mathbf{u}^h = \mathbf{u}^h + \mathbf{e}^h$	**fine grid correction**
$\mathbf{A}^h\mathbf{u}^h = \mathbf{f}^h$	**q steps post-smoothing iteration**
return $\mathbf{u}^h$	

Table 9.1: An AMG V(p.q) cycle algorithm to solve $\mathbf{Au} = \mathbf{f}$, where h and H represent grid sizes with $h < H$.

The coarse grid matrix operator $\mathbf{A}^H$ is computed by multiplying the fine grid matrix operator $\mathbf{A}^h$ on the left by the restriction operator and the right by the interpolation operator resulting in an $N \times N$ matrix.

The MG algorithm is best described using a recursion algorithm. After iteration on a fine grid, the fine grid error is smooth and the residual can be restricted to a coarse grid. The error on the coarse grid appears more oscillatory and is (smoothed) reduced by iteration. If this is the coarsest grid then the matrix equation is solved using a direct method or sufficient iterations of the solver to get within discretization error, otherwise the algorithm is called again with the coarse grid replacing the fine grid and the next coarser grid as the coarse grid. The coarse grid error, is interpolated to the fine grid where it corrects the fine grid approximation and post-smoothing iteration smooths the error. For structured grids, the choices for interpolating from a coarse grid to a fine grid or restricting from a fine grid to a coarse grid are reasonably obvious, see [1].

Algebraic multigrid methods were developed for unstructured grids and do not reference the grid but instead use interpolation and restriction operators derived from the matrix (see [1, 18, 24, 23]). The terms "coarse grid" and "fine grid" are still used but do not refer to actual grids. "Smooth error" is defined to be the error not reduced by iteration and "oscillatory error" is the error reduced by iteration. Coarse levels are chosen from the relative sizes of the off diagonal terms in the fine matrix. Once the coarse grid is chosen the restriction and interpolation operators are computed. Restriction operators need to be chosen so that "smooth error" on a fine grid will appear "oscillatory" on a coarse grid ensuring that it can be reduced by iterating on this grid. There are a number of algorithm options available in Trilinos to compute the coarse grid and the choice will depend on the original PDE and smoothing options. For any multigrid method, there are basic choices:

- pre-smoothing iterative solver and number of iterations
- post-smoothing iterative solver and number of iterations
- choosing coarse grids
- number of coarse grids or size of coarsest grid
- interpolation operator
- restriction operator
- coarsest grid solver
- cycle type

We use Trilinos solvers and it is possible to access Trilinos options either within the escript script or, if more complicated control is needed, in an XML file. There are many Trilinos packages that can be used by escript including MueLu - setup of AMG, Belos - linear solvers - Pseudo Block CG, Ifpack2 - iterative solvers (Jacobi, Gauss-Siedel), Amesos2 - direct solvers for coarse level and Voltan or Voltan2 - repartitioning for caorse grids

number of equations	1
problem: symmetric	True, $\mathbf{R}_h^H = (\mathbf{P}_H^h)^t$
pre-smoothing iterative solver	Symmetric Gauss-Seidel
post-smoothing iterative solver	Symmetric Gauss-Seidel
pre-smoothing iterations	1
post-smoothing iterations	1
minimum aggregate size	2
maximum aggregate size	unlimited
aggregation	uncoupled
maximum number of levels	10
maximum size of coarsest grid	2000
choosing coarse grids	classical smoothed aggregation
coarsest grid solver	SuperLU
cycle type	V(1,1)

Table 9.2: Default parameters for AMG-PCG

9.3.1 Default MueLu Trilinos options for algebraic multigrid preconditioned conjugate gradient (AMG-PCG)

More detail on the various options can be found in the MUELU user guide and other Trilinos user guides [22]. MUELU uses other Trilinos packages and to access these parameters the XML file must be used. Recall $\mathbf{R}_h^H$ is the restriction operator and $\mathbf{P}_H^h$ is the interpolation (prolongation) operator. The default values used are shown in Table 9.2. To use AMG-PCG with default paramerters we use script 9.7.

Listing 9.7: basic Trilinos PCG-AMG defaults only script

```
from esys.escript import *
from esys.weipa import saveVTK, saveSilo
from esys.escript.linearPDEs import LinearSinglePDE
from esys.finley import ReadGmsh

domain=ReadGmsh("simplemesh.msh", 3,  optimize=True )

pde = LinearSinglePDE(domain, isComplex=False)
pde.setSymmetryOn()
x = domain.getX()
pde.setValue(A=kronecker(3), Y=1, q=whereZero(x[0]-inf(x[0])))

options = pde.getSolverOptions()
options.setPackage(SolverOptions.TRILINOS)
options.setSolverMethod(SolverOptions.PCG)
options.setPreconditioner(SolverOptions.AMG)

u=pde.getSolution()
saveSilo("asimple",u=u)
```

Only two grids were used for the simple mesh. The first coarse grid, replaces, if possible, 27 fine grid nodes represented by one coarse grid node. In 2D this would be 9 fine grid nodes represented with one coarse grid node. The ratio of the fine to coarse grid in this example is 19.21. For a finer mesh, with mtop = 2 and mbase=1 in the geo file, 3 levels of grids and the coarsest grid is solved with a direct method. AMG-PCG took 11 iterations to reach tolerance.

9.3.2 Altering MUELU parameters

To access MUELU parameters in the python script we use the general form

```
options.setTrilinosParameter( "A", "B")
```

where "A" is the Trilinos parameter and "B" is its string value. It is extremely important to have correct spaces in the strings.

9.3.2.1 Debug output

Options are "none", "low", "medium", "high" and "extreme".

```
options.setTrilinosParameter("verbosity", "low")
```

Options are
`"low"` - setup time,
`"medium"` - basic AMG data, mesh sizes, smoothers + "low"
`"high"` - input data, relaxation solvers and data, aggregate data + "medium"
`"extreme"` - may include solver details that MueLu calls + "high "

9.3.2.2 Problem type

Changes default multigrid algorithm, block size and smoother. Options are

- `"unknown"`: default
- `"Poisson-2D" or "Poisson-3D"` : using smoothed aggregation, Chebyshev smoother and block size of 1,
- `"Elasticity-2D" and "Elasticity-3D"`: using smoothed aggregation, Chebyshev smoother and block size of 2 or 3 respectively,
- `"Poisson-2D-complex" and "Poisson-3D-complex"`: using smoothed aggregation, symmetric Gauss-Seidel and block size 1,
- `"Elasticity-2D-complex" and "Elasticity-3D-complex"`: using smoothed aggregation, symmetric Gauss-Seidel and block size 2 and 3 respectively,
- `"ConvectionDiffusion"`: using Petrov-Galerkin AMG, Gauss-Seidel and 1 block,
- `"MHD"`: using unsmoothed aggregation and Additive Schwarts method with one level of overlap and ILU(0) as a subdomain solver.

```
options.setTrilinosParameter("problem:type", "Poisson-3D")
```

9.3.2.3 number of equations

Number of PDE equations at each grid node.

```
options.setTrilinosParameter("number of equations", 1)
```

9.3.2.4 AMG algorithm

The multigrid algorithm for computing the coarse levels and interpolation and restriction operators is controlled with `"multigrid algorithm"`. The default value is smoothed aggregation and is selected with `"sa"` and a damping factor can be imposed. The other options are `"unsmoothed"`, no Jacobi prolongation improvement step; `"pg"`, $\mathbf{A}$ prolongation smoothing and $\mathbf{A}^T$ restriction smoothing; `"emin"` basis functions for grid transfer using energy constrained minimisation; `"interp"`, piecewise constant ("interpolation order" set to 0) or linear interpolation (`"interpolation order"` set to 1) from coarse to fine and is only possible with structured aggregation; and `"semicoarsen"`, coarsen fully in z direction (will need to set rate in this direction). It is also possible to use an implicit transpose for the restriction operator.

```
# smoothed aggregation
options.setTrilinosParameter("multigrid algorithm", "sa")
options.setTrilinosParameter("sa: damping factor", 1.3)
options.setTrilinosParameter("sa: use filtered matrix", True)
options.setTrilinosParameter("filtered matrix: use lumping", True)
options.setTrilinosParameter("filtered matrix: reuse eigenvalue", True)
# unsmoothed
```

```
options.setTrilinosParamter("multigrid algorithm", "unsmoothed")
# pg
options.setTrilinosParameter("multigrid algorithm", "pg")
# interpolation
options.setTrilinosParameter("multigrid algorithm", "interp")
options.setTrilinosParameter("interp: interpolation order", 1)
                                                           # 0, 1
options.setTrilinosParameter("interp: build coarse coordinates", True)
# emin
options.setTrilinosParameter("multigrid algorithm", "emin")
options.setTrilinosParameter("emin: iterative method", "cg")
                                                      # "cg", "gmres", "sd"
options.setTrilinosParameter("emin: num iterations", 2)
options.setTrilinosParameter("emin: num reuse iterations", 1)
options.setTrilinosParameter("emin: pattern", "AkPtent")
options.setTrilinosParameter("emin: pattern order", 1)
# semicoarsen
options.setTrilinosParameter("multigrid algorithm", "semicoarsen")
options.setTrilinosParameter("semicoarsen: coarsen rate", 3)
#
options.setTrilinosParameter("transpose: use implicit", False)
```

9.3.2.5 Maximum levels, coarse mesh, coarse solver

It is possible to limit the size of the coarsest level as well as limit the number of levels. The default for the size of the coarsest level is 2000. So once the size of the coarse level is less than 2000 then no more coarse levels are created. Additionally, it is possible to limit the number of levels by setting "max levels" in the hierarchy. This includes the fine grid. The default value is 10 but depending on fine grid size, changing this could improve performance of the algorithm. The coarsest level can be solved using a direct solver. Possibilities are KLU, KLU2, SuperLU, SuperLU_dist, Umfpack and Mumps

```
options.setTrilinosParameter("max levels", 10)
options.setTrilinosParameter("coarse: max size", 2000)
options.setTrilinosParameter("coarse: type", "SuperLU")
```

9.3.2.6 Aggregation

It is possible to influence aggregation options. If the fine mesh is a structured grid then aggregates can be created in a "structured" way and the aggregation attempts to form hexahedral coarse levels. This uses a default coarsening rate of 3 in each direction. The option "hybrid" allows user determined "structured" or "unstructured" aggregation for each level, To get optimal size coarse mesh (3^d in d dimensions) "uncoupled" or "coupled" is used with "coupled" allowing aggregates to span processors. It is suggested that "coupled" should be used with care. "brick" attempts to make rectangular aggregates. Some of the options are below with more detail and more options in the MUELU user guide.

```
options.setTrilinosParameter("aggregation: type", "structured")
options.setTrilinosParameter("aggregation: ordering", "natural")
                                                 # "natural", "graph", "random"
options.setTrilinosParameter("aggregation: drop scheme", "classical")
                                                 # "classical", "distance laplacian"
options.setTrilinosParameter("aggregation: drop tol", 0.0)
options.setTrilinosParameter("aggregation: min agg size", 2)
options.setTrilinosParameter("aggregation: max agg size", -1)
                                                # -1 means unlimited
options.setTrilinosParameter("aggregation: Dirichlet threshold", 1e-5)
```

9.3.2.7 Relationship between $\mathbf{R}_h^H$ and $\mathbf{P}_H^h$

For $\mathbf{R}_h^H = (\mathbf{P}_H^h)^t$

```
options.setTrilinosParameter("problem: symmetric", True)
```

this is the default.

9.3.2.8 Smoothers

In the escript script it is possible to choose smoother type, "RELAXATION", "CHEBYSHEV" and "ILUT" or "RILUT" but for more specific control the XML file needs to be used. It is possible to use different pre and post smoothers. "RELAXATION" could use Jacobi, Gauss-Seidel, symmetric Gauss-Seidel, multithreaded Gauss-Seidel. To specify which one the XML file must be used. Some examples for this are

```
options.setTrilinosParameter("smoother: pre or post", "both")
options.setTrilinosParameter("smoother: type", "RELAXATION")
options.setTrilinosParameter("smoother: pre type", "CHEBYSHEV")
options.setTrilinosParameter("smoother: post type", "RELAXATION")
```

9.3.2.9 Cycle type

Allowable cycle types are "V" and "W". The default is a "V" cycle.

```
options.setTrilinosParameter("cycle type", "V")
```

9.3.2.10 reuse

If multiple PDEs are being solved the reuse strategy can use elements of previous computations. The level of reuse varies from none to full. Options are `"none"`; `"S"`, symbolic coarse levels information; `"tP"`, reuse tentative prolongation operator; `"emin"`, reuse old prolongator for initial guess; `"RP"`, reuse smoothed restrictor and prolongator; `"RAP"`, compute only fine level smoothers and reuse all other operators, and `"full"`, reuse everything.

```
options.setTrilinosParameter("reuse: type", "full")
```

9.3.2.11 repartitioning

If there are multiple processors it might be benificial to repartition as the mesh are coarsened including perhaps using only one processor for the coarsest grid. This is to reduce communication costs for the caorser grids.

```
options.setTrilinosParameter("repartition: enable", False)
options.setTrilinosParameter("repartition: start level", 2)
options.setTrilinosParameter("repartition: min rows per proc", 800)
options.setTrilinosParameter("repartition: max imbalance", 1.2)
options.setTrilinosParameter("repartition: remap parts", True)
options.setTrilinosParameter("repartition: rebalance P and R", False)
```

9.3.3 commands in XML file

All the previous commands can be placed into an XML file. The XML file option allows the user to choose parameters for the programs that MUELU calls, so more control is possible on the iterative solvers.

```
from esys.escript import *
from esys.weipa import saveVTK, saveSilo
from esys.escript.linearPDEs import LinearSinglePDE
from esys.finley import ReadGmsh

domain=ReadGmsh("simplemesh.msh", 3,  optimize=True )

pde = LinearSinglePDE(domain, isComplex=False)
pde.setSymmetryOn()
x = domain.getX()
pde.setValue(A=kronecker(3), Y=1, q=whereZero(x[0]-inf(x[0])))
```

```
options = pde.getSolverOptions()
options.setPackage(SolverOptions.TRILINOS)
options.setSolverMethod(SolverOptions.PCG)
options.setPreconditioner(SolverOptions.AMG)
options.setTrilinosParameter("xml parameter file", "simplebob.xml")

u=pde.getSolution()
saveSilo("asimple",u=u)
```

It is possible to specify how many sweeps of the iterative solvers for pre and post smoothing and we could choose cycles with different numbers of pre and post sweeps. Amesos2 provides direct solvers including superLU and Mumps. MueLu passes parameters directly to solver library. To specify CHEBYSHEV parameters, for example, an XML file must be used.

Ifpack2 or Ifpack provides iterative matrix solvers Jacobi, Gauss Seidel, polynomial, distribution relaxation, domain decomposition solvers and incomplete factorizations.

A very simple XML file is in file 9.8

Listing 9.8: simplebob.xml

```
<ParameterList name="MueLu">
  <Parameter name="verbosity"                 type="string"    value="high"/>
  <Parameter name="max levels"                type="int"       value="4"/>
  <Parameter name="coarse: max size"          type="int"       value="200"/>
  <Parameter name="multigrid algorithm"       type="string"    value="sa"/>
  <Parameter name="reuse: type"               type="string"    value="full"/>
  <Parameter name="transpose: use implicit"   type="bool"      value="true"/>
  <Parameter name="sa: damping factor"        type="double"    value="0.1"/>
  <Parameter name="sa: use filtered matrix"   type="bool"      value="true"/>
</ParameterList>
```

A more complicated example that controls the number of pre and post sweeps is in the listing 9.9

Listing 9.9: complicatedbob.xml

```
<ParameterList name="MueLu">
  <!--    General    -->
  <Parameter name="verbosity"                 type="string"    value="high"/>
  <Parameter name="max levels"                type="int"       value="4"/>
  <Parameter name="coarse: max size"          type="int"       value="200"/>
  <Parameter name="multigrid algorithm"       type="string"    value="sa"/>
  <Parameter name="reuse: type"               type="string"    value="full"/>
  <Parameter name="transpose: use implicit"   type="bool"      value="true"/>
  <Parameter name="sa: damping factor"        type="double"    value="0.1"/>
  <Parameter name="sa: use filtered matrix"   type="bool"      value="true"/>

  <!-- Smoothing -->
  <Parameter name="smoother: pre or post"        type="string"  value="both"/>

  <Parameter name="smoother: pre type"           type="string"  value="CHEBYSHEV"/>
  <ParameterList name="smoother: pre params">
    <Parameter name="relaxation: type"           type="string"  value="Symmetric Gauss-Seidel"/>
    <Parameter name="relaxation: sweeps"         type="int"     value="5"/>
    <Parameter name="relaxation: damping factor" type="double"  value="0.9"/>
  </ParameterList>

  <ParameterList name="smoother: params">
    <Parameter name="chebyshev: degree"            type="int"     value="3"/>
    <Parameter name="chebyshev: ratio eigenvalue" type="double"  value="15"/>
  </ParameterList>

  <Parameter name="smoother: post type"           type="string"  value="RELAXATION"/>
  <ParameterList name="smoother: post params">
```

```
    <Parameter name="relaxation: type"           type="string"  value="Symmetric Gauss-Seidel"/>
    <Parameter name="relaxation: sweeps"         type="int"     value="5"/>
    <Parameter name="relaxation: damping factor" type="double"  value="0.9"/>
  </ParameterList>

  <!-- Aggregation -->
  <Parameter name="aggregation: type"           type="string"  value="uncoupled"/>
  <Parameter name="aggregation: min agg size"   type="int"     value="3"/>
  <Parameter name="aggregation: max agg size"   type="int"     value="27"/>

  <!--  for different level parameter list -->
  <ParameterList name="level 2">
    <Parameter name="smoother: type" type="string" value="CHEBYSHEV"/>
  </ParameterList>

</ParameterList>
```

CHAPTER

TEN

The escript symbolic toolbox

10.1 Introduction

esys.escript builds on the existing Sympy [20] symbolic maths library to provide a Symbol class with support for esys.escript Data objects. Symbol objects act as placeholders for a single mathematical symbol, such as x, or for arbitrarily complex mathematical expressions such as c*x**4 + alpha*exp(x) - 2*sin(beta * x), where alpha, beta, c, and x are also symbols (the symbolic “atoms" of the expression). With the help of the Evaluator class, these symbols and expressions can be evaluated by substituting numeric values and/or esys.escript Data objects for the atoms. Escript’s Symbol class has a shape (and thus a rank) as well as a dimensionality. Symbols are useful to perform mathematical simplifications, compute derivatives, take gradients and in the case of esys.escript describe PDEs. As an example of how the symbolic toolbox can be used, consider the following code extract.

```
import esys.escript as es
u = es.Symbol('u')
p = 2*u**2 + 3*u + 1
p2 = es.sin(u)
p3 = p.diff(u)              # p3 = derivative of p with respect to u
evalu = es.Evaluator()
evalu.addExpression(p)
evalu.addExpression(p2)
evalu.addExpression(p3)
evalu.subs(u=2*es.symconstants.pi)
evaluated=evalu.evaluate()
print("p3 =", p3)            # The symbols can be printed, this line will print p3.
print(evaluated)
```

Running this code outputs:
```
p3 = 4*u + 3
(1 + 6*pi + 8*pi**2, 0, 3 + 8*pi).
```
To get the numeric value of the expression we replace
`evalu.evaluate()`
with
`evalu.evaluate(evalf=True).`
This results in (98.806, 0, 28.132). The use of these Symbols becomes more interesting in the context of escript when they are integrated with escript Data objects.

10.2 NonlinearPDE

The `NonlinearPDE` class in escript makes use of the escript `Symbol` class and allows for the solution of PDEs of the form:

$$-X_{ij,j} + Y_i = 0 \tag{10.1}$$

where X and Y are both functions of $u_{k,j}$ and u_k, and u is the unknown function implemented as a `Symbol`. $\nabla \cdot x$ denotes divergence of x. The `NonlinearPDE` class uses the `Symbol` class to solve the nonlinear PDE given in Equation (10.1). The class incorporates Newton's method to find the zeroes of the left hand side of Equation (10.1) and as a consequence finding the X and Y which satisfy Equation (10.1). Consecutive updates are calculated until the equation is satisfied to the desired level of accuracy. The solution to each update step involves solving a linear PDE. The `NonlinearPDE` class uses X and Y to produce the coefficients of the linear PDE for the update step. The linear PDE class given in Section 4.1 is used to solve the linear PDEs from the update step. The coefficients of the linear PDE to be solved are calculated as follows:

$$A_{ijkl} = \frac{\partial \mathrm{X}_{ij}}{\partial u_{k,l}}, B_{ijkl} = \frac{\partial \mathrm{X}_{ij}}{\partial u_k}, C_{ijkl} = \frac{\partial \mathrm{Y}_{ij}}{\partial u_{k,l}}, D_{ijkl} = \frac{\partial \mathrm{Y}}{\partial u_k}$$

10.3 2D Plane Strain Problem

The `NonlinearPDE` class can be used to solve a 2D plane strain problem. In continuous media, stress is given by Lamé's Equation (10.2).

$$-\sigma_{ij,j} = f \tag{10.2}$$

Hook's Law provides a relation between σ and ϵ in the following form

$$\begin{bmatrix} \sigma_{00} \\ \sigma_{11} \\ \sigma_{01} \end{bmatrix} = \begin{bmatrix} c_{00} & c_{01} & c_{05} \\ c_{01} & c_{11} & c_{15} \\ c_{05} & c_{15} & c_{55} \end{bmatrix} \begin{bmatrix} \epsilon_{00} \\ \epsilon_{11} \\ 2\epsilon_{10} \end{bmatrix} \tag{10.3}$$

Where $\epsilon = symmetric(grad(u))$ or $\epsilon_{ij} = \frac{1}{2}\left(\frac{\partial u_i}{\partial x_j} + \frac{\partial u_j}{\partial x_i}\right)$, u is the unknown function and c_{ij} is the stiffness matrix. To fit this to the nonlinear PDE class' standard form, X is set to $C \times symmetric(grad(u))$. The following *python* extract shows how an example 2D plane strain problem can be set up.

```
from esys.escript import *
from esys.finley import Rectangle
#set up domain and symbols
mydomain = Rectangle(l0=1.,l1=1.,n0=10, n1=10)
u = Symbol('u',(2,), dim=2)
q = Symbol('q', (2,2))
sigma = Symbol('sigma',(2,2))
theta = Symbol('theta')
# q is a rotation matrix represented by a Symbol. Values can be substituted for
# theta.
q[0,0]=cos(theta)
q[0,1]=-sin(theta)
q[1,0]=sin(theta)
q[1,1]=cos(theta)
# Theta gets substituted by pi/4 and masked to lie between .3 and .7 in the
# vertical direction. Using this masking means that when q is used it will apply
# only to the specified area of the domain.
x = Function(mydomain).getX()
q=q.subs(theta,(symconstants.pi/4)*whereNonNegative(x[1]-.30)*
    whereNegative(x[1]-.70))
# epsilon is defined in terms of u and has the rotation applied.
epsilon0 = symmetric(grad(u))
epsilon = matrixmult(matrixmult(q,epsilon0),q.transpose(1))
# For the purposes of demonstration, an arbitrary c with isotropic constraints
# is chosen here. In order to act as an isotropic material c is chosen such that
# c00 = c11 = c01+c1+2*c55
```

```
c00 = 10
c01 = 8; c11 = 10
c05 = 0; c15 = 0; c55 = 1
# sigma is defined in terms of epsilon
sigma[0,0] = c00*epsilon[0,0]+c01*epsilon[1,1]+c05*2*epsilon[1,0]
sigma[1,1] = c01*epsilon[0,0]+c11*epsilon[1,1]+c15*2*epsilon[1,0]
sigma[0,1] = c05*epsilon[0,0]+c15*epsilon[1,1]+c55*2*epsilon[1,0]
sigma[1,0] = sigma[0,1]
sigma0=matrixmult(matrixmult(q.transpose(1),sigma),q)
# set up boundary conditions
x=mydomain.getX()
gammaD=whereZero(x[1])*[1,1]
yconstraint = FunctionOnBoundary(mydomain).getX()[1]
# The nonlinear PDE is set up, the values are substituted in and the solution is
# calculated, y represents an external shearing force acting on the domain.
# In this case a force of magnitude 50 is acting in the x[0] direction.
p = NonlinearPDE(mydomain, u, debug=NonlinearPDE.DEBUG0)
p.setValue(X=sigma0,q=gammaD,y=[-50,0]*whereZero(yconstraint-1),r=[1,1])
v = p.getSolution(u=[0,0])
```

The way in which the rotation matrix q is set up demonstrates the seamless integration of escript symbols and `Data` objects. A `Symbol` is used to set up the matrix, the values for theta are then later substituted in. The example also demonstrates how the symbolic toolbox can be used as an aid to easily move from a mathematical equation to an escript data object which can be used to do numerical calculations. Running the script calculates the unknown function u and assigns it to v. We can use v to calculate the stress and strain. Table 10.1, shows the result of running the above script under varying values of c and theta. Both isotropic and anisotropic cases are considered. For the anisotropic case, c_{ij} is chosen such that $c00 = c11 = c01 + c1 + 2 * c55$ does not hold. Two values of theta are also considered; one with a masked 60°rotation in the middle and one with no rotation. The last row of the table shows the difference between rotation in the middle and no rotation. In the isotropic case it can be seen that there is no difference in the output when the rotation is applied. There is however, an obvious difference when the anisotropic case is considered.

10.4 Classes

A number of classes are associated with escript symbols. A detailed listing of the definitions and usage is provided below.

10.4.1 Symbol class

class Symbol(symbol [, shape] [, Dim])

Defines a `Symbol` object. The first argument `symbol` is a string given to represent the `Symbol`. The string typically matches the name of the object, for instance `u=Symbol('u')`. Next optional `shape` argument defines whether the `Symbol` is a scalar, vector, matrix, or tensor and the length or size of it. `dim` is used to define the dimensionality of the object contained in the `Symbol`. For a `Symbol` definition `u = Symbol('u',(10,), dim=2)`, the value of u will be a vector with 10 components and the domain on which u will be used is 2-dimensional (this is relevant with operations such as `grad` where the number of spatial dimensions is important).

10.4.1.1 Symbol class methods

atoms([types])

Returns the atoms that form the current `Symbol`. By default, only objects that are truly atomic and cannot be divided into smaller pieces are returned: symbols, numbers, and number symbols like e and pi. It is possible to request atoms of any type via the `types` argument, however.

coeff(x [, expand=true])

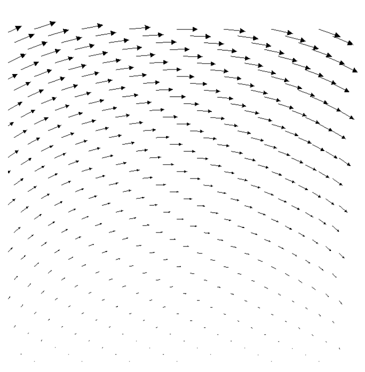
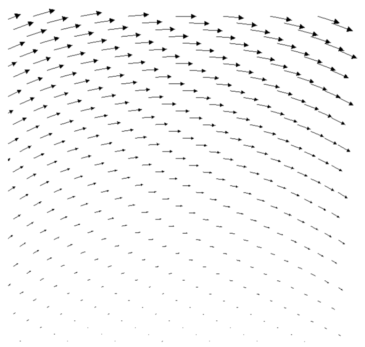
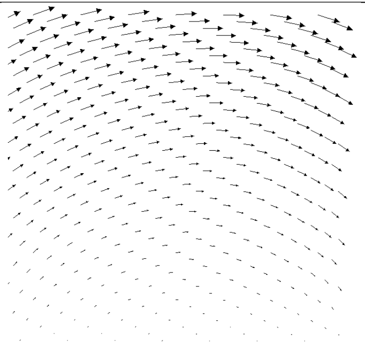
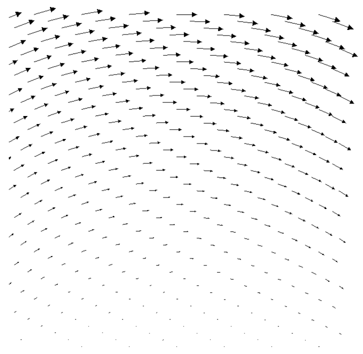
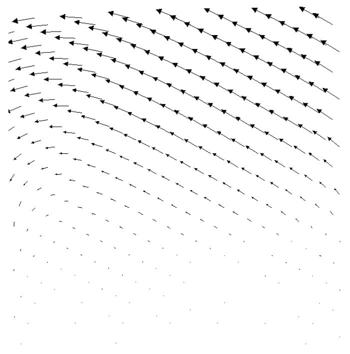
	Anisotropic	Isotropic
No rotation		
60° rotation		
difference		no difference

Table 10.1: Displacement vectors calculated using `NonlinearPDE`

Returns the coefficient of the term `x` or 0 if there is no `x`. If `x` is a scalar `Symbol` then `x` is searched in all components of this `Symbol`. Otherwise the shapes must match and the coefficients are checked component by component. For example:

```
x=Symbol('x', (2,2))
y=3*x
print y.coeff(x)
print y.coeff(x[1,1])
```

will print:

```
[[3 3]
 [3 3]]
[[0 0]
 [0 3]]
```

diff(symbols)
Takes the derivative of the `Symbol` object of which the method is called with respect to the symbols specified in the argument `symbols`.

evalf()
Applies the `sympy.evalf` operation on all elements of the `Symbol` which are of type or inherit from `sympy.Basic`.

expand()
Applies the `sympy.expand` operation on all elements in this `Symbol`.

getDim()
Returns the `Symbol`'s spatial dimensionality, or -1 if undefined.

getRank()
Returns the `Symbol`'s rank which is equal to the length of the shape.

getShape()
Returns the shape of this `Symbol`.

grad([where=none])
Returns the gradient of this `Symbol`. The `Symbol` must have a dimensionality defined in order for `grad` to work. As with the normal escript `grad` function a `FunctionSpace` can be specified using the `where` argument. The `FunctionSpace` should be wrapped in a `Symbol`. To do this, set up a `Symbol` and then use the `subs` function to substitute in the `FunctionSpace`.

inverse()
Find the inverse of the `Symbol` to which the function is applied. Inverse is only valid for square rank 2 symbols.

simplify()
Applies the `sympy.simplify` operation on all elements in this `Symbol`.

subs(old, new)
Substitutes or replaces a `Symbol` specified in `old` with whatever is in `new` for this `Symbol`. Consider:

```
import esys.escript as es
u=es.Symbol("u")
expr=2*u
expr.subs(u,2)
```

This prints 4.

trace(axis_offset)
Returns the trace of the `Symbol` object.

10.4.2 Evaluator class

The `Evaluator` class is intended to have a group of expressions added to it, substitutions can be made across all expressions and the expressions can then all be evaluated.

10.4.2.1 Evaluator class methods

class Evaluator([expressions])
An `Evaluator` object is initiated via `Evaluator()` with an optional argument of expressions to store.

addExpression(expression)
Adds an expression to this `Evaluator`.

evaluate([evalf=False][, args])
Evaluates all expressions in this `Evaluator` and returns the result as a tuple. `evalf` can be set to `True` to call `evalf` on any sympy symbols which may be part of the expression. `args` can be provided to make any substitutions before the expression is evaluated.

subs(old,new)

Substitutes or replaces a `Symbol` specified in `old` with whatever is in `new` for all expressions in the `Evaluator`.

10.4.3 NonlinearPDE class

class NonlinearPDE(domain, u)

Defines a general nonlinear, steady, second order PDE for an unknown function `u` on a given domain defined through a `Domain` object `domain`. `u` is a `Symbol` object. The general form is -div(X) + Y = 0

getSolution(subs)

Returns the solution of the PDE. `subs` contatins the substitutions for all symbols used in the coefficients including the initial value for the unknown `u`.

setOptions(opts)

```
  Allows setting options for the nonlinear PDE.
  The supported options are:
    tolerance
        error tolerance for the Newton method
    iteration_steps_max
        maximum number of Newton iterations
    omega_min
        minimum relaxation factor
    atol
        solution norms less than atol are assumed to be atol.
        This can be useful if one of your solutions is expected to
        be zero.
    quadratic_convergence_limit
        if the norm of the Newton-Raphson correction is reduced by
        less than quadratic_convergence_limit between two iteration
        steps, quadratic convergence is assumed.
    simplified_newton_limit
        if the norm of the defect is reduced by less than
        simplified_newton_limit between two iteration steps and
        quadratic convergence is detected, the iteration switches to the
        simplified Newton-Raphson scheme.
```

setValue([X][, Y] [, y] [, q][, r])

Assigns new values to coefficients. By default all values are assumed to be zero[1]. If the new coefficient value is not a `Data` object, it is converted into a `Data` object in the appropriate `FunctionSpace`.

10.4.4 Symconsts class

Symconsts provides symbolic constants for use in symbolic expressions. These constants are preferred to floating point implementation as they can cancel perfectly when mathematical expressions are evaluated, avoiding numerical imprecision.

```
usage:
  symconsts.pi this provides a Symbol object

Available constants currently include:
  pi and e
```

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

CHAPTER
ELEVEN

Escript `Splitworlds`

11.1 Introduction

By default, when `esys.escript` is used with multiple processes[1], it automatically uses all available resources to solve each PDE. When solving PDEs on high resolution domains, this is the most desirable solution technique. However there are cases, for example, geological inversion problems, for which it may be desirable to solve a large number of lower resolution problems simultaneously. To make these problems tractable, `esys.escript`'s splitworlds package allows you to subdivide `esys.escript`'s processes into smaller groups, named subworlds, that can each be assigned separate tasks.

For example, if escript is started with 8 MPI processes[2], then these could be subdivided into 2 groups of 4 processes, 4 groups of 2 or 8 groups of 1 node [3]. These smaller groups of processes are referred to as "subworlds" and a group of subworlds is referred to as a "splitworld".

11.2 The `Splitworld` class

The creation and management of a splitworld is managed using the splitworld class in the `esys.escript` module. Once the `esys.escript` module has been loaded, the command

```
sw=SplitWorld(n)
```

will initialise a "splitworld" that consists of n "subworlds". Each subworld in this splitworld controls $1/n^{th}$ of the resources available to escript. This command with throw an exception if the requested number of subworlds is not a factor of current number of MPI processes. If the number of MPI processes is not known in advance, you could alternatively run the command `Splitworld(getMPISizeWorld())` which will initialise as many subworlds as there are available MPI processes.

A "subworld" is the context in which a job executes. All processes that execute a particular job belong to the same subworld and the subworld provides the process with information about the spatial domain as well as some other variables. Note that creating a splitworld object does not prevent `esys.escript` from solving PDEs normally. You can write an escript that uses both modes of operation.

After a splitworld has been initialized, it is necessary to inform the splitworld what type of domain that the PDE is defined on. First load a domain object from an `esys.escript` module and then use the command `buildDomains` to instruct the splitworld what type of domain to create. For example, the command

[1]That is, when MPI is employed (probably in a cluster setting). This discussion is not concerned with using multiple threads within a single process (OpenMP). The functionality described here is compatible with OpenMP but orthogonal to it (in the same sense that MPI and OpenMP are orthogonal).

[2]see the `run-escript` launcher or your system's mpirun for details.

[3]1 group of 8 processes is also possible, but perhaps not particularly useful aside from debugging scripts.

```
from esys.finley import Rectangle
sw.buildDomains(Rectangle,n0=10,n1=10)
```

will load the rectangle object from the `esys.finley` module into memory and then instruct the splitworld `sw` that each subworld is going to solve PDEs on a `esys.finley` rectangle. It is not possible to create subworlds inside the same splitworld class that utilize different domain types.

Each subworld stores domain information independently. This means that objects built directly (or indirectly) on domains inside a subworld cannot be passed directly into, out of, or between subworlds. This includes Data objects, FunctionSpaces and LinearPDEs.

The work that you want the subworld to perform is described by a python function. The same function can be executed many times (possibly with different parameters) and each execution is handled by a separate instance. Jobs can be added to the splitworld using one of two commands:

```
sw.addJob(FunctionJob, JobName, par1=value1, par2=value2)
sw.addJobPerWorld(FunctionJob, JobName, par1=value1, par2=value2)
```

`FunctionJob` is the name of an `esys.escript` class that is used to handle some variables inside the subworld and `JobName` is the name of the python function describing the work to be done in the subworld. The first of these commands, `addJob`, adds a single job to execute on an available subworld. The second command, `addJobPerWorld` creates a job instance on every subworld[4]. One use of this would be to run a setup program on each subworld. The parameters `par1` and `par2` are illustrative only and can be customized.

Since task functions run by jobs must have the parameter list `(self, **kwargs)`, keyword arguments are the best way to get extra information into `JobName`. However, it is also possible to pass information downward from the top-level python script to a subworld and to pass information between jobs. To create a variable on all subworlds, use the command

```
sw.addVariable("variable name",variable_type)
```

where `"variable name"` is the name of the variable and `variable_type` is a string describing the type of variable to create. Supported variable types are `"local"`, which creates an integer variable, `"float"` which creates a floating point variable and `"Data"S`, which creates a data object. Each subworld has a local copy of the variable. Similarly, to remove an added variable, run the command

```
sw.removeVariable("variable",variable_type)
```

in the top level python script.

To access a variable inside a subworld task, first declare it and then import it using

```
self.declareImport("variable name")
var = self.importValue("variable name")
```

More information on this feature is included in Section 11.4 .

Once all the jobs have been added to the stack, they can be executed by running[5]

```
sw.runJobs()
```

If any of the jobs raise an exception (or if there is some other problem), then an exception will be raised in the top level python script to help you diagnose the fault[6].

[4]Note that this is not the same as calling `addJob` n times where n is the number of subworlds. It is better not to make assumptions about how SplitWorld distributes jobs to subworlds.

[5]If each instance of `JobName` is writing output to the console independently, it may be necessary to run `esys.escript` with the "-o" flag to ensure that each MPI process outputs data to a separate file.

[6]Some things preventing this:

- Only one of the exceptions will be reported (if multiple jobs raise, you will only see one message).
- The exception does not contain the payload and type of the original exception.
- We do not guarantee that all jobs have been attempted if an exception is raised.
- Variables may be reset and so values may be lost.

11.3 Example

This example script initialises a splitworld that consists of as many subworlds as there are running MPI processes, instructs the splitworld that each subworld is going to solve a PDE on a `esys.ripley` Rectangle, adds twenty jobs to the job stack and then runs everything:

```
from esys.escript import *
from esys.escript.linearPDEs import Poisson
from esys.ripley import Rectangle

# Initialise a Splitworld
sw=SplitWorld(getMPISizeWorld())

# Initialise a domain on each Subworld inside the Splitworld
sw.buildDomains(Rectangle, n0=100, n1=100)

# This python function describes the work that each subworld is going to do.
# In this case we solve a Poisson equation
def task(self, **kwargs):
    v=kwargs['v']
    dom=self.domain
    pde=Poisson(dom)
    x=dom.getX()
    gammaD=whereZero(x[0])+whereZero(x[1])
    pde.setValue(f=v, q=gammaD)
    soln=pde.getSolution()
    soln.dump('soln%d'%v)

# Add some jobs
for i in range(1,20):
    sw.addJob(FunctionJob, task, v=i)

# Run the jobs
sw.runJobs()
```

11.4 Classes and Functions

11.4.1 The Splitworld class

SplitWorld(n)
Returns a SplitWorld which contains n subworlds; will raise an exception if this is not possible.

All of these methods outlined in this subsection are only to be called by the top level python script. Do not attempt to use them inside a job:

addVariable(name, constructor, args)
Adds a variable to each of the subworlds built by the function `constructor` with arguments `args`.

clearVariable(name)
Clears the value of the named variable on all worlds. The variable no longer has a value but a new value can be exported for it.

copyVariable(source, dest)
Copies[7] the value into the named variable. This avoids the need to create jobs merely to importValue+exportValue into the new name.

[7]This is a deep copy for Data objects.

getDoubleVariable(name)
Extracts a floating point value of the named variable to the top level python script. If the named variable does not support this an exception will be raised.

getSubWorldCount()
Returns the number of subworlds.

getSubWorldID()
Returns the id of the local world.

getVarList()
Return a python list of pairs `[name, hasvalue]` (one for each variable). `hasvalue` is True if the variable currently has a value. Mainly useful for debugging.

buildDomains(constructor, args)
Indicates how subworlds should construct their domains. *Note that the splitworld code does not support multi-resolution ripley domains yet.*

addJob(FunctionJob, function, args)
Submit a job to run `function` with `args` to be executed in an available subworld.

addJobPerWorld(FunctionJob, function, args)
Submit a job to run `function` with `args` to be executed in each subworld. Individual jobs can use properties of `self` such as `swid` or `jobid` to distinguish between themselves.

removeVariable(name)
Removes a variable and its value from all subworlds.

runJobs( )
Runs all queued jobs in the splitworld.

11.4.2 FunctionJob methods and members

The following parameters may be accessed inside a FunctionJob.

jobid
Returns the id of the current job

swcount
Returns the number of subworlds in the SplitWorld.

swid
Returns the id of the subworld this job is running in.

importValue(name)
Retrieves the value for the named variable. This is a shallow copy so modifications made in the function may affect the variable (LocalOnly). Do not abuse this, use `exportValue` instead.

exportValue(name, value)
Contributes a new value for the named variable.

APPENDIX
A

Einstein Notation

Compact notation is used in equations such continuum mechanics and linear algebra; it is known as Einstein notation or the Einstein summation convention. It makes the conventional notation of equations involving tensors more compact by shortening and simplifying them.

There are two rules which make up the convention. Firstly, the rank of a tensor is represented by an index. For example, a is a scalar, b_i represents a vector, and c_{ij} represents a matrix. Secondly, if an expression contains repeated subscripted variables, they are assumed to be summed over all possible values, from 0 to n. For example, the expression

$$y = a_0 b_0 + a_1 b_1 + \ldots + a_n b_n \tag{A.1}$$

can be represented as

$$y = \sum_{i=0}^{n} a_i b_i \tag{A.2}$$

then in Einstein notation:

$$y = a_i b_i \tag{A.3}$$

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 ',' in the subscript indicates the partial derivative.
For a tensor:

$$\sigma_{ij} = \begin{bmatrix} \sigma_{00} & \sigma_{01} & \sigma_{02} \\ \sigma_{10} & \sigma_{11} & \sigma_{12} \\ \sigma_{20} & \sigma_{21} & \sigma_{22} \end{bmatrix} \tag{A.6}$$

The δ_{ij} is the Kronecker δ-symbol, which is a matrix with ones in its diagonal entries ($i = j$) and zeros in the remaining entries ($i \neq j$).

$$\delta_{ij} = \begin{cases} 1, & \text{if } i = j \\ 0, & \text{if } i \neq j \end{cases} \tag{A.7}$$

Index

Bibliography

[1] William L. Briggs, Van Emden Henson, and Steve F. McCormick. *A Multigrid Tutorial: Second Edition*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 2000.

[2] P. G. Ciarlet and J. L. Lions. *Handbook of Numerical Analysis*, volume 2. North Holland, Amsterdam, 1991.

[3] Scipy Community. *Numpy and Scipy Documentation*.

[4] Timothy A. Davis. Algorithm 832: UMFPACK V4.3—an unsymmetric-pattern multifrontal method. *ACM Transactions on Mathematical Software (TOMS)*, 30:196–199, 2004.

[5] Joel Fenwick and Lutz Gross. Lazy Evaluation of PDE Coefficients in the EScript System. In Jinjun Chen and Rajiv Ranjan, editors, *Parallel and Distributed Computing 2010 (AusPDC2010)*, volume 107 of *Conferences in Research and Practice in Information Technology*, pages 71–76, January 2010.

[6] Christophe Geuzaine and Jean-Francois Remacle. *Gmsh Reference Manual*, 1.12 edition, Aug 2003.

[7] V. Girault and P. A. Raviart. *Finite Element Methods for Navier-Stokes Equations- Theory and Algorithms*. Springer Verlag, Berlin, 1986.

[8] Paradigm GOCAD homepage. `http://www.pdgm.com/Products/GOCAD`.

[9] Michael A. Heroux, Roscoe A. Bartlett, Vicki E. Howle, Robert J. Hoekstra, Jonathan J. Hu, Tamara G. Kolda, Richard B. Lehoucq, Kevin R. Long, Roger P. Pawlowski, Eric T. Phipps, Andrew G. Salinger, Heidi K. Thornquist, Ray S. Tuminaro, James M. Willenbring, Alan Williams, and Kendall S. Stanley. An overview of the Trilinos project. *ACM Trans. Math. Softw.*, 31(3):397–423, 2005.

[10] John Hunter, Michael Droettboom, and Darren Dale. *matplotlib*, July 2009.

[11] Sandeep Koranne. Hierarchical data format 5: Hdf5. In *Handbook of Open Source Tools*, pages 191–200. Springer, 2011.

[12] *Mayavi: 3D scientific data visualization and plotting in Python*, 2015.

[13] INTEL's Math Kernel Library.

[14] MPI. http://www.mpi-forum.org.

[15] OpenMP. http://openmp.org.

[16] Y. Saad. *Iterative Methods for Sparse Linear Systems*. PWS Publishing Company, 20 Park Plaza, Boston, MA 02116, USA, 1996.

[17] Y. Shapira. *Matrix-Based Multigrid*. Springer, 2008.

[18] K. Stuben. A review of algebraic multigrid. *Journal of Computational and Applied Mathematics*, 128(1–2):281–309, 2001. Numerical Analysis 2000. Vol. VII: Partial Differential Equations.

[19] B. Suchomel and Y. Saad. ARMS: an algebraic recursive multilevel solver for general sparse linear systems. *Numerical Linear Algebra with Applications*, 9(5):1099–1506, 2002.

[20] Sympy homepage. http://sympy.org/en/index.html.

[21] The Scipy community. *Numpy and Scipy Documentation*.

[22] Trilinos homepage. https://trilinos.github.io/.

[23] Ray S. Tuminaro. Parallel Smoothed Aggregation Multigrid: Aggregation Strategies on Massively Parallel Machines. In *Proceedings of the 2000 ACM/IEEE Conference on Supercomputing*, SC '00, Washington, DC, USA, 2000. IEEE Computer Society.

[24] P. Vaněk, J. Mandel, and M. Brezina. Algebraic multigrid by smoothed aggregation for second and fourth order elliptic problems. *Computing*, 56(3):179–196, 1996.

[25] VisIt homepage. https://wci.llnl.gov/codes/visit/home.html.

[26] R. Weiss. *Parameter-Free Iterative Linear Solvers*. Mathematical Research, vol. 97. Akademie Verlag, Berlin, 1996.

[27] Gaussian blur. `http://en.wikipedia.org/wiki/Gaussian_blur`.

[28] Thomas Williams and Colin Kelley. gnuplot homepage. http://www.gnuplot.info/, March 2009.

[29] O. C. Zienkiewicz. *The Finite Element Method in Engineering Science*. McGraw-Hill, London, second edition, 1971.