The *escript* COOKBOOK

Release - 3.2.1

(r3613)

Antony Hallam, Lutz Gross, et al.

September 28, 2011

Earth Systems Science Computational Centre (ESSCC)

School of Earth Sciences

The University of Queensland

Brisbane, Australia

Email: esys@esscc.uq.edu.au

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

Abstract

escript is a *python* based environment that has been developed to solve complex mathematical models, particularly coupled, non-linear and time-dependent partial differential equations. The intention of this cookbook is to introduce new users to *escript* and provide a set of examples which demonstrate the major concepts and can be adapted to new problems. Although most of the examples in this cookbook are focused on the disciplines of geophysics and geology, they provide a solid introduction to *escript* and its capabilities.

Researchers and Developers

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

Cihan Altinay `esys.weipa` visualisation package, SCons build system rework.

Artak Amirbekyan Solvers, OSX work [prior to 3.2.1].

Joel Fenwick Lazy evaluation, maintenance of escript module, release wrangler.

Lin Gao Performance analysis.

Lutz Gross Patriarch, technical lead, solvers, large chunks of the original code.

Contents

CHAPTER

ONE

Introduction

1.1 Why *escript*?

escript is a scripting environment for mathematical modelling based on partial differential equations (PDEs). It provides a high-level of abstraction from the underlying numerical schemes and their implementations. By freeing the user from considerations like data constructs, meshing and parallelisation, the user can concentrate on the modelling aspects of the problem and still properly utilise the powerful mathematical capabilities of PDEs.

escript is built upon the interpreted programming language *python*[1], a scripting language with many intrinsic functions and capabilities. Additionally, there are also a large number of software packages for *python* which can be used in conjunction with *escript*. These packages include functions and data constructs for linear algebra, statistics, visualisation, image processing and data plotting among others. Furthermore, most *escript* scripts are scalable and able to run on single core desktop computers right through to multi-core supercomputers[2] with no modifications to the scripts.

There are many benefits for using a software platform like *escript* for projects that involve mathematical modelling. Building on top of an existing environment such as *escript* is in many cases much simpler and more cost effective than building an original implementation from the ground up. A modelling environment needs data structures and solution algorithms which take time to develop and test properly, *escript* has already covered these aspects and its implementation has been widely tested for bugs. Although existing environments may not provide the user with the fastest algorithms for their problems, it is generally the case, that the overall time needed to identify, implement and test the optimal algorithm will exceed the time needed to implement and solve the prob-

[1] see www.python.org

[2] *escript* supports distributed memory architectures with multi-core processors through MPI and threading. See the *escript* user guide at https://launchpad.net/escript-finley/+download for details.

lem with pre-developed and tested software. This is particularly true if a simulation does not need to be executed repetitively, or has relatively short lifetime. A model for a publication or thesis would be one such instance.

When it comes to solving partial differential equations, *escript* is ideal as it is especially designed for this task. Other implementations are merely an add-on to a linear algebra focused system (*e.g.* MATLAB). The *escript* approach gives the user a cleaner environment to work with and provides better efficiency when dealing with PDE coefficients. Data structures in *escript* allow the user to abstract away details such as data types of these coefficients. For example, if a model has been tested with a constant PDE coefficient then the unchanged script can be run with variable coefficients from a database or as a function of a dependent variable. This capability of *escript* is possible because *escript* uses the language of PDEs (as opposed to linear algebra) to describe a model. As it turns out, the *escript* approach can be applied efficiently in very large software projects as it leads to a clearer structure for the code, by separating modelling issues from low-level numerical and computational performance issues. At the same time, this arrangement also allows for the implementation of complex model coupling on a higher-level.

The use of *python* as the platform for *escript* makes the development of models simple from a user perspective, as *python* is intuitive and easy to learn. This simplicity does not hamper experienced users either as *python* also provides access to a very large number of tools. This makes it an attractive environment to work in. Best of all, *escript* is released under an open software license and is freely available for download.

1.2 How to use this Cookbook

This manual is written with the intention of giving new users a practical introduction to *escript*. It demonstrates how to solve a variety of problems from simple to advanced. We recommend that new users work through the *first few sets of examples* in Chapters 2, 3 and 4). These chapters contain the necessary basic knowledge, and explain some of the common aspects and modules of *escript*. The simple examples demonstrate how to create, solve and visualise PDE based models. Future chapters (as they are added to this tutorial) will cover more advanced topics with more complex models and methods. Further examples are available in the *escript* user guide.

All examples covered in this cookbook have been scripted and are ready to run. They are available from the *doc/examples/cookbook/* folder in the *escript* directory. These scripts provide a basis for users to develop their own models while at the same time demonstrating the steps required to completely solve and visualise the PDE problems.

1.3 Quickstart

For information on how to install and run *escript* please look at the *installation* and *user* guides which are available for download from launchpad at https://launchpad.net/escript-finley/+download.

1.4 Escript and Python Basics

The *python* scripting language is a powerful and easy to learn environment with a wide variety of applications. *escript* has been developed as a packaged module for *python* specifically to solve complex partial differential equations. As a result, all the conventions and programming syntax associated with *python* are coherent with *escript*. If you are unfamiliar with *python*, there are a large number of simple to advanced guides and tutorials available online. These texts should provide an introduction that is comprehensive enough to use *escript*. A handful of *python* tutorials are listed below.

- http://hetland.org/writing/instant-python.html is a very crisp introduction. It covers everything you need to get started with *escript*.
- A nice and easy to follow introduction: http://www.sthurlow.com/python/
- Another crisp tutorial: http://www.zetcode.com/tutorials/pythontutorial/.
- A very comprehensive tutorial from the *python* authors: http://www.python.org/doc/2.5.2/tut/tut.html. It covers much more than what you will ever need for *escript*.
- Another comprehensive tutorial: http://www.tutorialspoint.com/python/index.htm

1.4.1 The `esys` Modules

escript is part of the *esys* package. Apart from the particle simulation library `ESyS-Particle`[3] which is not covered in this tutorial *esys* also includes the following modules

1. `esys.escript` is the PDE solving module.
2. `esys.finley` is the discretisation tool and finite element package.
3. `esys.pycad` is a package for creating irregular shaped domains.

Further explanations of each of these are available in the *escript* user guide or in the API documentation[4]. *escript* is also dependent on a few other open-source packages which are not maintained by the *escript* development team.

[3] see https://launchpad.net/esys-particle
[4] Available from https://launchpad.net/escript-finley/+download

These are `numpy` (an array and matrix handling package), `matplotlib`[5] (a simple plotting tool) and `gmsh`[6] (which is required by `esys.pycad`). These packages (**except** for `gmsh`) are included with the support bundles.

[5] `numpy` and `matplotlib` are part of the SciPy package, see http://www.scipy.org/
[6] See http://www.geuz.org/gmsh/

CHAPTER

TWO

Getting Started with Heat Diffusion

We start by examining a simple one dimensional heat diffusion equation. This problem provides a good starting example to build our knowledge of *escript* and demonstrate how to solve simple partial differential equations (PDEs)[1]

2.1 Example 1: One Dimensional Heat Diffusion in Granite

The first model consists of two blocks of isotropic material, for instance granite, sitting next to each other (Figure 2.1). Initial temperature in *Block 1* is `T1` and in *Block 2* is `T2`. We assume that the system is insulated. What would happen to the temperature distribution in each block over time? Intuition tells us that heat will be transported from the hotter block to the cooler one until both blocks have the same temperature.

2.1.1 1D Heat Diffusion Equation

We can model the heat distribution of this problem over time using the one dimensional heat diffusion equation[2]; which is defined as:

$$\rho c_p \frac{\partial T}{\partial t} - \kappa \frac{\partial^2 T}{\partial x^2} = q_H \tag{2.1}$$

where ρ is the material density, c_p is the specific heat and κ is the thermal conductivity[3]. Here we assume that these material parameters are **constant**. The heat source is defined by the right hand side of Equation (2.1) as q_H;

[1] Wikipedia provides an excellent and comprehensive introduction to *Partial Differential Equations* http://en.wikipedia.org/wiki/Partial_differential_equation, however their relevance to *escript* and implementation should become much clearer as we develop our understanding further into the cookbook.

[2] A detailed discussion on how the heat diffusion equation is derived can be found at http://online.redwoods.edu/instruct/darnold/DEProj/sp02/AbeRichards/paper.pdf

[3] A list of some common thermal conductivities is available from Wikipedia http://en.wikipedia.org/wiki/List_of_thermal_conductivities

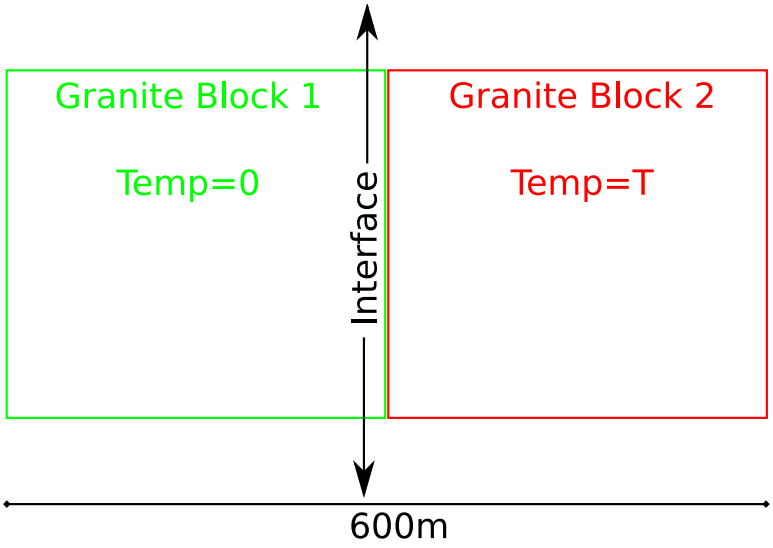


Figure 2.1: Example 1: Temperature differential along a single interface between two granite blocks.

this can take the form of a constant or a function of time and space. For example $q_H = q_0 e^{-\gamma t}$ where we have the output of our heat source decaying with time. There are also two partial derivatives in Equation (2.1); $\frac{\partial T}{\partial t}$ describes the change in temperature with time while $\frac{\partial^2 T}{\partial x^2}$ is the spatial change of temperature. As there is only a single spatial dimension to our problem, our temperature solution T is only dependent on the time t and our signed distance from the block-block interface x.

2.1.2 PDEs and the General Form

It is possible to solve PDE Equation (2.1) analytically and obtain an exact solution to our problem. However, it is not always practical to solve the problem this way. Alternatively, computers can be used to find the solution. To do this, a numerical approach is required to discretise the PDE Equation (2.1) across time and space, this reduces the problem to a finite number of equations for a finite number of spatial points and time steps. These parameters together define the model. While discretisation introduces approximations and a degree of error, a sufficiently sampled model is generally accurate enough to satisfy the accuracy requirements for the final solution.

Firstly, we discretise the PDE Equation (2.1) in time. This leaves us with a steady linear PDE which involves spatial derivatives only and needs to be solved in each time step to progress in time. *escript* can help us here.

For time discretisation we use the Backward Euler approximation scheme[4]. It is based on the approximation

$$\frac{\partial T(t)}{\partial t} \approx \frac{T(t) - T(t-h)}{h} \tag{2.2}$$

[4]see http://en.wikipedia.org/wiki/Euler_method

for $\frac{\partial T}{\partial t}$ at time t where h is the time step size. This can also be written as;

$$\frac{\partial T}{\partial t}(t^{(n)}) \approx \frac{T^{(n)} - T^{(n-1)}}{h} \tag{2.3}$$

where the upper index n denotes the n$^{\text{th}}$ time step. So one has

$$\begin{aligned} t^{(n)} &= t^{(n-1)} + h \\ T^{(n)} &= T(t^{(n-1)}) \end{aligned} \tag{2.4}$$

Substituting Equation (2.3) into Equation (2.1) we get;

$$\frac{\rho c_p}{h}(T^{(n)} - T^{(n-1)}) - \kappa \frac{\partial^2 T^{(n)}}{\partial x^2} = q_H \tag{2.5}$$

Notice that we evaluate the spatial derivative term at the current time $t^{(n)}$ - therefore the name **backward Euler** scheme. Alternatively, one can evaluate the spatial derivative term at the previous time $t^{(n-1)}$. This approach is called the **forward Euler** scheme. This scheme can provide some computational advantages, which are not discussed here. However, the **forward Euler** scheme has a major disadvantage. Namely, depending on the material parameters as well as the domain discretization of the spatial derivative term, the time step size h needs to be chosen sufficiently small to achieve a stable temperature when progressing in time. Stability is achieved if the temperature does not grow beyond its initial bounds and becomes non-physical. The backward Euler scheme, which we use here, is unconditionally stable meaning that under the assumption of a physically correct problem set-up the temperature approximation remains physical for all time steps. The user needs to keep in mind that the discretisation error introduced by Equation (2.2) is sufficiently small, thus a good approximation of the true temperature is computed. It is therefore very important that any results are viewed with caution. For example, one may compare the results for different time and spatial step sizes.

To get the temperature $T^{(n)}$ at time $t^{(n)}$ we need to solve the linear differential equation Equation (2.5) which only includes spatial derivatives. To solve this problem we want to use *escript*.

In *escript* any given PDE can be described by the general form. For the purpose of this introduction we illustrate a simpler version of the general form for full linear PDEs which is available in the *escript* user's guide. A simplified form that suits our heat diffusion problem[5] is described by;

$$-\nabla \cdot (A \cdot \nabla u) + Du = f \tag{2.6}$$

where A, D and f are known values and u is the unknown solution. The symbol ∇ which is called the *Nabla operator* or *del operator* represents the spatial derivative of its subject - in this case u. Lets assume for a moment

[5]The form in the *escript* users guide which uses the Einstein convention is written as $-(A_{jl}u_{,l})_{,j} + Du = Y$

that we deal with a one-dimensional problem then ;

$$\nabla = \frac{\partial}{\partial x} \tag{2.7}$$

and we can write Equation (2.6) as;

$$-A\frac{\partial^2 u}{\partial x^2} + Du = f \tag{2.8}$$

if A is constant. To match this simplified general form to our problem Equation (2.5) we rearrange Equation (2.5);

$$\frac{\rho c_p}{h}T^{(n)} - \kappa\frac{\partial^2 T^{(n)}}{\partial x^2} = q_H + \frac{\rho c_p}{h}T^{(n-1)} \tag{2.9}$$

The PDE is now in a form that satisfies Equation (2.6) which is required for *escript* to solve our PDE. This can be done by generating a solution for successive increments in the time nodes $t^{(n)}$ where $t^{(0)} = 0$ and $t^{(n)} = t^{(n-1)} + h$ where $h > 0$ is the step size and assumed to be constant. In the following the upper index (n) refers to a value at time $t^{(n)}$. Finally, by comparing Equation (2.9) with Equation (2.8) one can see that;

$$u = T^{(n)}; A = \kappa; D = \frac{\rho c_p}{h}; f = q_H + \frac{\rho c_p}{h}T^{(n-1)} \tag{2.10}$$

2.1.3 Boundary Conditions

With the PDE sufficiently modified, consideration must now be given to the boundary conditions of our model. Typically there are two main types of boundary conditions known as **Neumann** and **Dirichlet** boundary conditions[6], respectively. A **Dirichlet boundary condition** is conceptually simpler and is used to prescribe a known value to the unknown solution (in our example the temperature) on parts of the boundary or on the entire boundary of the region of interest. We discuss the Dirichlet boundary condition in our second example presented in Section 2.2.

However, for this example we have made the model assumption that the system is insulated, so we need to add an appropriate boundary condition to prevent any loss or inflow of energy at the boundary of our domain. Mathematically this is expressed by prescribing the heat flux $\kappa\frac{\partial T}{\partial x}$ to zero. In our simplified one dimensional model this is expressed in the form;

$$\kappa\frac{\partial T}{\partial x} = 0 \tag{2.11}$$

or in a more general case as

$$\kappa\nabla T \cdot n = 0 \tag{2.12}$$

[6]More information on Boundary Conditions is available at Wikipedia http://en.wikipedia.org/wiki/Boundary_conditions

where n is the outer normal field at the surface of the domain. The $\cdot$ (dot) refers to the dot product of the vectors ∇T and n. In fact, the term $\nabla T \cdot n$ is the normal derivative of the temperature T. Other notations used here are[7];

$$\nabla T \cdot n = \frac{\partial T}{\partial n} \ . \tag{2.13}$$

A condition of the type Equation (2.12) defines a **Neumann boundary condition** for the PDE.

The PDE Equation (2.9) and the Neumann boundary condition 2.9 (potentially together with the Dirichlet boundary conditions) define a **boundary value problem**. It is the nature of a boundary value problem to allow making statements about the solution in the interior of the domain from information known on the boundary only. In most cases we use the term partial differential equation but in fact it is a boundary value problem. It is important to keep in mind that boundary conditions need to be complete and consistent in the sense that at any point on the boundary either a Dirichlet or a Neumann boundary condition must be set.

Conveniently, *escript* makes a default assumption on the boundary conditions which the user may modify where appropriate. For a problem of the form in Equation (2.6) the default condition[8] is;

$$-n \cdot A \cdot \nabla u = 0 \tag{2.14}$$

which is used everywhere on the boundary. Again n denotes the outer normal field. Notice that the coefficient A is the same as in the *escript* PDE 2.6. With the settings for the coefficients we have already identified in Equation (2.10) this condition translates into

$$\kappa \frac{\partial T}{\partial x} = 0 \tag{2.15}$$

for the boundary of the domain. This is identical to the Neumann boundary condition we want to set. *escript* will take care of this condition for us. We discuss the Dirichlet boundary condition later.

2.1.4 Outline of the Implementation

To solve the heat diffusion equation (Equation (2.1)) we write a simple *python* script. At this point we assume that you have some basic understanding of the *python* programming language. If not, there are some pointers and links available in Section 1.4. The script (discussed in Section 2.1.7) has four major steps. Firstly, we need to define the domain where we want to calculate the temperature. For our problem this is the joint blocks of granite which has a rectangular shape. Secondly, we need to define the PDE to solve in each time step to get the updated temperature. Thirdly, we need to define the coefficients of the PDE and finally we need to solve the PDE. The last two steps need to be repeated until the final time marker has been reached. The work flow is described in Figure (2.2).

[7] The *escript* notation for the normal derivative is $T_{,i} n_i$.
[8] In the *escript* user guide which uses the Einstein convention this is written as $n_j A_{jl} u_{,l} = 0$.

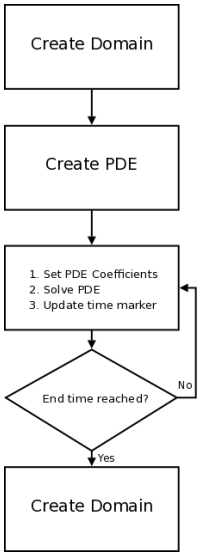


Figure 2.2: Workflow for developing an *escript* model and solution

In the terminology of *python*, the domain and PDE are represented by **objects**. The nice feature of an object is that it is defined by its usage and features rather than its actual representation. So we will create a domain object to describe the geometry of the two granite blocks. Then we define PDEs and spatially distributed values such as the temperature on this domain. Similarly, to define a PDE object we use the fact that one needs only to define the coefficients of the PDE and solve the PDE. The PDE object has advanced features, but these are not required in simple cases.

2.1.5 The Domain Constructor in *escript*

Whilst it is not strictly relevant or necessary, a better understanding of how values are spatially distributed (*e.g.* Temperature) and how PDE coefficients are interpreted in *escript* can be helpful.

There are various ways to construct domain objects. The simplest form is a rectangular shaped region with a length and height. There is a ready to use function for this named `rectangle()`. Besides the spatial dimensions this function requires to specify the number of elements or cells to be used along the length and height, see Figure (2.3). Any spatially distributed value and the PDE is represented in discrete form using this element representation[9]. Therefore we will have access to an approximation of the true PDE solution only. The quality of the approximation depends - besides other factors - mainly on the number of elements being used. In fact, the approximation becomes better when more elements are used. However, computational cost grows with the number of elements being used. It is therefore important that you find the right balance between the demand in accuracy and acceptable resource usage.

In general, one can think about a domain object as a composition of nodes and elements. As shown in Fig-

[9]We use the finite element method (FEM), see http://en.wikipedia.org/wiki/Finite_element_method for details.

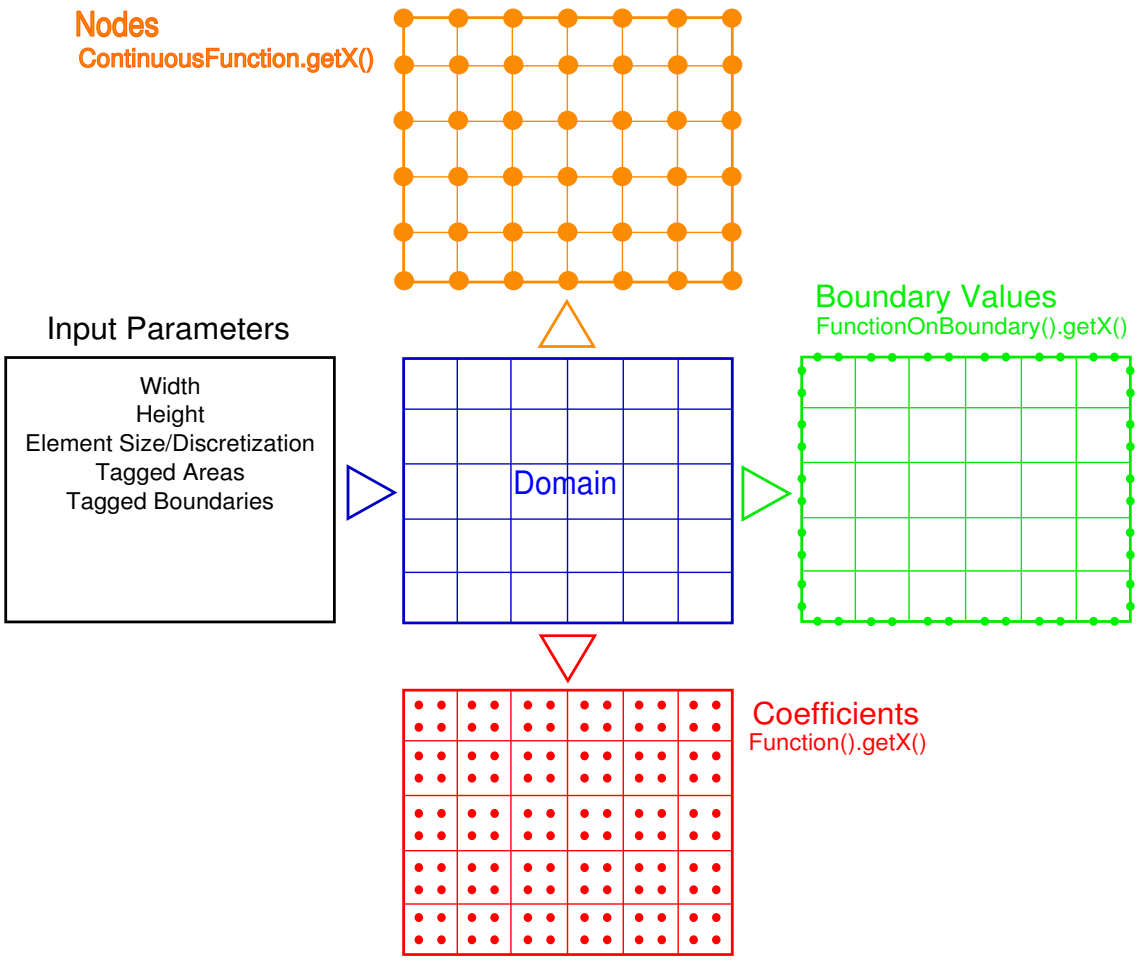


Figure 2.3: *escript* domain construction overview

ure (2.3), an element is defined by the nodes that are used to describe its vertices. To represent spatially distributed values the user can use the values at the nodes, at the elements in the interior of the domain or at the elements located on the surface of the domain. The different approach used to represent values is called **function space** and is attached to all objects in *escript* representing a spatially distributed value such as the solution of a PDE. The three function spaces we use at the moment are;

1. the nodes, called by `ContinuousFunction(domain)` ;
2. the elements/cells, called by `Function(domain)` ; and
3. the boundary, called by `FunctionOnBoundary(domain)`.

A function space object such as `ContinuousFunction(domain)` has the method `getX` attached to it. This method returns the location of the so-called **sample points** used to represent values of the particular function space. So the call `ContinuousFunction(domain).getX()` will return the coordinates of the nodes used to describe the domain while `Function(domain).getX()` returns the coordinates of numerical integration points within elements, see Figure (2.3).

This distinction between different representations of spatially distributed values is important in order to be able to vary the degrees of smoothness in a PDE problem. The coefficients of a PDE do not need to be continuous, thus this qualifies as a `Function()` type. On the other hand a temperature distribution must be continuous and needs to be represented with a `ContinuousFunction()` function space. An influx may only be defined at the boundary and is therefore a `FunctionOnBoundary()` object. *escript* allows certain transformations of the function spaces. A `ContinuousFunction()` can be transformed into a `FunctionOnBoundary()` or `Function()`. On the other hand there is not enough information in a `FunctionOnBoundary()` to transform it to a `ContinuousFunction()`. These transformations, which are called **interpolation** are invoked automatically by *escript* if needed.

Later in this introduction we discuss how to define specific areas of geometry with different materials which are represented by different material coefficients such as the thermal conductivities κ. A very powerful technique to define these types of PDE coefficients is tagging. Blocks of materials and boundaries can be named and values can be defined on subregions based on their names. This is a method for simplifying PDE coefficient and flux definitions. It makes scripting much easier and we will discuss this technique in Section 4.1.

2.1.6 A Clarification for the 1D Case

It is necessary for clarification that we revisit our general PDE from equation (2.6) for a two dimensional domain. *escript* is inherently designed to solve problems that are multi-dimensional and so Equation (2.6) needs to be read as a higher dimensional problem. In the case of two spatial dimensions the *Nabla operator* has in fact two

components $\nabla = (\frac{\partial}{\partial x}, \frac{\partial}{\partial y})$. Assuming the coefficient A is constant, the Equation (2.6) takes the following form;

$$-A_{00}\frac{\partial^2 u}{\partial x^2} - A_{01}\frac{\partial^2 u}{\partial x \partial y} - A_{10}\frac{\partial^2 u}{\partial y \partial x} - A_{11}\frac{\partial^2 u}{\partial y^2} + Du = f \tag{2.16}$$

Notice that for the higher dimensional case A becomes a matrix. It is also important to notice that the usage of the Nabla operator creates a compact formulation which is also independent from the spatial dimension. To make the general PDE Equation (2.16) one dimensional as shown in Equation (2.8) we need to set

$$A_{00} = A; A_{01} = A_{10} = A_{11} = 0 \tag{2.17}$$

2.1.7 Developing a PDE Solution Script

The scripts referenced in this section are; example01a.py

We write a simple *python* script which uses the `esys.escript`, `esys.finley` and `matplotlib` modules. By developing a script for *escript*, the heat diffusion equation can be solved at successive time steps for a predefined period using our general form Equation (2.9). Firstly it is necessary to import all the libraries[10] that we will require.

```
from esys.escript import *
# This defines the LinearPDE module as LinearPDE
from esys.escript.linearPDEs import LinearPDE
# This imports the rectangle domain function from finley.
from esys.finley import Rectangle
# A useful unit handling package which will make sure all our units
# match up in the equations under SI.
from esys.escript.unitsSI import *
```

It is generally a good idea to import all of the `esys.escript` library, although if the functions and classes required are known they can be specified individually. The function `LinearPDE` has been imported explicitly for ease of use later in the script. `Rectangle` is going to be our type of domain. The module `unitsSI` provides support for SI unit definitions with our variables.

Once our library dependencies have been established, defining the problem specific variables is the next step. In general the number of variables needed will vary between problems. These variables belong to two categories. They are either directly related to the PDE and can be used as inputs into the *escript* solver, or they are script variables used to control internal functions and iterations in our problem. For this PDE there are a number of

[10]The libraries contain predefined scripts that are required to solve certain problems, these can be simple like sine and cosine functions or more complicated like those from our *escript* library.

constants which need values. Firstly, the domain upon which we wish to solve our problem needs to be defined. There are different types of domains in `esys.escript` which we demonstrate in later tutorials but for our granite blocks, we simply use a rectangular domain.

Using a rectangular domain simplifies our granite blocks (which would in reality be a *3D* object) into a single dimension. The granite blocks will have a lengthways cross section that looks like a rectangle. As a result we do not need to model the volume of the block due to symmetry. There are four arguments we must consider when we decide to create a rectangular domain, the domain *length*, *width* and *step size* in each direction. When defining the size of our problem it will help us determine appropriate values for our model arguments. If we make our dimensions large but our step sizes very small we increase the accuracy of our solution. Unfortunately we also increase the number of calculations that must be solved per time step. This means more computational time is required to produce a solution. In this *1D* problem, the bar is defined as being 1 metre long. An appropriate step size `ndx` would be 1 to 10% of the length. Our `ndy` needs only be 1, this is because our problem stipulates no partial derivatives in the y direction. Thus the temperature does not vary with y. Hence, the model parameters can be defined as follows; note we have used the `unitsSI` convention to make sure all our input units are converted to SI.

```
mx = 500.*m #meters - model length
my = 100.*m #meters - model width
ndx = 50 # mesh steps in x direction
ndy = 1 # mesh steps in y direction
boundloc = mx/2 # location of boundary between the two blocks
```

The material constants and the temperature variables must also be defined. For the granite in the model they are defined as:

```
#PDE related
rho = 2750. *kg/m**3 #kg/m^{3} density of iron
cp = 790.*J/(kg*K) # J/Kg.K thermal capacity
rhocp = rho*cp
kappa = 2.2*W/m/K   # watts/m.Kthermal conductivity
qH=0 * J/(sec*m**3) # J/(sec.m^{3}) no heat source
T1=20 * Celsius # initial temperature at Block 1
T2=2273. * Celsius # base temperature at Block 2
```

Finally, to control our script we will have to specify our timing controls and where we would like to save the output from the solver. This is simple enough:

```
t=0 * day  # our start time, usually zero
tend=50 * yr # - time to end simulation
```

```
outputs = 200 # number of time steps required.
h=(tend-t)/outputs #size of time step
#user warning statement
print "Expected Number of time outputs is: ", (tend-t)/h
i=0 #loop counter
```

Now that we know our inputs we will build a domain using the `Rectangle()` function from *finley*. The four arguments allow us to define our domain `model` as:

```
#generate domain using rectangle
blocks = Rectangle(l0=mx,l1=my,n0=ndx, n1=ndy)
```

`blocks` now describes a domain in the manner of Section 2.1.5.

With a domain and all the required variables established, it is now possible to set up our PDE so that it can be solved by *escript*. The first step is to define the type of PDE that we are trying to solve in each time step. In this example it is a single linear PDE[11]. We also need to state the values of our general form variables.

```
mypde=LinearPDE(blocks)
A=zeros((2,2)))
A[0,0]=kappa
mypde.setValue(A=A, D=rhocp/h)
```

In many cases it may be possible to decrease the computational time of the solver if the PDE is symmetric. Symmetry of a PDE is defined by;

$$A_{jl} = A_{lj} \tag{2.18}$$

Symmetry is only dependent on the A coefficient in the general form and the other coefficients D as well as the right hand side Y. From the above definition we can see that our PDE is symmetric. The `LinearPDE` class provides the method `checkSymmetry` to check if the given PDE is symmetric. As our PDE is symmetrical we enable symmetry via;

```
myPDE.setSymmetryOn()
```

Next we need to establish the initial temperature distribution `T`. We need to assign the value `T1` to all sample points left to the contact interface at $x_0 = \frac{mx}{2}$ and the value `T2` right to the contact interface. *escript* provides the `whereNegative` function to construct this. More specifically, `whereNegative` returns the value 1 at those sample points where the argument has a negative value. Otherwise zero is returned. If `x` are the x_0 coordinates of the sample points used to represent the temperature distribution then `x[0]-boundloc` gives us a negative value for all sample points left to the interface and non-negative value to the right of the interface. So with;

[11] in contrast to a system of PDEs which we discuss later.

```
# ... set initial temperature ....
T= T1*whereNegative(x[0]-boundloc)+T2*(1-whereNegative(x[0]-boundloc))
```

we get the desired temperature distribution. To get the actual sample points `x` we use the `getX()` method of the function space `Solution(blocks)` which is used to represent the solution of a PDE;

```
x=Solution(blocks).getX()
```

As `x` are the sample points for the function space `Solution(blocks)` the initial temperature `T` is using these sample points for representation. Although *escript* is trying to be forgiving with the choice of sample points and to convert where necessary the adjustment of the function space is not always possible. So it is advisable to make a careful choice on the function space used.

Finally we initialise an iteration loop to solve our PDE for all the time steps we specified in the variable section. As the right hand side of the general form is dependent on the previous values for temperature `T` across the bar this must be updated in the loop. Our output at each time step is `T` the heat distribution and `totT` the total heat in the system.

```
while t < tend:
        i+=1 #increment the counter
        t+=h #increment the current time
        mypde.setValue(Y=qH+rhocp/h*T) # set variable PDE coefficients
        T=mypde.getSolution() #get the PDE solution
        totE = integrate(rhocp*T) #get the total heat (energy) in the system
```

The last statement in this script calculates the total energy in the system as the volume integral of $\rho c_p T$ over the block. As the blocks are insulated no energy should be lost or added. The total energy should stay constant for the example discussed here.

2.1.8 Running the Script

The script presented so far is available under `example01a.py`. You can edit this file with your favourite text editor. On most operating systems[12] you can use the **`run-escript`** command to launch *escript* scripts. For the example script use;

```
run-escript example01a.py
```

The program will print a progress report. Alternatively, you can use the python interpreter directly;

```
python example01a.py
```

if the system is configured correctly (please talk to your system administrator).

[12]The `run-escript` launcher is not supported under *MS Windows* yet.

2.1.9 Plotting the Total Energy

The scripts referenced in this section are; example01b.py

escript does not include its own plotting capabilities. However, it is possible to use a variety of free *python* packages for visualisation. Two types will be demonstrated in this cookbook; `matplotlib`[13] and `VTK`[14]. The `matplotlib` package is a component of SciPy[15] and is good for basic graphs and plots. For more complex visualisation tasks, in particular two and three dimensional problems we recommend the use of more advanced tools. For instance, *Mayavi2*[16] which is based upon the `VTK` toolkit. The usage of `VTK` based visualisation is discussed in Chapter 3.1 which focuses on a two dimensional PDE.

For our simple granite block problem, we have two plotting tasks. Firstly, we are interested in showing the behaviour of the total energy over time and secondly, how the temperature distribution within the block is developing over time. Let us start with the first task.

The idea is to create a record of the time marks and the corresponding total energies observed. *python* provides the concept of lists for this. Before the time loop is opened we create empty lists for the time marks `t_list` and the total energies `E_list`. After the new temperature has been calculated by solving the PDE we append the new time marker and the total energy value for that time to the corresponding list using the `append` method. With these modifications our script looks as follows:

```
t_list=[]
E_list=[]
# ... start iteration:
while t<tend:
      t+=h
      mypde.setValue(Y=qH+rhocp/h*T) # set variable PDE coefficients
      T=mypde.getSolution() #get the PDE solution
      totE=integrate(rhocp*T)
      t_list.append(t)    # add current time mark to record
      E_list.append(totE) # add current total energy to record
```

To plot t over $totE$ we use `matplotlib` a module contained within `pylab` which needs to be loaded before use;

```
import pylab as pl # plotting package.
```

[13] http://matplotlib.sourceforge.net/
[14] http://www.vtk.org/
[15] http://www.scipy.org
[16] http://code.enthought.com/projects/mayavi/

Here we are not using `from pylab import *` in order to avoid name clashes for function names within *escript*.

The following statements are added to the script after the time loop has been completed;

```
pl.plot(t_list,E_list)
pl.title("Total Energy")
pl.axis([0,max(t_list),0,max(E_list)*1.1])
pl.savefig("totE.png")
```

The first statement hands over the time marks and corresponding total energies to the plotter. The second statement sets the title for the plot. The third statement sets the axis ranges. In most cases these are set appropriately by the plotter.

The last statement generates the plot and writes the result into the file `totE.png` which can be displayed by (almost) any image viewer. As expected the total energy is constant over time, see Figure (2.4).

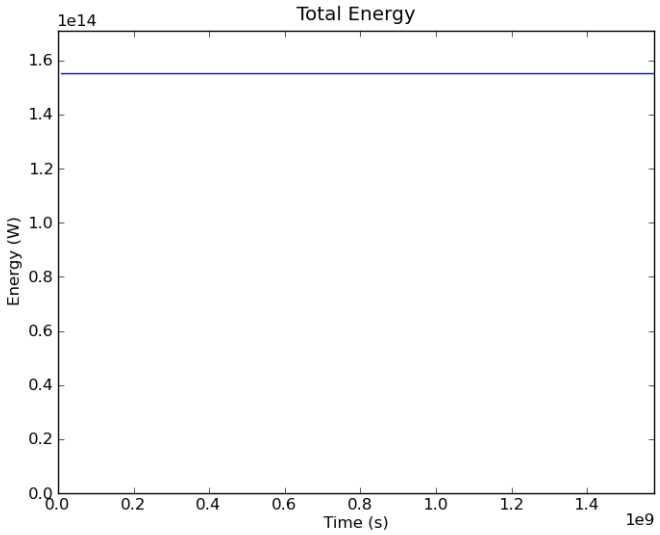


Figure 2.4: Example 1b: Total Energy in the Blocks over Time (in seconds)

2.1.10 Plotting the Temperature Distribution

The scripts referenced in this section are; example01c.py

For plotting the spatial distribution of the temperature we need to modify the strategy we have used for the total energy. Instead of producing a final plot at the end we will generate a picture at each time step which can be browsed as a slide show or composed into a movie. The first problem we encounter is that if we produce an image at each time step we need to make sure that the images previously generated are not overwritten.

To develop an incrementing file name we can use the following convention. It is convenient to put all image files showing the same variable - in our case the temperature distribution - into a separate directory. As part of the `os` module[17] *python* provides the `os.path.join` command to build file and directory names in a platform independent way. Assuming that `save_path` is the name of the directory we want to put the results in the command is;

```
import os
os.path.join(save_path, "tempT%03d.png"%i )
```

where `i` is the time step counter. There are two arguments to the `join` command. The `save_path` variable is a predefined string pointing to the directory we want to save our data, for example a single sub-folder called `data` would be defined by;

```
save_path = "data"
```

while a sub-folder of `data` called `example01` would be defined by;

```
save_path = os.path.join("data","example01")
```

The second argument of `join` contains a string which is the file name or subdirectory name. We can use the operator `%` to use the value of `i` as part of our filename. The sub-string `%03d` indicates that we want to substitute a value into the name;

- `0` means that small numbers should have leading zeroes;
- `3` means that numbers should be written using at least 3 digits; and
- `d` means that the value to substitute will be a decimal integer.

To actually substitute the value of `i` into the name write `%i` after the string. When done correctly, the output files from this command will be placed in the directory defined by `save_path` as;

[17] The `os` module provides a powerful interface to interact with the operating system, see http://docs.python.org/library/os.html.

```
blockspyplot001.png
blockspyplot002.png
blockspyplot003.png
...
```

and so on.

A sub-folder check/constructor is available in *escript*. The command;

```
mkDir(save_path)
```

will check for the existence of `save_path` and if missing, create the required directories.

We start by modifying our solution script. Prior to the `while` loop we need to extract our finite solution points to a data object that is compatible with `matplotlib`. First we create the node coordinates of the sample points used to represent the temperature as a *python* list of tuples or a `numpy` array as requested by the plotting function. We need to convert the array `x` previously set as `Solution(blocks).getX()` into a *python* list and then to a `numpy` array. The x_0 component is then extracted via an array slice to the variable `plx`;

```
import numpy as np # array package.
#convert solution points for plotting
plx = x.toListOfTuples()
plx = np.array(plx) # convert to tuple to numpy array
plx = plx[:,0] # extract x locations
```

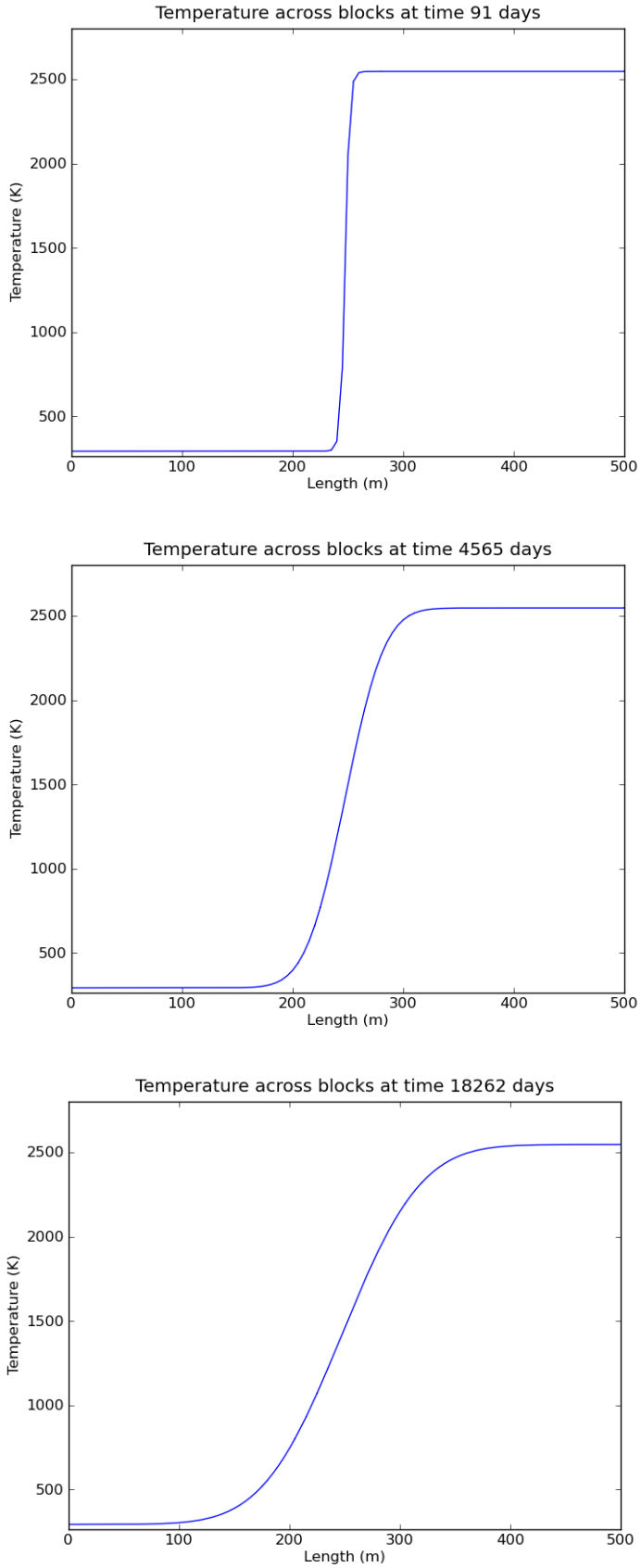


Figure 2.5: Example 1c: Temperature (T) distribution in the blocks at time steps 1, 50 and 200

We use the same techniques provided by `matplotlib` as we have used to plot the total energy over time. For each time step we generate a plot of the temperature distribution and save each to a file. The following is appended to the end of the `while` loop and creates one figure of the temperature distribution. We start by converting the solution to a tuple and then plotting this against our *x coordinates* `plx` we have generated before. We add a title to the diagram before it is rendered into a file. Finally, the figure is saved to a `*.png` file and cleared for the following iteration.

```
# ... start iteration:
while t<tend:
      i+=1
      t+=h
      mypde.setValue(Y=qH+rhocp/h*T)
      T=mypde.getSolution()
      totE=integrate(rhocp*T)
      print "time step %s at t=%e days completed. total energy = %e."%(i,t/day,totE)
      t_list.append(t)
      E_list.append(totE)

      #establish figure 1 for temperature vs x plots
      tempT = T.toListOfTuples()
      pl.figure(1) #current figure
      pl.plot(plx,tempT) #plot solution
      # add title
      pl.axis([0,mx,T1*.9,T2*1.1])
      pl.title("Temperature across blocks at time %d days"%(t/day))
      #save figure to file
      pl.savefig(os.path.join(save_path,"tempT", "blockspyplot%03d.png"%i))
      pl.clf() #clear figure
```

Some results are shown in Figure (2.5).

2.1.11 Making a Video

Our saved plots from the previous section can be cast into a video using the following command appended to the end of the script. The `mencoder`command is not available on every platform, so some users need to use an alternative video encoder.

```
# compile the *.png files to create a *.avi video that shows T change
# with time. This operation uses Linux mencoder. For other operating
```

```
# systems it is possible to use your favourite video compiler to
# convert image files to videos.

os.system("mencoder mf://"+save_path+"/tempT"+"/*.png -mf type=png:\
           w=800:h=600:fps=25 -ovc lavc -lavcopts vcodec=mpeg4 -oac copy -o \
           example01tempT.avi")
```

2.2 Example 2: One Dimensional Heat Diffusion in an Iron Rod

The scripts referenced in this section are; example02.py

Our second example is of a cold iron bar at a constant temperature of $T_{ref} = 20°C$, see Figure (2.6). The bar is perfectly insulated on all sides with a heating element at one end keeping the temperature at a constant level $T_0 = 100°C$. As heat is applied energy will disperse along the bar via conduction. With time the bar will reach a constant temperature equivalent to that of the heat source.

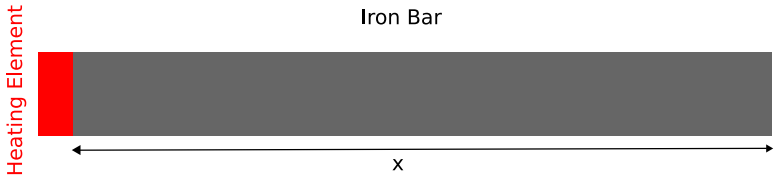


Figure 2.6: Example 2: One dimensional model of an Iron bar

This problem is very similar to the example of temperature diffusion in granite blocks presented in the previous Section 2.1. Thus, it is possible to modify the script we have already developed for the granite blocks to suit the iron bar problem. The obvious differences between the two problems are the dimensions of the domain and different materials involved. This will change the time scale of the model from years to hours. The new settings are

```
#Domain related.
mx = 1*m #meters - model length
my = .1*m #meters - model width
ndx = 100 # mesh steps in x direction
ndy = 1 # mesh steps in y direction - one dimension means one element
#PDE related
rho = 7874. *kg/m**3 #kg/m^{3} density of iron
cp = 449.*J/(kg*K) # J/Kg.K thermal capacity
rhocp = rho*cp
```

```
kappa = 80.*W/m/K   # watts/m.Kthermal conductivity
qH = 0 * J/(sec*m**3) # J/(sec.m^{3}) no heat source
Tref = 20 * Celsius  # base temperature of the rod
T0 = 100 * Celsius # temperature at heating element
tend= 0.5 * day # - time to end simulation
```

We also need to alter the initial value for the temperature. Now we need to set the temperature to T_0 at the left end of the rod where we have $x_0 = 0$ and T_{ref} elsewhere. Instead of `whereNegative` function we now use `whereZero` which returns the value one for those sample points where the argument (almost) equals zero and the value zero elsewhere. The initial temperature is set to

```
# ... set initial temperature ....
T= T0*whereZero(x[0])+Tref*(1-whereZero(x[0]))
```

2.2.1 Dirichlet Boundary Conditions

In the iron rod model we want to keep the initial temperature T_0 on the left side of the domain constant with time. This implies that when we solve the PDE Equation (2.5), the solution must have the value T_0 on the left hand side of the domain. As mentioned already in Section 2.1.3 where we discussed boundary conditions, this kind of scenario can be expressed using a **Dirichlet boundary condition**. Some people also use the term **constraint** for the PDE.

To define a Dirichlet boundary condition we need to specify where to apply the condition and determine what value the solution should have at these locations. In *escript* we use q and r to define the Dirichlet boundary conditions for a PDE. The solution u of the PDE is set to r for all sample points where q has a positive value. Mathematically this is expressed in the form;

$$u(x) = r(x) \text{ for any } x \text{ with } q(x) > 0 \tag{2.19}$$

In the case of the iron rod we can set

```
q=whereZero(x[0])
r=T0
```

to prescribe the value T_0 for the temperature at the left end of the rod where $x_0 = 0$. Here we use the `whereZero` function again which we have already used to set the initial value. Notice that r is set to the constant value T_0 for all sample points. In fact, values of r are used only where q is positive. Where q is non-positive, r may have any value as these values are not used by the PDE solver.

To set the Dirichlet boundary conditions for the PDE to be solved in each time step we need to add some statements;

```
mypde=LinearPDE(rod)
A=zeros((2,2)))
A[0,0]=kappa
q=whereZero(x[0])
mypde.setValue(A=A, D=rhocp/h, q=q, r=T0)
```

It is important to remark here that if a Dirichlet boundary condition is prescribed on the same location as any Neumann boundary condition, the Neumann boundary condition will be **overwritten**. This applies to Neumann boundary conditions that *escript* sets by default and those defined by the user.

Besides some cosmetic modification this is all we need to change. The total energy over time is shown in Figure (2.7). As heat is transferred into the rod by the heater the total energy is growing over time but reaches a plateau when the temperature is constant in the rod, see Figure (2.8). You will notice that the time scale of this model is several order of magnitudes faster than for the granite rock problem due to the different length scale and material parameters. In practice it can take a few model runs before the right time scale has been chosen[18].

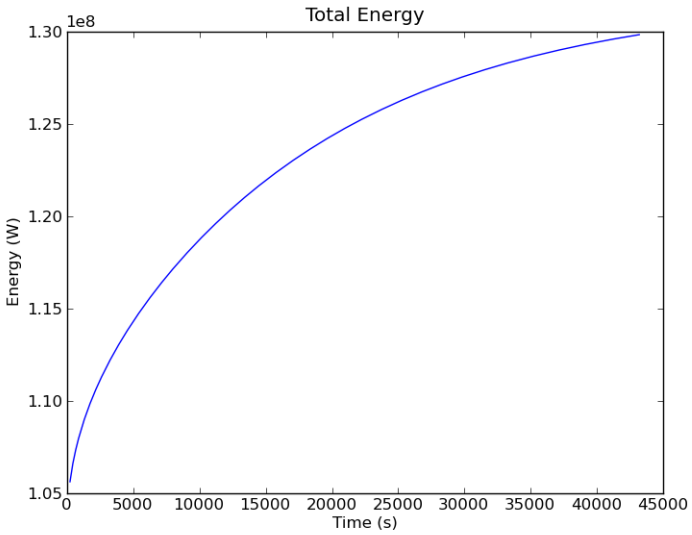


Figure 2.7: Example 2: Total Energy in the Iron Rod over Time (in seconds)

2.3 For the Reader

1. Move the boundary line between the two granite blocks to another part of the domain.

[18] An estimate of the time scale for a diffusion problem is given by the formula $\frac{\rho c_p L_0^2}{4\kappa}$, see http://en.wikipedia.org/wiki/Fick%27s_laws_of_diffusion

2. Split the domain into multiple granite blocks with varying temperatures.

3. Vary the mesh step size. Do you see a difference in the answers? What does happen with the compute time?

4. Insert an internal heat source (Hint: The internal heat source is given by q_H.)

5. Change the boundary condition for the iron rod example such that the temperature at the right end is kept at a constant level T_{ref}, which corresponds to the installation of a cooling element (Hint: Modify q and r).

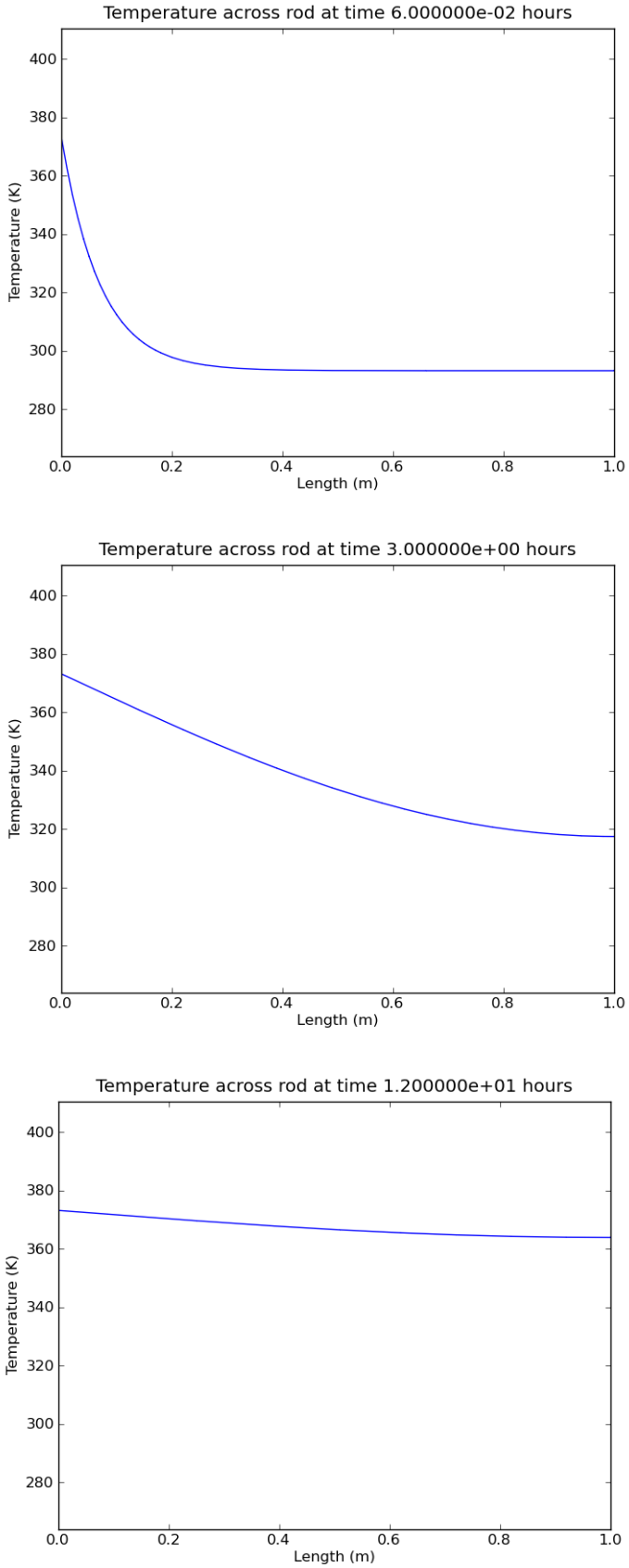


Figure 2.8: Example 2: Temperature (T) distribution in the iron rod at time steps 1, 50 and 200

CHAPTER

THREE

Heat Diffusion in Two Dimensions

The scripts referenced in this section are; example03a.py and cblib.py

Building upon our success from the 1D models, it is now prudent to expand our domain by another dimension. For this example we use a very simple magmatic intrusion as the basis for our model. The simulation will be a single event where some molten granite has formed a cylindrical dome at the base of some cold sandstone country rock. Assuming that the cylinder is very long we model a cross-section as shown in Figure (3.1). We will implement the same diffusion model as we have used for the granite blocks in Section 2.1 but will add the second spatial dimension and show how to define variables depending on the location of the domain. We use `onedheatdiff001b.py` as the starting point to develop this model.

3.1 Example 3: Two Dimensional Heat Diffusion for a basic Magmatic Intrusion

To expand upon our 1D problem, the domain must first be expanded. In fact, we have solved a two dimensional problem already but essentially ignored the second dimension. In our definition phase we create a square domain in x and y[1] that is 600 meters along each side Figure (3.1). Now we set the number of discrete spatial cells to 150 in both directions and the radius of the intrusion to 200 meters with the centre located at the 300 meter mark on the x-axis. Thus, the domain variables are;

```
mx = 600*m #meters - model length
```

[1] In *escript* the notation x_0 and x_1 is used for x and y, respectively.

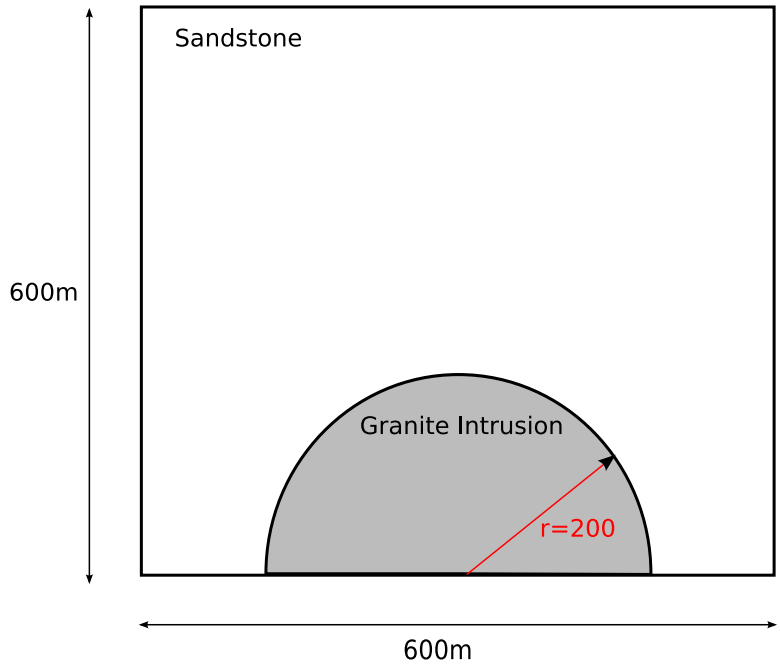


Figure 3.1: Example 3: 2D model: granitic intrusion of sandstone country rock

```
my = 600*m #meters - model width
ndx = 150 #mesh steps in x direction
ndy = 150 #mesh steps in y direction
r = 200*m #meters - radius of intrusion
ic = [300*m, 0] #coordinates of the centre of intrusion (meters)
qH=0.*J/(sec*m**3) #our heat source temperature is zero
```

As before we use

```
model = Rectangle(l0=mx,l1=my,n0=ndx, n1=ndy)
```

to generate the domain.

There are two fundamental changes to the PDE that we have discussed in Section 2.1. Firstly, because the material coefficients for granite and sandstone are different, we need to deal with PDE coefficients which vary with their location in the domain. Secondly, we need to deal with the second spatial dimension. We can investigate these two modifications at the same time. In fact, the temperature model Equation (2.1) we utilised in Section 2.1 applied for the 1D case with a constant material parameter only. For the more general case examined in this chapter, the correct model equation is

$$\rho c_p \frac{\partial T}{\partial t} - \frac{\partial}{\partial x}\kappa\frac{\partial T}{\partial x} - \frac{\partial}{\partial y}\kappa\frac{\partial T}{\partial y} = q_H \tag{3.1}$$

Notice that for the 1D case we have $\frac{\partial T}{\partial y} = 0$ and for the case of constant material parameters $\frac{\partial}{\partial x}\kappa = \kappa\frac{\partial}{\partial x}$ thus this new equation coincides with a simplified model equation for this case. It is more convenient to write this equation

using the ∇ notation as we have already seen in Equation (2.6);

$$\rho c_p \frac{\partial T}{\partial t} - \nabla \cdot \kappa \nabla T = q_H \tag{3.2}$$

We can easily apply the backward Euler scheme as before to end up with

$$\frac{\rho c_p}{h} T^{(n)} - \nabla \cdot \kappa \nabla T^{(n)} = q_H + \frac{\rho c_p}{h} T^{(n-1)} \tag{3.3}$$

which is very similar to Equation (2.9) used to define the temperature update in the simple 1D case. The difference is in the second order derivative term $\nabla \cdot \kappa \nabla T^{(n)}$. Under the light of the more general case we need to revisit the *escript* PDE form as shown in 2.16. For the 2D case with variable PDE coefficients the form needs to be read as

$$-\frac{\partial}{\partial x} A_{00} \frac{\partial u}{\partial x} - \frac{\partial}{\partial x} A_{01} \frac{\partial u}{\partial y} - \frac{\partial}{\partial y} A_{10} \frac{\partial u}{\partial x} - \frac{\partial}{\partial x} A_{11} \frac{\partial u}{\partial y} + Du = f \tag{3.4}$$

So besides the settings $u = T^{(n)}$, $D = \frac{\rho c_p}{h}$ and $f = q_H + \frac{\rho c_p}{h} T^{(n-1)}$ as we have used before (see Equation (2.10)) we need to set

$$A_{00} = A_{11} = \kappa; A_{01} = A_{10} = 0 \tag{3.5}$$

The fundamental difference to the 1D case is that A_{11} is not set to zero but κ, which brings in the second dimension. It is important to note that the coefficients of the PDE may depend on their location in the domain which does not influence the usage of the PDE form. This is very convenient as we can introduce spatial dependence to the PDE coefficients without modification to the way we call the PDE solver.

A very convenient way to define the matrix A in Equation (3.5) can be carried out using the Kronecker δ symbol[2]. The *escript* function `kronecker` returns this matrix;

$$\texttt{kronecker(model)} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \tag{3.6}$$

As the argument `model` represents a two dimensional domain the matrix is returned as a 2×2 matrix (in the case of a three-dimensional domain a 3×3 matrix is returned). The call

```
mypde.setValue(A=kappa*kronecker(model),D=rhocp/h)
```

sets the PDE coefficients according to Equation (3.5).

[2] see http://en.wikipedia.org/wiki/Kronecker_delta

We need to check the boundary conditions before we turn to the question: how do we set κ. As pointed out Equation (2.14) makes certain assumptions on the boundary conditions. In our case these assumptions translate to;

$$-n \cdot \kappa \nabla T^{(n)} = -n_0 \cdot \kappa \frac{\partial T^{(n)}}{\partial x} - n_1 \cdot \kappa \frac{\partial T^{(n)}}{\partial y} = 0 \tag{3.7}$$

which sets the normal component of the heat flux $-\kappa \cdot (\frac{\partial T^{(n)}}{\partial x}, \frac{\partial T^{(n)}}{\partial y})$ to zero. This means that the region is insulated which is what we want. On the left and right face of the domain where we have $(n_0, n_1) = (\mp 1, 0)$ this means $\frac{\partial T^{(n)}}{\partial x} = 0$ and on the top and bottom on the domain where we have $(n_0, n_1) = (\pm 1, 0)$ this is $\frac{\partial T^{(n)}}{\partial y} = 0$.

3.2 Setting variable PDE Coefficients

Now we need to look into the problem of how we define the material coefficients κ and ρc_p depending on their location in the domain. We can make use of the technique used in the granite block example in Section 2.1 to set up the initial temperature. However, the situation is more complicated here as we have to deal with a curved interface between the two sub-domains.

Prior to setting up the PDE, the interface between the two materials must be established. The distance $s \geq 0$ between two points $[x, y]$ and $[x_0, y_0]$ in Cartesian coordinates is defined as:

$$(x - x_0)^2 + (y - y_0)^2 = s^2 \tag{3.8}$$

If we define the point $[x_0, y_0]$ as ic which denotes the centre of the semi-circle of our intrusion, then for all the points $[x, y]$ in our model we can calculate a distance to ic. All the points that fall within the given radius r of our intrusion will have a corresponding value $s < r$ and all those belonging to the country rock will have a value $s > r$. By subtracting r from both of these conditions we find $s - r < 0$ for all intrusion points and $s - r > 0$ for all country rock points. Defining these conditions within the script is quite simple and is done using the following command:

```
bound = length(x-ic)-r #where the boundary will be located
```

This definition of the boundary can now be used with the `whereNegative` command again to help define the material constants and temperatures in our domain. If `kappai` and `kappac` are the thermal conductivities for the intrusion material granite and for the surrounding sandstone, then we set;

```
x=Function(model).getX()
bound = length(x-ic)-r
kappa = kappai * whereNegative(bound) + kappac * (1-whereNegative(bound))
mypde.setValue(A=kappa*kronecker(model))
```

Notice that we are using the sample points of the `Function` function space as expected for the PDE coefficient `A`[3]. The corresponding statements are used to set ρc_p.

Our PDE has now been properly established. The initial conditions for temperature are set out in a similar manner:

```
#defining the initial temperatures.
x=Solution(model).getX()
bound = length(x-ic)-r
T= Ti*whereNegative(bound)+Tc*(1-whereNegative(bound))
```

where `Ti` and `Tc` are the initial temperature in the regions of the granite and surrounding sandstone, respectively. It is important to notice that we reset `x` and `bound` to refer to the appropriate sample points of a PDE solution[4].

3.3 Contouring *escript* Data using `matplotlib`.

To complete our transition from a 1D to a 2D model we also need to modify the plotting procedure. As before we use `matplotlib` to do the plotting in this case a contour plot. For 2D contour plots `matplotlib` expects that the data are regularly gridded. We have no control over the location and ordering of the sample points used to represent the solution. Therefore it is necessary to interpolate our solution onto a regular grid.

In Section 2.1.10 we have already learned how to extract the x coordinates of sample points as `numpy` array to hand the values to `matplotlib`. This can easily be extended to extract both the x and the y coordinates;

```
import numpy as np
def toXYTuple(coords):
    coords = np.array(coords.toListOfTuples()) #convert to Tuple
    coordX = coords[:,0] #X components.
    coordY = coords[:,1] #Y components.
    return coordX,coordY
```

For convenience we have put this function into `clib.py` so we can use this function in other scripts more easily.

We now generate a regular 100×100 grid over the domain (mx and my are the dimensions in the x and y directions) which is done using the `numpy` function `linspace`.

```
from clib import toXYTuple
# get sample points for temperature as  for contouring
coordX, coordY = toXYTuple(T.getFunctionSpace().getX())
# create regular grid
xi = np.linspace(0.0,mx,75)
```

[3] For the experienced user: use `x=mypde.getFunctionSpace("A").getX()`.
[4] For the experienced user: use `x=mypde.getFunctionSpace("r").getX()`.

```
yi = np.linspace(0.0,my,75)
```

The values `[xi[k], yi[l]]` are the grid points.

The remainder of our contouring commands resides within a `while` loop so that a new contour is generated for each time step. For each time step the solution must be re-gridded for `matplotlib` using the `griddata` function. This function interprets irregularly located values `tempT` at locations defined by `coordX` and `coordY` as values at the new coordinates of a rectangular grid defined by `xi` and `yi`. The output is `zi`. It is now possible to use the `contourf` function which generates colour filled contours. The colour gradient of our plots is set via the command `pl.matplotlib.pyplot.autumn()`, other colours are listed on the `matplotlib` web page[5]. Our results are then contoured, visually adjusted using the `matplotlib` functions and then saved to a file. `pl.clf()` clears the figure in readiness for the next time iteration.

```
#grid the data.
zi = pl.matplotlib.mlab.griddata(coordX,coordY,tempT,xi,yi)
# contour the gridded data, plotting dots at the randomly spaced data points.
pl.matplotlib.pyplot.autumn()
pl.contourf(xi,yi,zi,10)
CS = pl.contour(xi,yi,zi,5,linewidths=0.5,colors='k')
pl.clabel(CS, inline=1, fontsize=8)
pl.axis([0,600,0,600])
pl.title("Heat diffusion from an intrusion.")
pl.xlabel("Horizontal Displacement (m)")
pl.ylabel("Depth (m)")
pl.savefig(os.path.join(save_path,"Tcontour%03d.png") %i)
pl.clf()
```

The function `pl.contour` is used to add labelled contour lines to the plot. The results for selected time steps are shown in Figure (3.2).

3.4 Advanced Visualisation using VTK

The scripts referenced in this section are; example03b.py

An alternative approach to `matplotlib` for visualisation is the usage of a package which is based on the Visualization Toolkit (VTK) library[6]. There is a variety of packages available. Here we use the package *Mayavi2*[7] as an example.

[5] see http://matplotlib.sourceforge.net/api/
[6] see http://www.vtk.org/
[7] see http://code.enthought.com/projects/mayavi/

Mayavi2 is an interactive, GUI driven tool which is really designed to visualise large three dimensional data sets where `matplotlib` is not suitable. But it is also very useful when it comes to two dimensional problems. The decision of which tool is the best can be subjective and users should determine which package they require and are most comfortable with. The main difference between using *Mayavi2* (or other VTK based tools) and `matplotlib` is that the actual visualisation is detached from the calculation by writing the results to external files and importing them into *Mayavi2*. In 3D the best camera position for rendering a scene is not obvious before the results are available. Therefore the user may need to try different settings before the best is found. Moreover, in many cases a 3D interactive visualisation is the only way to really understand the results (e.g. using stereographic projection).

To write the temperatures at each time step to data files in the VTK file format one needs to import `saveVTK` from the `esys.weipa` module and call it:

```
from esys.weipa import saveVTK
while t<=tend:
      i+=1 #counter
      t+=h #current time
      mypde.setValue(Y=qH+T*rhocp/h)
      T=mypde.getSolution()
      saveVTK(os.path.join(save_path,"data.%03d.vtu"%i, T=T)
```

The data files, e.g. `data.001.vtu`, contain all necessary information to visualise the temperature and can be directly processed by *Mayavi2*. Note that there is no re-gridding required. The file extension `.vtu` is automatically added if not supplied to `saveVTK`.

After you run the script you will find the result files `data.*.vtu` in the result directory `data/example03`. Run the command

```
>> mayavi2 -d data.001.vtu -m Surface &
```

from the result directory. *Mayavi2* will start up a window similar to Figure (3.3). The right hand side shows the temperature at the first time step. To show the results at other time steps you can click at the item `VTK XML file (data.001.vtu) (timeseries)` at the top left hand side. This will bring up a new window similar to the window shown in Figure (3.4). By clicking at the arrows in the top right corner you can move forwards and backwards in time. You will also notice the text **T** next to the item `Point scalars name`. The name **T** corresponds to the keyword argument name `T` we have used in the `saveVTK` call. In this menu item you can select other results you may have written to the output file for visualisation.

For the advanced user: Using the `matplotlib` to visualise spatially distributed data is not MPI compatible. However, the `saveVTK` function can be used with MPI. In fact, the function collects the values of the sample

points spread across processor ranks into a single file. For more details on writing scripts for parallel computing please consult the *user's guide*.

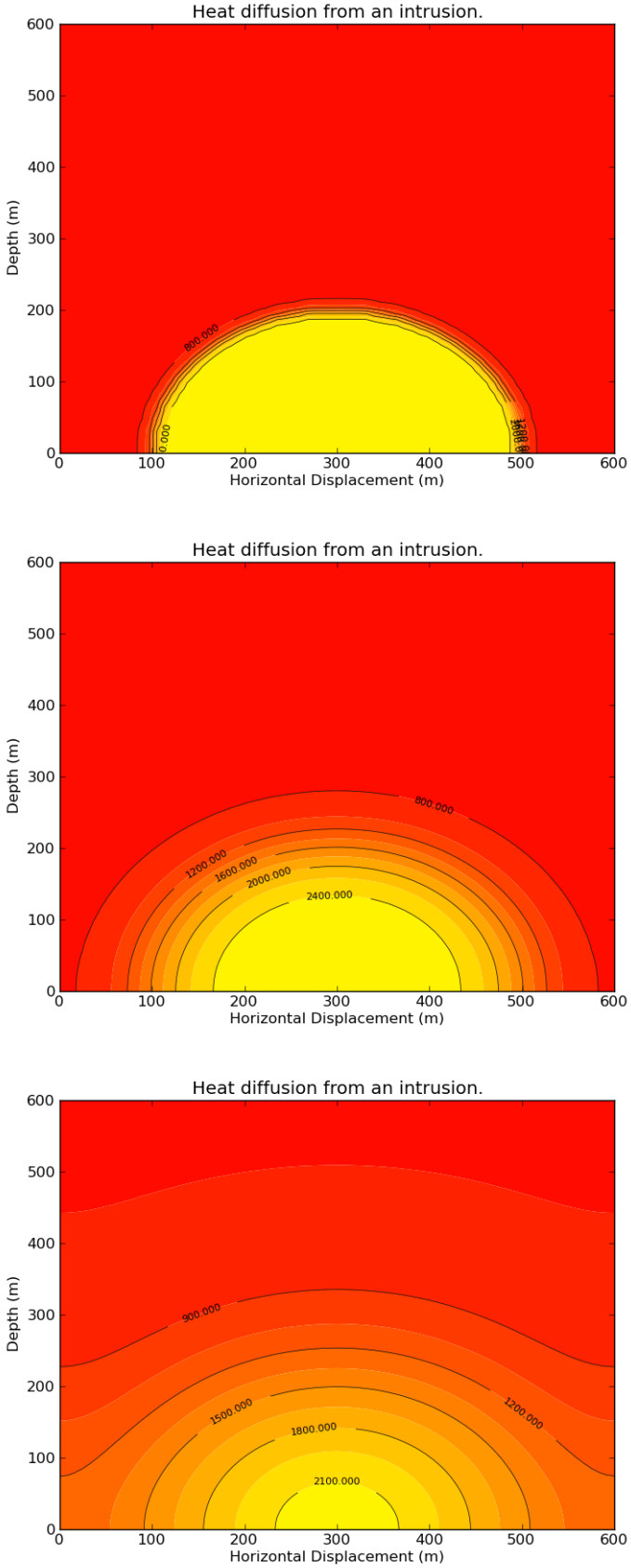


Figure 3.2: Example 3a: 2D model: Total temperature distribution (T) at time step 1, 20 and 200. Contour lines show temperature.

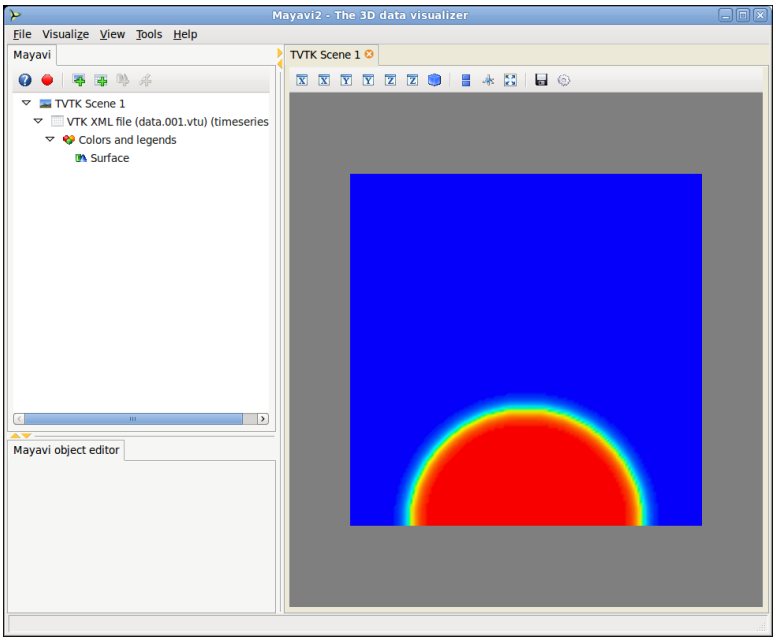


Figure 3.3: Example 3b: *Mayavi2* start up window

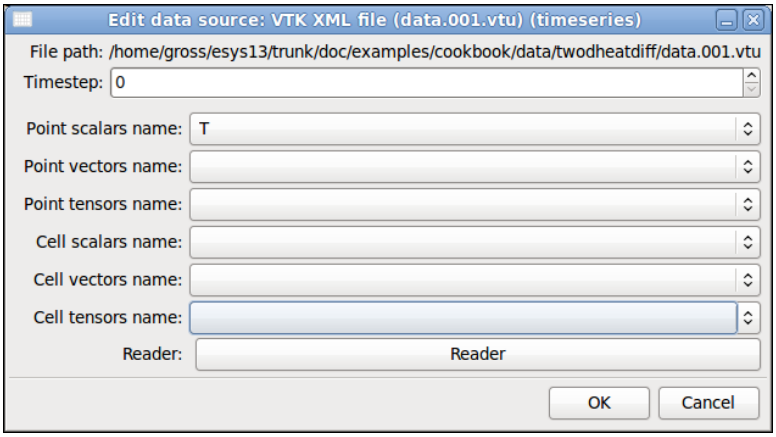


Figure 3.4: Example 3b: *Mayavi2* data control window

CHAPTER

FOUR

Complex Geometries

4.1 Steady-state Heat Refraction

In this chapter we demonstrate how to handle more complex geometries.

Steady-state heat refraction will give us an opportunity to investigate some of the richer features that the *escript* package has to offer. One of these is `esys.pycad`. The advantage of using `esys.pycad` is that it offers an easy method for developing and manipulating complex domains. In conjunction with `esys.pycad.gmsh` we can generate finite element meshes that conform to our domain's shape providing accurate modelling of interfaces and boundaries. Another useful function of `esys.pycad` is that we can tag specific areas of our domain with labels as we construct them. These labels can then be used in *escript* to define properties like material constants and source locations.

We proceed in this chapter by first looking at a very simple geometry. Whilst a simple rectangular domain is not very interesting the example is elaborated upon later by introducing an internal curved interface.

4.2 Example 4: Creating the Domain with `esys.pycad`

The scripts referenced in this section are; example04a.py

We modify the example in Chapter 3 in two ways: we look at the steady state case with slightly modified boundary conditions and use a more flexible tool to generate the geometry. Let us look at the geometry first.

We want to define a rectangular domain of width $5km$ and depth $6km$ below the surface of the Earth. The domain is subject to a few conditions. The temperature is known at the surface and the basement has a known heat

flux. Each side of the domain is insulated and the aim is to calculate the final temperature distribution.

In `esys.pycad` there are a few primary constructors that build upon each other to define domains and boundaries. The ones we use are:

```
from esys.pycad import *
Point() #Create a point in space.
Line() #Creates a line from a number of points.
CurveLoop() #Creates a closed loop from a number of lines.
PlaneSurface() #Creates a surface based on a CurveLoop
```

So to construct our domain as shown in Figure (4.1), we first need to create the corner points. From the corner points we build the four edges of the rectangle. The four edges then form a closed loop which defines our domain as a surface. We start by inputting the variables we need to construct the model.

```
width=5000.0*m   #width of model
depth=-6000.0*m  #depth of model
```

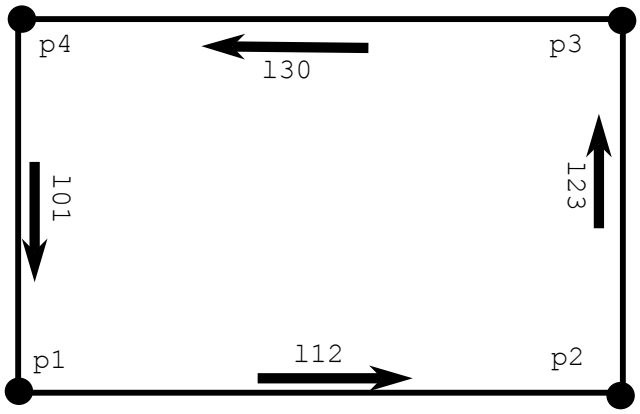


Figure 4.1: Example 4: Rectangular Domain for `esys.pycad`

The variables are then used to construct the four corners of our domain, which from the origin has the dimensions of 5000 meters width and -6000 meters depth. This is done with the `Point()` function which accepts x, y and z coordinates. Our domain is in two dimensions so z should always be zero.

```
# Overall Domain
p0=Point(0.0,       0.0, 0.0)
p1=Point(0.0,     depth, 0.0)
p2=Point(width, depth, 0.0)
p3=Point(width,    0.0, 0.0)
```

Now lines are defined using our points. This forms a rectangle around our domain:

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

Note that lines have a direction. These lines form the basis for our domain boundary, which is a closed loop.

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

Be careful to define the curved loop in an **anti-clockwise** manner otherwise the meshing algorithm may fail. Finally we can define the domain as

```
rec = PlaneSurface(c)
```

At this point the introduction of the curved loop seems to be unnecessary but this concept plays an important role if holes are introduced.

Now we are ready to hand over the domain `rec` to a mesher which subdivides the domain into triangles (or tetrahedra in 3D). In our case we use `esys.pycad.gmsh`. We create an instance of the `Design` class which will handle the interface to `esys.pycad.gmsh`:

```
from esys.pycad.gmsh import Design
d=Design(dim=2, element_size=200*m)
```

The argument `dim` defines the spatial dimension of the domain[1]. The second argument `element_size` defines the element size which is the maximum length of a triangle edge in the mesh. The element size needs to be chosen with care in order to avoid very dense meshes. If the mesh is too dense, the computational time will be long but if the mesh is too sparse, the modelled result will be poor. In our case with an element size of 200m and a domain length of 6000m we will end up with about $\frac{6000m}{200m} = 30$ triangles in each spatial direction. So we end up with about $30 \times 30 = 900$ triangles which is a size that can be handled easily. The domain `rec` can simply be added to the `Design`;

```
d.addItem(rec)
```

We have the plan to set a heat flux on the bottom of the domain. One can use the masking technique to do this but `esys.pycad` offers a more convenient technique called tagging. With this technique items in the domain are named using the `PropertySet` class. We can then later use this name to set values specifically for those sample points located on the named items. Here we name the bottom face of the domain where we will set the heat influx:

```
ps=PropertySet("linebottom",l12))
d.addItem(ps)
```

[1] If `dim`=3 the rectangle would be interpreted as a surface in the three dimensional space.

Now we are ready to hand over the `Design` to *finley*:

```
from esys.finley import MakeDomain
domain=MakeDomain(d)
```

The `domain` object can now be used in the same way like the return object of the `Rectangle` object we have used previously to generate a mesh. It is common practice to separate the mesh generation from the PDE solution. The main reason for this is that mesh generation can be computationally very expensive in particular in 3D. So it is more efficient to generate the mesh once and write it to a file. The mesh can then be read in every time a new simulation is run. *finley* supports this in the following way[2]:

```
# write domain to a text file
domain.write("example04.fly")
```

and then for reading in another script:

```
# read domain from text file
from esys.finley import ReadMesh
domain =ReadMesh("example04.fly")
```

Before we discuss how to solve the PDE for this problem, it is useful to present two additional options of the `Design` class. These allow the user to access the script which is used by `esys.pycad.gmsh` to generate the mesh as well as the generated mesh itself. This is done by setting specific names for these files:

```
d.setScriptFileName("example04.geo")
d.setMeshFileName("example04.msh")
```

Conventionally the extension `geo` is used for the script file of the `esys.pycad.gmsh` geometry and the extension `msh` for the mesh file. Normally these files are deleted after usage. Accessing these files can be helpful to debug the generation of more complex geometries. The geometry and the mesh can be visualised from the command line using

```
gmsh example04.geo  # show geometry
gmsh example04.msh  # show mesh
```

The mesh is shown in Figure (4.2).

[2] An alternative is using the `dump` and `load` functions. They work with a binary format and tend to be much smaller.

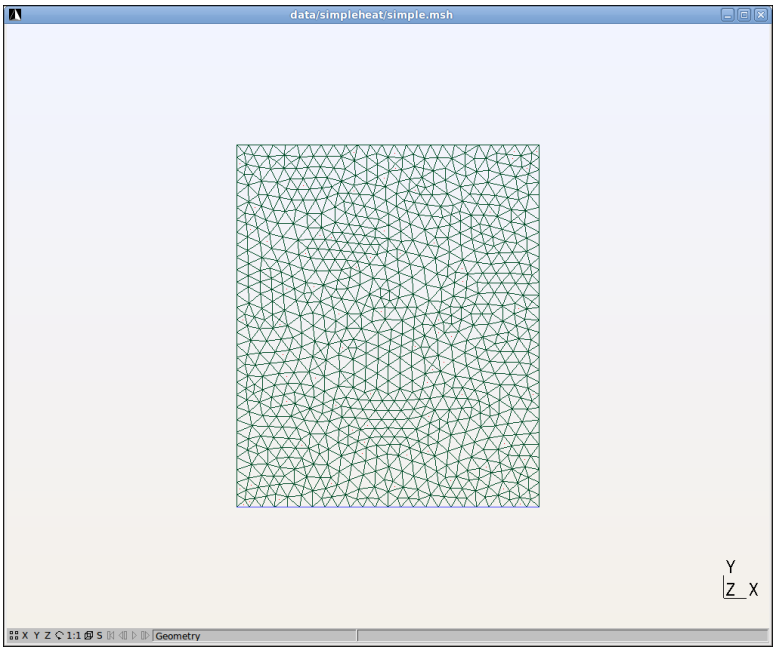


Figure 4.2: Example 4a: Mesh over rectangular domain, see Figure (4.1)

4.3 The Steady-state Heat Equation

The scripts referenced in this section are; example04b.py, cblib

A temperature equilibrium or steady state is reached when the temperature distribution in the model does not change with time. To calculate the steady state solution the time derivative term in Equation (3.2) needs to be set to zero;

$$-\nabla \cdot \kappa \nabla T = q_H \tag{4.1}$$

This PDE is easier to solve than the PDE in Equation (3.3), as no time steps (iterations) are required. The `D` term from Equation (3.3) is simply dropped in this case.

```
mypde=LinearPDE(domain)
mypde.setValue(A=kappa*kronecker(model), Y=qH)
```

The temperature at the top face of the domain is known as `Ttop` ($= 20C$). In Section 2.2 we have already discussed how this constraint is added to the PDE:

```
x=Solution(domain).getX()
mypde.setValue(q=whereZero(x[1]-sup(x[1])),r=Ttop)
```

Notice that we use the `sup` function to calculate the maximum of y coordinates of the relevant sample points.

In all cases so far we have assumed that the domain is insulated which translates into a zero normal flux $-n \cdot \kappa \nabla T$, see Equation (3.7). In the modelling set-up of this chapter we want to set the normal heat flux at the bottom to `qin` while still maintaining insulation at the left and right face. Mathematically we can express this as

$$-n \cdot \kappa \nabla T = q_S \tag{4.2}$$

where q_S is a function of its location on the boundary. Its value becomes zero for locations on the left or right face of the domain while it has the value `qin` at the bottom face. Notice that the value of q_S at the top face is not relevant as we prescribe the temperature here. We could define q_S by using the masking techniques demonstrated earlier. The tagging mechanism provides an alternative and in many cases more convenient way of defining piecewise constant functions such as q_S. Recall now that the bottom face was denoted with the name `linebottom` when we defined the domain. We can use this now to create q_S;

```
qS=Scalar(0,FunctionOnBoundary(domain))
qS.setTaggedValue("linebottom",qin)
```

In the first line `qS` is defined as a scalar value over the sample points on the boundary of the domain. It is initialised to zero for all sample points. In the second statement the values for those sample points which are located on the

line marked by `linebottom` are set to `qin`.

The Neumann boundary condition assumed by *escript* has the form

$$n \cdot A \cdot \nabla u = y \tag{4.3}$$

In comparison to the version in Equation (2.14) we have used so far the right hand side is now the new PDE coefficient y. As we have not specified y in our previous examples, *escript* has assumed the value zero for y. A comparison of Equation (4.3) and Equation (4.2) reveals that one needs to choose $y = -q_S$;

```
qS=Scalar(0,FunctionOnBoundary(domain))
qS.setTaggedValue("linebottom",qin)
mypde.setValue(y=-qS)
```

To plot the results we use the `matplotlib` library as shown in Section 3.3. For convenience the interpolation of the temperature to a rectangular grid for contour plotting is made available via the `toRegGrid` function in the `cblib` module. Your result should look similar to Figure (4.3).

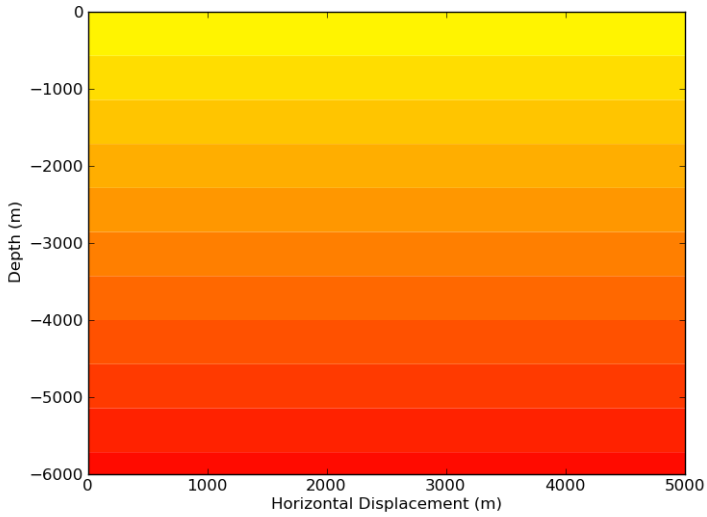


Figure 4.3: Example 4b: Result of simple steady state heat problem

4.4 Example 5: A Heat Refraction Model

The scripts referenced in this section are; example05a.py and cblib.py

Our heat refraction model will be a large anticlinal structure that is subject to a constant temperature at the surface and experiencing a steady heat flux at its base. Our aim is to show that the temperature flux across the surface is not linear from bottom to top, but is in fact warped by the structure of the model. The heat flow pattern demonstrates the dependence upon the material properties and the shape of the interface.

The script of Section 4.2 is modified by subdividing the block into two parts. The curve separating the two blocks is given as a spline, see Figure (4.4). The data points used to define the curve may be imported from a database of measurements (*e.g.* borehole depth data), but for simplicity it is assumed here that the coordinates are known in an analytic form.

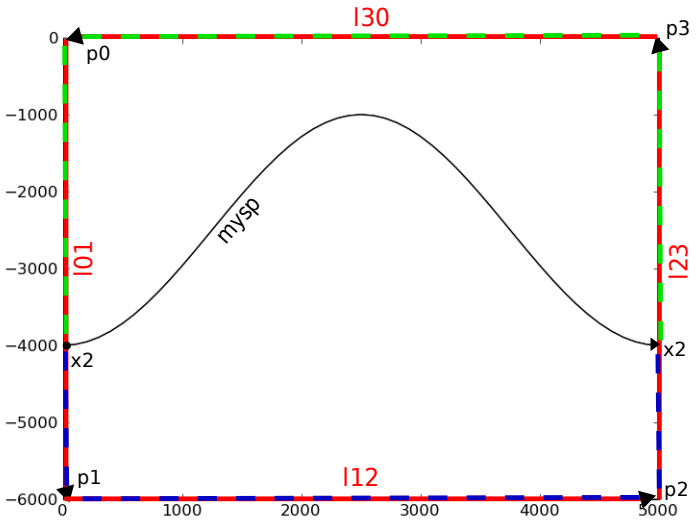


Figure 4.4: Example 5a: Heat refraction model with point and line labels

There are two modes available in this example. When `modal=1`, this indicates to the script that the model should be an anticline. Otherwise, when `modal=-1`, the model is a syncline. The modal operator simply changes the orientation of the boundary function so that it is either upwards or downwards curving. A `save_path` has also been defined so that we can easily separate our data from other examples and our scripts.

It is now possible to start defining our domain and boundaries.

The curve defining our clinal structure is located approximately in the middle of the domain and has a sinusoidal shape. We define the curve by generating points at discrete intervals; 51 in this case, and then create a smooth curve through the points using the `Spline()` function.

```
# Material Boundary
```

```
x=[ Point(i*dsp\
    ,-dep_sp+modal*orit*h_sp*cos(pi*i*dsp/dep_sp+pi))\
     for i in range(0,sspl)\
    ]
mysp = Spline(*tuple(x)) #*tuple() forces x to become a tuple
```

The start and end points of the spline can be returned to help define the material boundaries.

```
x1=mysp.getStartPoint()
x2=mysp.getEndPoint()
```

The top block or material above the clinal/spline boundary is defined in an **anti-clockwise** manner by creating lines and then a closed loop. By meshing the sub-domain we also need to assign it a planar surface.

```
# TOP BLOCK
tbl1=Line(p0,x1)
tbl2=mysp
tbl3=Line(x2,p3)
l30=Line(p3, p0)
tblockloop = CurveLoop(tbl1,tbl2,tbl3,l30)
tblock = PlaneSurface(tblockloop)
```

This process is repeated for every other sub-domain. In this example there is only one other, the bottom block. The process is similar to the top block but with a few differences. The spline points must be reversed by setting the spline as negative.

```
bbl4=-mysp
```

This reverse spline option unfortunately does not work for the `getLoopCoords` command, however, the `matplotlib` polygon tool will accept clock-wise oriented points so we can define a new curve.

```
#clockwise check
bblockloop=CurveLoop(mysp,Line(x2,p2),Line(p2,p1),Line(p1,x1))
```

The last few steps in creating the domain require that the previously defined domain and sub-domain points are submitted to generate a mesh that can be imported into *escript*. To initialise the mesh it first needs some design parameters. In this case we have 2 dimensions `dim` and a specified number of finite elements that need to be applied to the domain `element_size`. It then becomes a simple task of adding the sub-domains and flux boundaries to the design. Each element of our model can be given an identifier which makes it easier to define the sub-domain properties in the solution script. This is done using the `PropertySet()` function. The geometry and mesh are then saved so the *escript* domain can be created.

```
# Create a Design which can make the mesh
d=Design(dim=2, element_size=200)
# Add the sub-domains and flux boundaries.
d.addItems(PropertySet("top",tblock),PropertySet("bottom",bblock),\
                    PropertySet("linebottom",l12))
# Create the geometry, mesh and escript domain
d.setScriptFileName(os.path.join(save_path,"example05.geo"))
d.setMeshFileName(os.path.join(save_path,"example05.msh"))
domain=MakeDomain(d,optimizeLabeling=True)
```

The creation of our domain and its mesh is now complete.

With the mesh imported it is now possible to use our tagging property to set up our PDE coefficients. In this case κ is set via the `setTaggedValue()` function which takes two arguments, the name of the tagged points and the value to assign to them.

```
# set up kappa (thermal conductivity across domain) using tags
kappa=Scalar(0,Function(domain))
kappa.setTaggedValue("top",2.0*W/m/K)
kappa.setTaggedValue("bottom",4.0*W/m/K)
```

No further changes are required to the PDE solution step, see Figure (4.5) for the result.

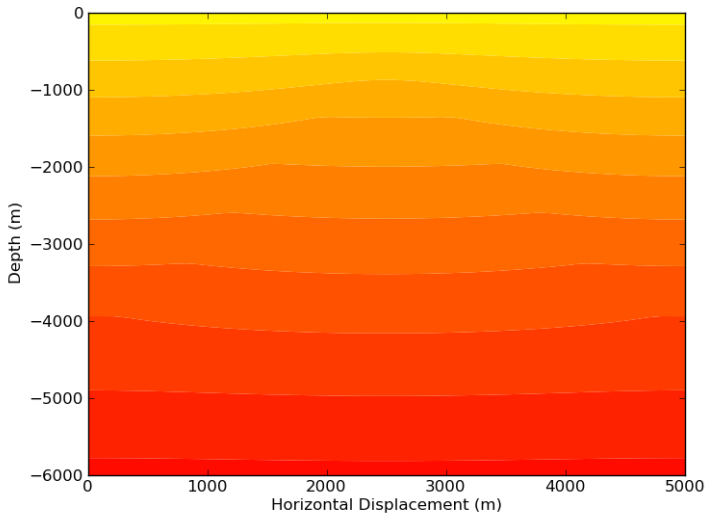


Figure 4.5: Example 5a: Temperature Distribution in the Heat Refraction Model

4.5 Line Profiles of 2D Data

The scripts referenced in this section are; example05b.py and cblib.py

We want to investigate the profile of the data of the last example. Of particular interest is the depth profile of the heat flux which is the second component of $-\kappa\nabla T$. The script from the previous section is extended to show how a vertical profile can be plotted.

The first important piece of information, is that *escript* assumes that $-\kappa\nabla T$ is not smooth and that the point values of this solution are defined at numerical interpolation points. This assumption is reasonable as the flux is the product of the piecewise constant function κ and the gradient of the temperature T which has a discontinuity at the rock interface. Before plotting this function we need to smooth the solution using the `Projector()` class;

```
from esys.escript.pdetools import Projector
proj=Projector(domain)
qu=proj(-kappa*grad(T))
```

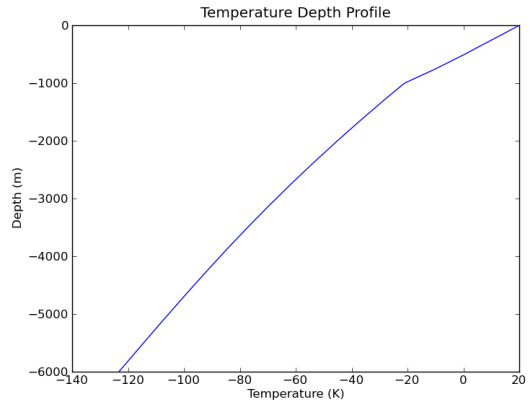
The `proj` object provides a mechanism to distribute values given at the numerical interpolation points to the nodes of the FEM mesh - the heat flux in this example. `qu` has the same function space as the temperature `T`. The smoothed flux is interpolated to a regular 200×200 grid via;

```
xiq,yiq,ziq = toRegGrid(qu[1],200,200)
```

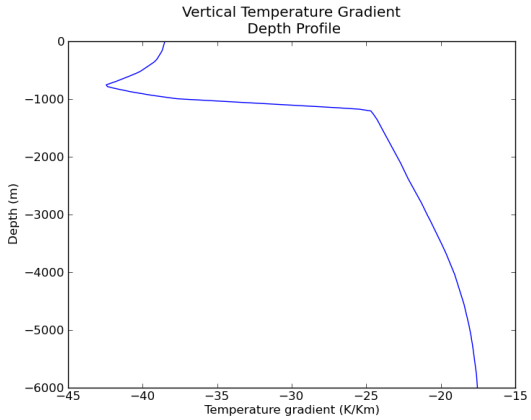
using the `toRegGrid` function from the cookbook library which we are using for the contour plot. At return `ziq[j,i]` is the value of vertical heat flux at point (`xiq[i]`,`yiq[j]`). We can easily create deep profiles now by plotting slices `ziq[:,i]` over `yiq`. The following script creates a deep profile at $x_0 = \frac{width}{2}$;

```
cut=int(len(xiq)/2)
pl.plot(ziq[:,cut]*1000.,yiq)
pl.title("Vertical Heat Flow Depth Profile")
pl.xlabel("Heat Flow (mW/m^2)")
pl.ylabel("Depth (m)")
pl.savefig(os.path.join(save_path,"hf.png"))
```

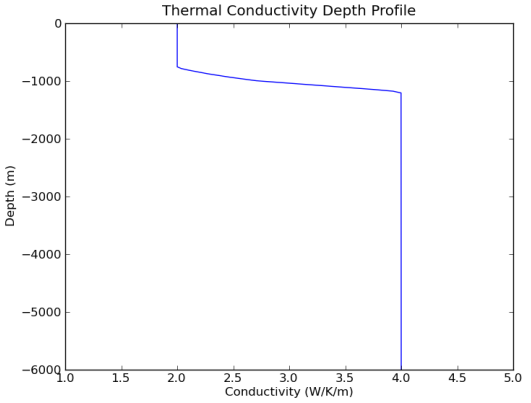
This process can be repeated for various variations of the solution. Figure (4.6) shows temperature, temperature gradient, thermal conductivity and heat flow.



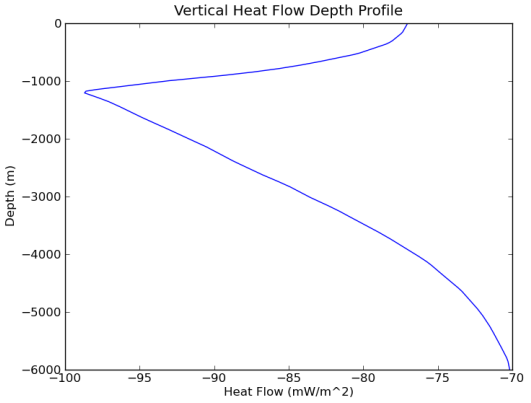
(a) Temperature Depth Profile



(b) Temperature Gradient Depth Profile



(c) Thermal Conductivity Profile



(d) Heat Flow Depth Profile

Figure 4.6: Example 5b: Depth profiles down centre of model

4.6 Arrow Plots in `matplotlib`

The scripts referenced in this section are; example05c.py and cblib.py

The distribution of the flux $-\kappa\nabla T$ is now visualised over the domain and the results are plotted in Figure (4.7). The plot puts together three components. A contour plot of the temperature, a coloured representation of the two sub-domains where colour represents the thermal conductivity in the particular region, and finally the arrows representing the local direction of the steepest gradient of the flux.

Contours have already been discussed in Section 3.3. To show sub-domains, we need to go back to `esys.pycad` data to get the points used to describe the boundary of the sub-domains. We have created the `CurveLoop` class object `tblockloop` to define the boundary of the upper sub-domain. We use the `getPolygon()` method of `CurveLoop` to get access to the `Points` used to define the boundary. The statement

```
[ p.getCoordinates() for p in tblockloop.getPolygon() ]
```

creates a list of the node coordinates of all the points in question. In order to simplify the selection of the x and y coordinates the list is converted into `numpy` array. To add the area coloured in brown to the plot we use;

```
import pylab as pl
import numarray as np
tpg=np.array([p.getCoordinates() for p in tblockloop.getPolygon() ])
pl.fill(tpg[:,0],tpg[:,1],'brown',label='2 W/m/k',zorder=-1000)
```

The same code is applied to `bblockloop` to create the red area for this sub-domain.

To plot vectors representing the flux orientation we use the `quiver` function in `pylab`. The function places vectors at locations in the domain. For instance one can plot vectors at the locations of the sample points used by *escript* to represent the flux `-kappa*grad(T)`. As a vector is plotted at each sample point one typically ends up with too many vectors. So one needs to select a subset of points as follows:

First we create a coarse grid of points on a rectangular mesh, e.g. 20×20 points. Here we choose a grid of points which are located at the centre of a `nx` $\times$ `ny` grid;

```
dx = width/nx # x spacing
dy = depth/ny # y spacing
grid = [ ] # the grid points
for j in xrange(0,ny-1):
    for i in xrange(0,nx-1):
            grid.append([dx/2+dx*i,dy/2+dy*j])
```

With the `Locator` function *escript* provides a mechanism to identify sample points that are closest to the grid points we have selected and to retrieve the data at these points;

```
from esys.escript.pdetools import Locator
flux=-kappa*grad(T)
fluxLoc = Locator(flux.getFunctionSpace(),grid)
subflux= fluxLoc(flux)
```

`subflux` now contains a list of flux components at certain sample points. To get the list of the sample point coordinates one can use the `getX()` method of the `Locator`;

```
subfluxloc = fluxLoc.getX()
```

To simplify the selection of x and y components it is convenient to transform `subflux` and `subfluxloc` to `numpy` arrays `xflux`, `flux`. This function is implemented in the `subsample` function within the `clib.py` file so we can use it in other examples. One can easily use this function to create a vector plot of the flux;

```
from cblib import subsample
xflux, flux=subsample(-kappa*grad(T), nx=20, ny=20)
pl.quiver(xflux[:,0],xflux[:,1],flux[:,0],flux[:,1], angles='xy',color="white")
```

Finally, we add a title and labels;

```
pl.title("Heat Refraction across a clinal structure.")
pl.xlabel("Horizontal Displacement (m)")
pl.ylabel("Depth (m)")
pl.title("Heat Refraction across a clinal structure \n with gradient quivers.")
pl.savefig(os.path.join(saved_path,"flux.png"))
```

to get the desired result, see Figure (4.7).

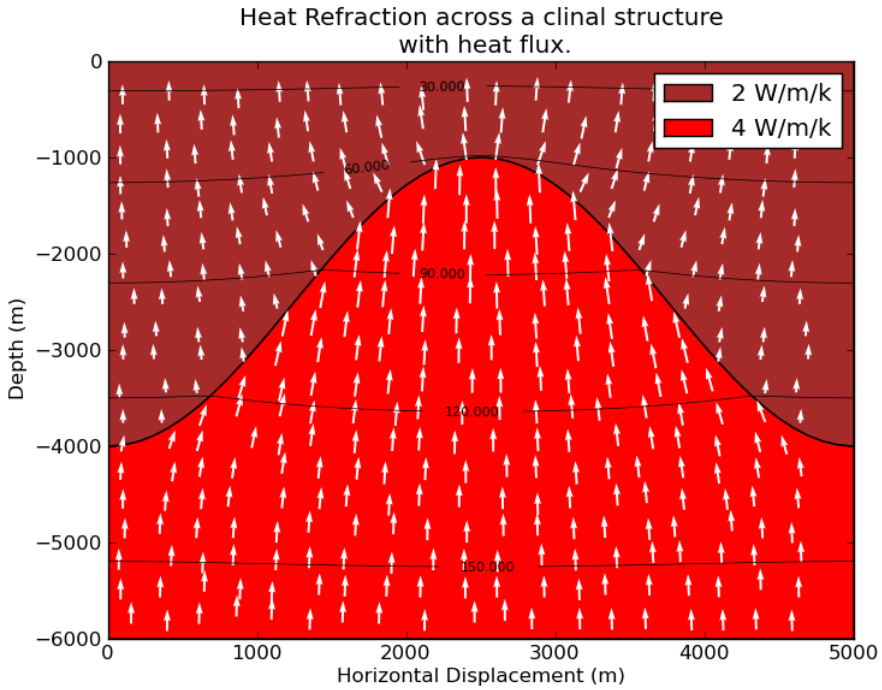


Figure 4.7: Example 5c: Heat refraction model with gradient indicated by vectors

4.7 Example 6:Fault and Overburden Model

The scripts referenced in this section are; example06.py and cblib.py

A slightly more complicated model can be found in the example file *heatrefraction2_solver.py* where three blocks are used within the model, see Figure (4.8). It is left to the reader to work through this example.

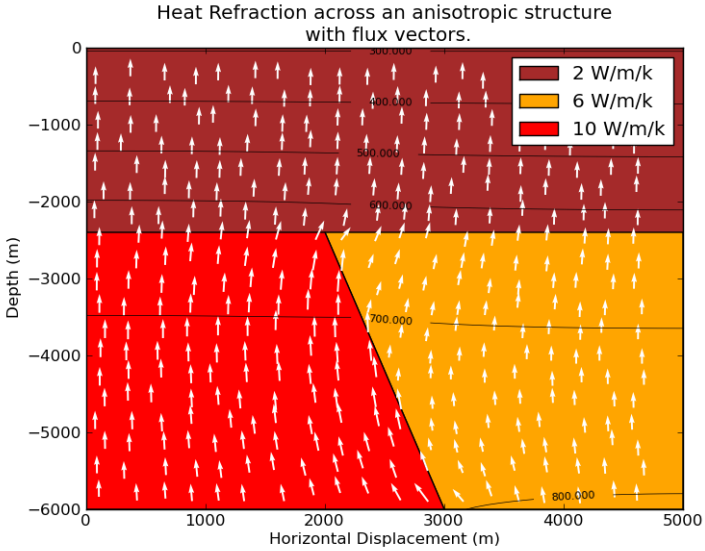


Figure 4.8: Example 6: Heat refraction model with three blocks and heat flux

CHAPTER

FIVE

Acoustic Wave Propagation

The acoustic wave equation governs the propagation of pressure waves. Wave types that obey this law tend to travel in liquids or gases where shear waves or longitudinal style wave motion is not possible. An obvious example is sound waves.

The acoustic wave equation is defined as;

$$\nabla^2 p - \frac{1}{c^2}\frac{\partial^2 p}{\partial t^2} = 0 \tag{5.1}$$

where p is the pressure, t is the time and c is the wave velocity. In this chapter the acoustic wave equation is demonstrated. Important steps include the translation of the Laplacian ∇^2 to the *escript* general form, the stiff equation stability criterion and solving for the displacement or acceleration solution.

5.1 The Laplacian in *escript*

The Laplacian operator which can be written as Δ or ∇^2, is calculated via the divergence of the gradient of the object, which in this example is the scalar p. Thus we can write;

$$\nabla^2 p = \nabla \cdot \nabla p = \sum_i^n \frac{\partial^2 p}{\partial x_i^2} \tag{5.2}$$

For the two dimensional case in Cartesian coordinates Equation 5.2 becomes;

$$\nabla^2 p = \frac{\partial^2 p}{\partial x^2} + \frac{\partial^2 p}{\partial y^2} \tag{5.3}$$

In *escript* the Laplacian is calculated using the divergence representation and the intrinsic functions *grad()* and *trace()*. The function *grad* will return the spatial gradients of an object. For a rank 0 solution, this is of the form;

$$\nabla p = \left[\frac{\partial p}{\partial x_0}, \frac{\partial p}{\partial x_1} \right] \tag{5.4}$$

Larger ranked solution objects will return gradient tensors. For example, a pressure field which acts in the directions p_0 and p_1 would return;

$$\nabla p = \begin{bmatrix} \frac{\partial p_0}{\partial x_0} & \frac{\partial p_1}{\partial x_0} \\ \frac{\partial p_0}{\partial x_1} & \frac{\partial p_1}{\partial x_1} \end{bmatrix} \tag{5.5}$$

Equation 5.4 corresponds to the Linear PDE general form value X. Notice however, that the general form contains the term $X_{i,j}$[1], hence for a rank 0 object there is no need to do more then calculate the gradient and submit it to the solver. In the case of the rank 1 or greater object, it is also necessary to calculate the trace. This is the sum of the diagonal in Equation 5.5.

Thus when solving for equations containing the Laplacian one of two things must be completed. If the object `p` is less than rank 1 the gradient is calculated via;

```
gradient=grad(p)
```

and if the object is greater then or equal to a rank 1 tensor, the trace of the gradient is calculated.

```
gradient=trace(grad(p))
```

These values can then be submitted to the PDE solver via the general form term X. The Laplacian is then computed in the solution process by taking the divergence of X.

Note, if you are unsure about the rank of your tensor, the *getRank* command will return the rank of the PDE object.

```
rank = p.getRank()
```

5.2 Numerical Solution Stability

Unfortunately, the wave equation belongs to a class of equations called **stiff** PDEs. These types of equations can be difficult to solve numerically as they tend to oscillate about the exact solution, which can eventually lead to a catastrophic failure. To counter this problem, explicitly stable schemes like the backwards Euler method, and correct parameterisation of the problem are required.

There are two variables which must be considered for stability when numerically trying to solve the wave

[1] This is the first derivative in the j^{th} direction for the i^{th} component of the solution.

equation. For linear media, the two variables are related via;

$$f = \frac{v}{\lambda} \tag{5.6}$$

The velocity v that a wave travels in a medium is an important variable. For stability the analytical wave must not propagate faster then the numerical wave is able to, and in general, needs to be much slower then the numerical wave. For example, a line 100m long is discretised into 1m intervals or 101 nodes. If a wave enters with a propagation velocity of 100m/s then the travel time for the wave between each node will be 0.01 seconds. The time step, must therefore be significantly less then this. Of the order $10E-4$ would be appropriate. This stability criterion is known as the Courant–Friedrichs–Lewy condition given by

$$dt = f \cdot \frac{dx}{v} \tag{5.7}$$

where dx is the mesh size and f is a safety factor. To obtain a time step of $10E-4$, a safety factor of $f = 0.1$ was used.

The wave frequency content also plays a part in numerical stability. The Nyquist-sampling theorem states that a signals bandwidth content will be accurately represented when an equispaced sampling rate f_n is equal to or greater then twice the maximum frequency of the signal f_s, or;

$$f_n \geqslant f_s \tag{5.8}$$

For example, a 50Hz signal will require a sampling rate greater then 100Hz or one sample every 0.01 seconds. The wave equation relies on a spatial frequency, thus the sampling theorem in this case applies to the solution mesh spacing. This relationship confirms that the frequency content of the input signal directly affects the time discretisation of the problem.

To accurately model the wave equation with high resolutions and velocities means that very fine spatial and time discretisation is necessary for most problems. This requirement makes the wave equation arduous to solve numerically due to the large number of time iterations required in each solution. Models with very high velocities and frequencies will be the worst affected by this problem.

5.3 Displacement Solution

The scripts referenced in this section are; example07a.py

We begin the solution to this PDE with the centred difference formula for the second derivative;

$$f''(x) \approx \frac{f(x+h-2f(x)+f(x-h)}{h^2} \tag{5.9}$$

substituting Equation 5.9 for $\frac{\partial^2 p}{\partial t^2}$ in Equation 5.1;

$$\nabla^2 p - \frac{1}{c^2 h^2}\left[p_{(t+1)} - 2p_{(t)} + p_{(t-1)}\right] = 0 \tag{5.10}$$

Rearranging for $p_{(t+1)}$;

$$p_{(t+1)} = c^2 h^2 \nabla^2 p_{(t)} + 2p_{(t)} - p_{(t-1)} \tag{5.11}$$

this can be compared with the general form of the `esys.escript.LinearPDEs` module and it becomes clear that $D = 1$, $X_{i,j} = -c^2 h^2 \nabla^2 p_{(t)}$ and $Y = 2p_{(t)} - p_{(t-1)}$.

The solution script is similar to others that we have created in previous chapters. The general steps are;

1. The necessary libraries must be imported.
2. The domain needs to be defined.
3. The time iteration and control parameters need to be defined.
4. The PDE is initialised with source and boundary conditions.
5. The time loop is started and the PDE is solved at consecutive time steps.
6. All or select solutions are saved to file for visualisation later on.

Parts of the script which warrant more attention are the definition of the source, visualising the source, the solution time loop and the VTK data export.

5.3.1 Pressure Sources

As the pressure is a scalar, one need only define the pressure for two time steps prior to the start of the solution loop. Two known solutions are required because the wave equation contains a double partial derivative with respect to time. This is often a good opportunity to introduce a source to the solution. This model has the source located at it's centre. The source should be smooth and cover a number of samples to satisfy the frequency stability criterion. Small sources will generate high frequency signals. Here, when using a rectangular domain, the source is defined by a cosine function.

```
U0=0.01 # amplitude of point source
xc=[500,500] #location of point source
```

```
# define small radius around point xc
src_radius = 30
# for first two time steps
u=U0*(cos(length(x-xc)*3.1415/src_radius)+1)*\
        whereNegative(length(x-xc)-src_radius)
u_m1=u
```

5.3.2 Visualising the Source

There are two options for visualising the source. The first is to export the initial conditions of the model to VTK, which can be interpreted as a scalar surface in Mayavi2. The second is to take a cross section of the model which will require the *Locator* function. First `Locator` must be imported;

```
from esys.escript.pdetools import Locator
```

The function can then be used on the domain to locate the nearest domain node to the point or points of interest.

It is now necessary to build a list of (x, y) locations that specify where are model slice will go. This is easily implemented with a loop;

```
cut_loc=[]
src_cut=[]
for i in range(ndx/2-ndx/10,ndx/2+ndx/10):
    cut_loc.append(xstep*i)
    src_cut.append([xstep*i,xc[1]])
```

We then submit the output to `Locator` and finally return the appropriate values using the `getValue` function.

```
src=Locator(mydomain,src_cut)
src_cut=src.getValue(u)
```

It is then a trivial task to plot and save the output using `matplotlib` (Figure 5.1).

```
pl.plot(cut_loc,src_cut)
pl.axis([xc[0]-src_radius*3,xc[0]+src_radius*3,0.,2*U0])
pl.savefig(os.path.join(savepath,"source_line.png"))
```

5.3.3 Point Monitoring

In the more general case where the solution mesh is irregular or specific locations need to be monitored, it is simple enough to use the *Locator* function.

```
rec=Locator(mydomain,[250.,250.])
```

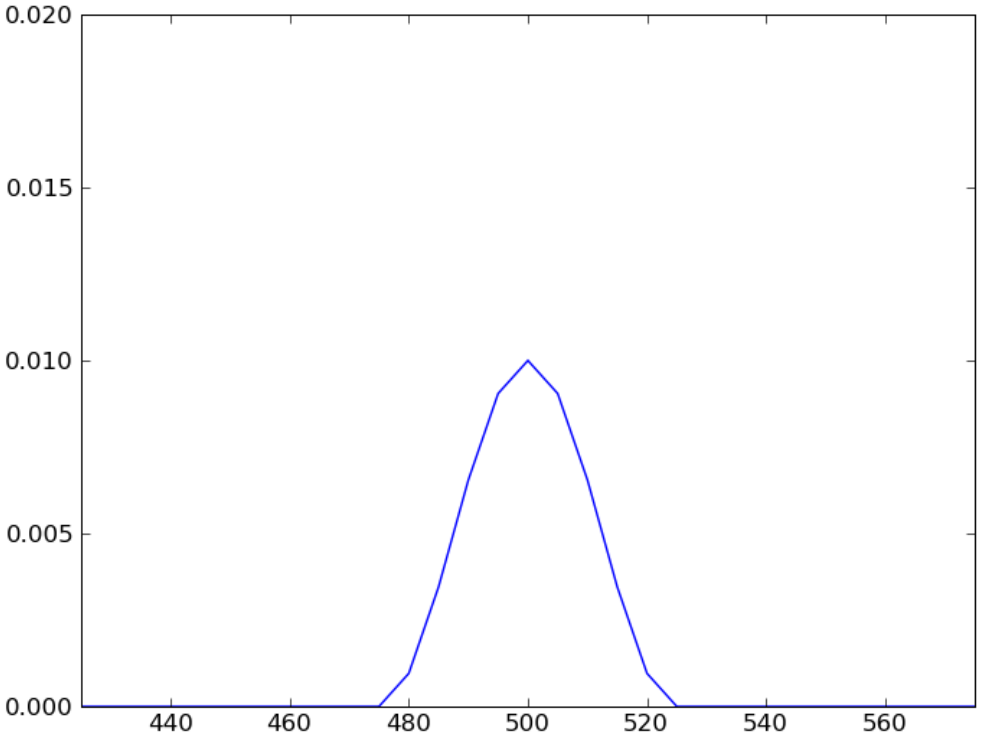


Figure 5.1: Cross section of the source function.

When the solution `u` is updated we can extract the value at that point via;

```
u_rec=rec.getValue(u)
```

For consecutive time steps one can record the values from `u_rec` in an array initialised as `u_rec0=[]` with;

```
u_rec0.append(rec.getValue(u))
```

It can be useful to monitor the value at a single or multiple individual points in the model during the modelling process. This is done using the `Locator` function.

5.4 Acceleration Solution

The scripts referenced in this section are; example07b.py

An alternative method to the displacement solution, is to solve for the acceleration $\frac{\partial^2 p}{\partial t^2}$ directly. The displacement can then be derived from the acceleration after a solution has been calculated The acceleration is given by a

modified form of Equation 5.10;

$$\nabla^2 p - \frac{1}{c^2} a = 0 \tag{5.12}$$

and can be solved directly with $Y = 0$ and $X = -c^2 \nabla^2 p_{(t)}$. After each iteration the displacement is re-evaluated via;

$$p_{(t+1)} = 2p_{(t)} - p_{(t-1)} + h^2 a \tag{5.13}$$

5.4.1 Lumping

For *escript*, the acceleration solution is prefered as it allows the use of matrix lumping. Lumping or mass lumping as it is sometimes known, is the process of aggressively approximating the density elements of a mass matrix into the main diagonal. The use of Lumping is motivaed by the simplicity of diagonal matrix inversion. As a result, Lumping can significantly reduce the computational requirements of a problem. Care should be taken however, as this function can only be used when the A, B and C coefficients of the general form are zero.

More information about the lumping implementation used in *escript* and its accuracy can be found in the user guide.

To turn lumping on in *escript* one can use the command;

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

It is also possible to check if lumping is set using;

```
print mypde.isUsingLumping()
```

5.5 Stability Investigation

It is now prudent to investigate the stability limitations of this problem. First, we let the frequency content of the source be very small. If we define the source as a cosine input, then the wavlength of the input is equal to the radius of the source. Let this value be 5 meters. Now, if the maximum velocity of the model is $c = 380.0ms^{-1}$, then the source frequency is $f_r = \frac{380.0}{5} = 76.0Hz$. This is a worst case scenario with a small source and the models maximum velocity.

Furthermore, we know from section 5.2, that the spatial sampling frequency must be at least twice this value to ensure stability. If we assume the model mesh is a square equispaced grid, then the sampling interval is the side length divided by the number of samples, given by $\Delta x = \frac{1000.0m}{400} = 2.5m$ and the maximum sampling frequency capable at this interval is $f_s = \frac{380.0ms^{-1}}{2.5m} = 152Hz$ this is just equal to the required rate satisfying Equation 5.8.

Figure 5.2 depicts three examples where the grid has been undersampled, sampled correctly, and over sampled. The grids used had 200, 400 and 800 nodes per side respectively. Obviously, the oversampled grid retains the best resolution of the modelled wave.

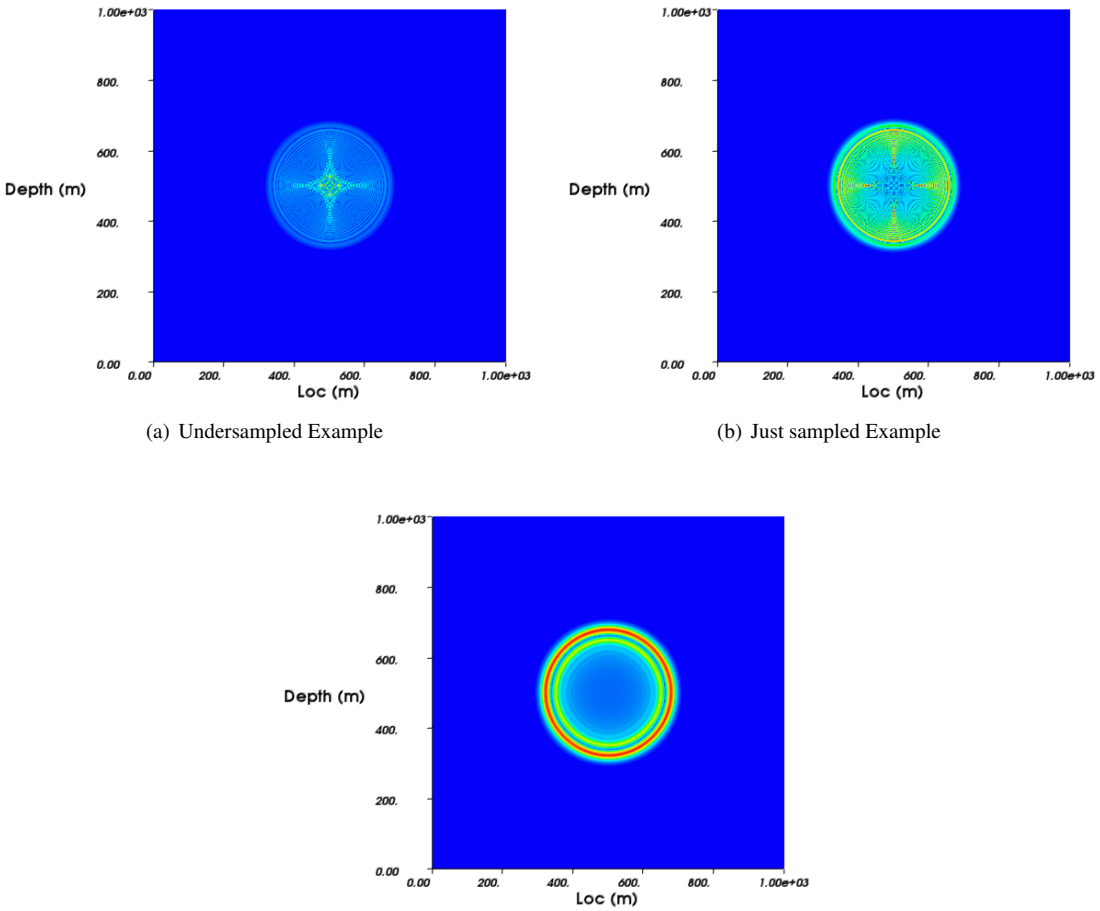
The time step required for each of these examples is simply calculated from the propagation requirement. For a maximum velocity of $380.0ms^{-1}$,

$$\Delta t \leq \frac{1000.0m}{200}\frac{1}{380.0} = 0.013s \tag{5.14a}$$

$$\Delta t \leq \frac{1000.0m}{400}\frac{1}{380.0} = 0.0065s \tag{5.14b}$$

$$\Delta t \leq \frac{1000.0m}{800}\frac{1}{380.0} = 0.0032s \tag{5.14c}$$

Observe that for each doubling of the number of nodes in the mesh, we halve the time step. To illustrate the impact this has, consider our model. If the source is placed at the center, it is $500m$ from the nearest boundary. With a velocity of $380.0ms^{-1}$ it will take $\approx 1.3s$ for the wavefront to reach that boundary. In each case, this equates to 100, 200 and 400 time steps. This is again, only a best case scenario, for true stability these time values may need to be halved and possibly halved again.



(a) Undersampled Example

(b) Just sampled Example

(c) Over sampled Example

Figure 5.2: Sampling Theorem example for stability investigation

CHAPTER

SIX

Seismic Wave Propagation

6.1 Seismic Wave Propagation in Two Dimensions

The scripts referenced in this section are; example08a.py

We will now expand upon the previous chapter by introducing a vector form of the wave equation. This means that the waves will have not only a scalar magnitude as for the pressure wave solution, but also a direction. This type of scenario is apparent in wave types that exhibit compressional and transverse particle motion. An example of this would be seismic waves.

Wave propagation in the earth can be described by the elastic wave equation

$$\rho \frac{\partial^2 u_i}{\partial t^2} - \frac{\partial \sigma_{ij}}{\partial x_j} = 0 \tag{6.1}$$

where σ is the stress given by

$$\sigma_{ij} = \lambda u_{k,k} \delta_{ij} + \mu(u_{i,j} + u_{j,i}) \tag{6.2}$$

and λ and μ represent Lame's parameters. Specifically for seismic waves, μ is the propagation materials shear modulus. In a similar process to the previous chapter, we will use the acceleration solution to solve this PDE. By substituting a directly for $\frac{\partial^2 u_i}{\partial t^2}$ we can derive the acceleration solution. Using a we can see that Equation 6.1 becomes

$$\rho a_i - \frac{\partial \sigma_{ij}}{\partial x_j} = 0 \tag{6.3}$$

Thus the problem will be solved for acceleration and then converted to displacement using the backwards difference

approximation as for the acoustic example in the previous chapter.

Consider now the stress σ. One can see that the stress consists of two distinct terms:

$$\lambda u_{k,k}\delta_{ij} \tag{6.4a}$$

$$\mu(u_{i,j} + u_{j,i}) \tag{6.4b}$$

One simply recognizes in Equation 6.4a that $u_{k,k}$ is the trace of the displacement solution and that δ_{ij} is the kronecker delta function with dimensions equivalent to u. The second term Equation 6.4b is the sum of u with its own transpose. Putting these facts together we see that the spatial differential of the stress is given by the gradient of u and the aforementioned operations. This value is then submitted to the *escript* PDE as X.

```
g=grad(u); stress=lam*trace(g)*kmat+mu*(g+transpose(g))
mypde.setValue(X=-stress) # set PDE values
```

The solution is then obtained via the usual method and the displacement is calculated so that the memory variables can be updated for the next time iteration.

```
accel = mypde.getSolution() #get PDE solution for acceleration
u_p1=(2.*u-u_m1)+h*h*accel #calculate displacement
u_m1=u; u=u_p1 # shift values by 1
```

Saving the data has been handled slightly differently in this example. The VTK files generated can be quite large and take a significant amount of time to save to the hard disk. To avoid doing this at every iteration a test is devised which saves only at specific time intervals.

To do this there are two new parameters in our script.

```
# data recording times
rtime=0.0 # first time to record
rtime_inc=tend/20.0 # time increment to record
```

Currently the PDE solution will be saved to file 20 times between the start of the modelling and the final time step. With these parameters set, an if statement is introduced to the time loop

```
if (t >= rtime):
        saveVTK(os.path.join(savepath,"ex08a.%05d.vtu"%n),displacement=length(u),\
        acceleration=length(accel),tensor=stress)
        rtime=rtime+rtime_inc #increment data save time
```

`t` is the time counter. Whenever the recording time `rtime` is less then `t` the solution is saved and `rtime` is incremented. This limits the number of outputs and increases the speed of the solver.

6.2 Multi-threading

The wave equation solution can be quite demanding on CPU time. Enhancements can be made by accessing multiple threads or cores on your computer. This does not require any modification to the solution script and only comes into play when *escript* is called from the shell. To use multiple threads *escript* is called using the `-t` option with an integer argument for the number of threads required. For example

```
$escript -t 4 example08a.py
```

would call the script in this section and solve it using 4 threads.

The computation times on an increasing number of cores is outlined in Table 6.1.

Table 6.1: Computation times for an increasing number of cores.

Number of Cores	Time (s)
1	691.0
2	400.0
3	305.0
4	328.0
5	323.0
6	292.0
7	282.0
8	445.0

6.3 Vector source on the boundary

The scripts referenced in this section are; example08b.py

For this particular example, we will introduce the source by applying a displacement to the boundary during the initial time steps. The source will again be a radially propagating wave but due to the vector nature of the PDE used, a direction will need to be applied to the source.

The first step is to choose an amplitude and create the source as in the previous chapter.

```
U0=0.01 # amplitude of point source
# will introduce a spherical source at middle left of bottom face
xc=[ndx/2,0]

############################################FIRST TIME STEPS AND SOURCE
# define small radius around point xc
src_length = 40; print "src_length = ",src_length
# set initial values for first two time steps with source terms
```

```
xb=FunctionOnBoundary(domain).getX()
y=source[0]*(cos(length(x-xc)*3.1415/src_length)+1)*\
                whereNegative(length(xb-src_length))
src_dir=numpy.array([0.,1.]) # defines direction of point source as down
y=y*src_dir
```

where `xc` is the source point on the boundary of the model. Note that because the source is specifically located on the boundary, we have used the `FunctionOnBoundary` call to ensure the nodes used to define the source are also located upon the boundary. These boundary nodes are passed to source as `xb`. The source direction is then defined as an (x, y) array and multiplied by the source function. The directional array must have a magnitude of $|1|$ otherwise the amplitude of the source will become modified. For this example, the source is directed in the $-y$ direction.

```
src_dir=numpy.array([0.,-1.]) # defines direction of point source as down
y=y*src_dir
```

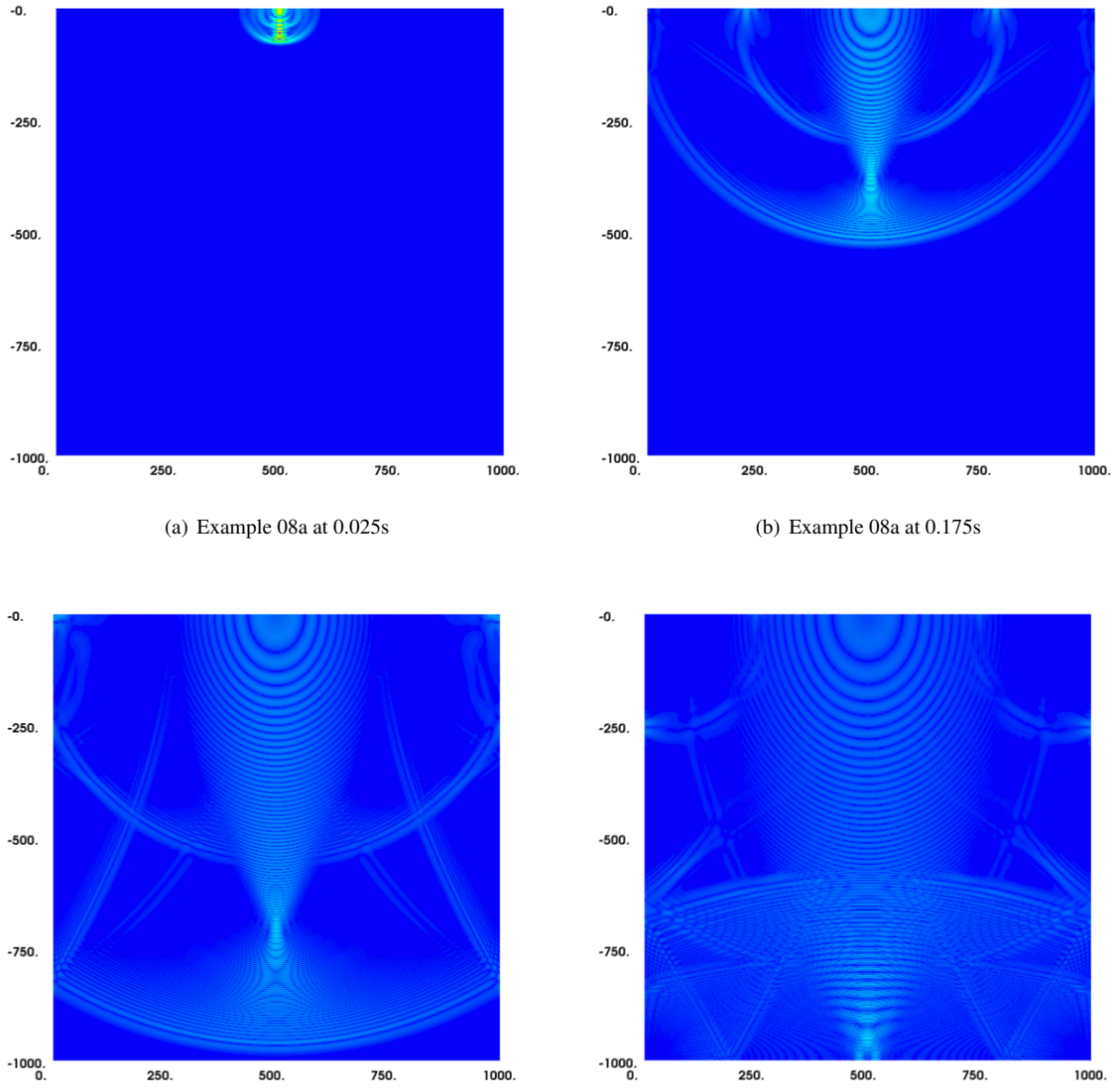
The function can then be applied as a boundary condition by setting it equal to y in the general form.

```
mypde.setValue(y=y) #set the source as a function on the boundary
```

The final step is to qualify the initial conditions. Due to the fact that we are no longer using the source to define our initial condition to the model, we must set the model state to zero for the first two time steps.

```
# initial value of displacement at point source is constant (U0=0.01)
# for first two time steps
u=[0.0,0.0]*wherePositive(x)
u_m1=u
```

If the source is time progressive, y can be updated during the iteration stage. This is covered in the following section.



(a) Example 08a at 0.025s

(b) Example 08a at 0.175s

(c) Example 08a at 0.325s

(d) Example 08a at 0.475s

Figure 6.1: Results of Example 08 at various times.

6.4 Time variant source

The scripts referenced in this section are; example08b.py

Until this point, all of the wave propagation examples in this cookbook have used impulsive sources which are smooth in space but not time. It is however, advantageous to have a time smoothed source as it can reduce the temporal frequency range and thus mitigate aliasing in the solution.

It is quite simple to implement a source which is smooth in time. In addition to the original source function the only extra requirement is a time function. For this example the time variant source will be the derivative of a Gaussian curve defined by the required dominant frequency (Figure 6.2).

```
#Creating the time function of the source.
dfeq=50 #Dominant Frequency
a = 2.0 * (np.pi * dfeq)**2.0
t0 = 5.0 / (2.0 * np.pi * dfeq)
srclength = 5. * t0
ls = int(srclength/h)
print 'source length',ls
source=np.zeros(ls,'float') # source array
ampmax=0
for it in range(0,ls):
    t = it*h
    tt = t-t0
    dum1 = np.exp(-a * tt * tt)
    source[it] = -2. * a * tt * dum1
    if (abs(source[it]) > ampmax):
        ampmax = abs(source[it])
    time[t]=t*h
```

We then build the source and the first two time steps via;

```
# set initial values for first two time steps with source terms
y=source[0]
*(cos(length(x-xc)*3.1415/src_length)+1)*whereNegative(length(x-xc)-src_length)
src_dir=numpy.array([0.,-1.]) # defines direction of point source as down
y=y*src_dir
mypde.setValue(y=y) #set the source as a function on the boundary
# initial value of displacement at point source is constant (U0=0.01)
# for first two time steps
```

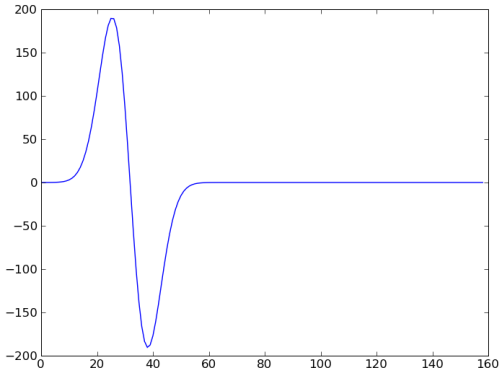


Figure 6.2: Time variant source with a dominant frequency of 50Hz.

```
u=[0.0,0.0]*whereNegative(x)
u_m1=u
```

Finally, for the length of the source, we are required to update each new solution in the iterative section of the solver. This is done via;

```
# increment loop values
t=t+h; n=n+1
if (n < ls):
        y=source[n]**(cos(length(x-xc)*3.1415/src_length)+1)*\
                  whereNegative(length(x-xc)-src_length)
        y=y*src_dir; mypde.setValue(y=y) #set the source as a function on the
boundary
```

6.5 Absorbing Boundary Conditions

To mitigate the effect of the boundary on the model, absorbing boundary conditions can be introduced. These conditions effectively dampen the wave energy as they approach the boundary and thus prevent that energy from being reflected. This type of approach is typically used when a model is shrunk to decrease computational requirements. In practise this applies to almost all models, especially in earth sciences where the entire planet or a large enough portional of it cannot be modelled efficiently when considering small scale problems. It is impractical to calculate the solution for an infinite model and thus ABCs allow us the create an approximate solution with small to zero boundary effects on a model with a solvable size.

To dampen the waves, the method of Cerjan [2] where the solution and the stress are multiplied by a damping

function defined on n nodes of the domain adjacent to the boundary, given by;

$$\gamma = \sqrt{\frac{|-log(\gamma_b)|}{n^2}} \tag{6.5}$$

$$y = e^{-(\gamma x)^2} \tag{6.6}$$

This is applied to the bounding 20-50 pts of the model using the location specifiers of *escript*;

```
# Define where the boundary decay will be applied.
bn=30.
bleft=xstep*bn; bright=mx-(xstep*bn); bbot=my-(ystep*bn)
# btop=ystep*bn # don't apply to force boundary!!!

# locate these points in the domain
left=x[0]-bleft; right=x[0]-bright; bottom=x[1]-bbot

tgamma=0.98   # decay value for exponential function
def calc_gamma(G,npts):
    func=np.sqrt(abs(-1.*np.log(G)/(npts**2.)))
    return func

gleft  = calc_gamma(tgamma,bleft)
gright = calc_gamma(tgamma,bleft)
gbottom= calc_gamma(tgamma,ystep*bn)

print 'gamma', gleft,gright,gbottom

# calculate decay functions
def abc_bfunc(gamma,loc,x,G):
    func=exp(-1.*(gamma*abs(loc-x))**2.)
    return func

fleft=abc_bfunc(gleft,bleft,x[0],tgamma)
fright=abc_bfunc(gright,bright,x[0],tgamma)
fbottom=abc_bfunc(gbottom,bbot,x[1],tgamma)
# apply these functions only where relevant
abcleft=fleft*whereNegative(left)
abcright=fright*wherePositive(right)
```

```
abcbottom=fbottom*wherePositive(bottom)
# make sure the inside of the abc is value 1
abcleft=abcleft+whereZero(abcleft)
abcright=abcright+whereZero(abcright)
abcbottom=abcbottom+whereZero(abcbottom)
# multiply the conditions together to get a smooth result
abc=abcleft*abcright*abcbottom
```

Note that the boundary conditions are not applied to the surface, as this is effectively a free surface where normal reflections would be experienced. Special conditions can be introduced at this surface if they are known. The resulting boundary damping function can be viewed in 6.6.

6.6 Second order Meshing

For stiff problems like the wave equation it is often prudent to implement second order meshing. This creates a more accurate mesh approximation with some increased processing cost. To turn second order meshing on, the `rectangle` function accepts an `order` keyword argument.

```
domain=Rectangle(l0=mx,l1=my,n0=ndx, n1=ndy,order=2) # create the domain
```

Other pycad functions and objects have similar keyword arguments for higher order meshing.

Note that when implementing second order meshing, a smaller timestep is required then for first order meshes as the second order essentially reduces the size of the mesh by half.

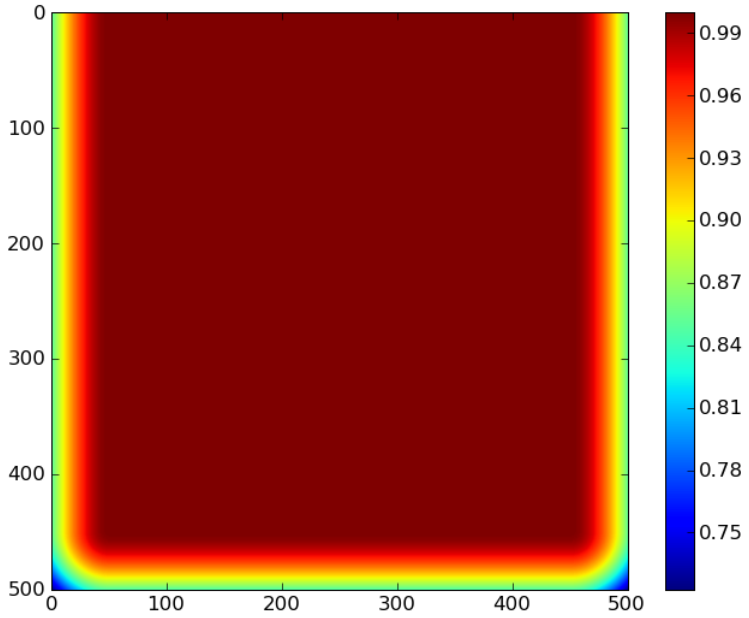
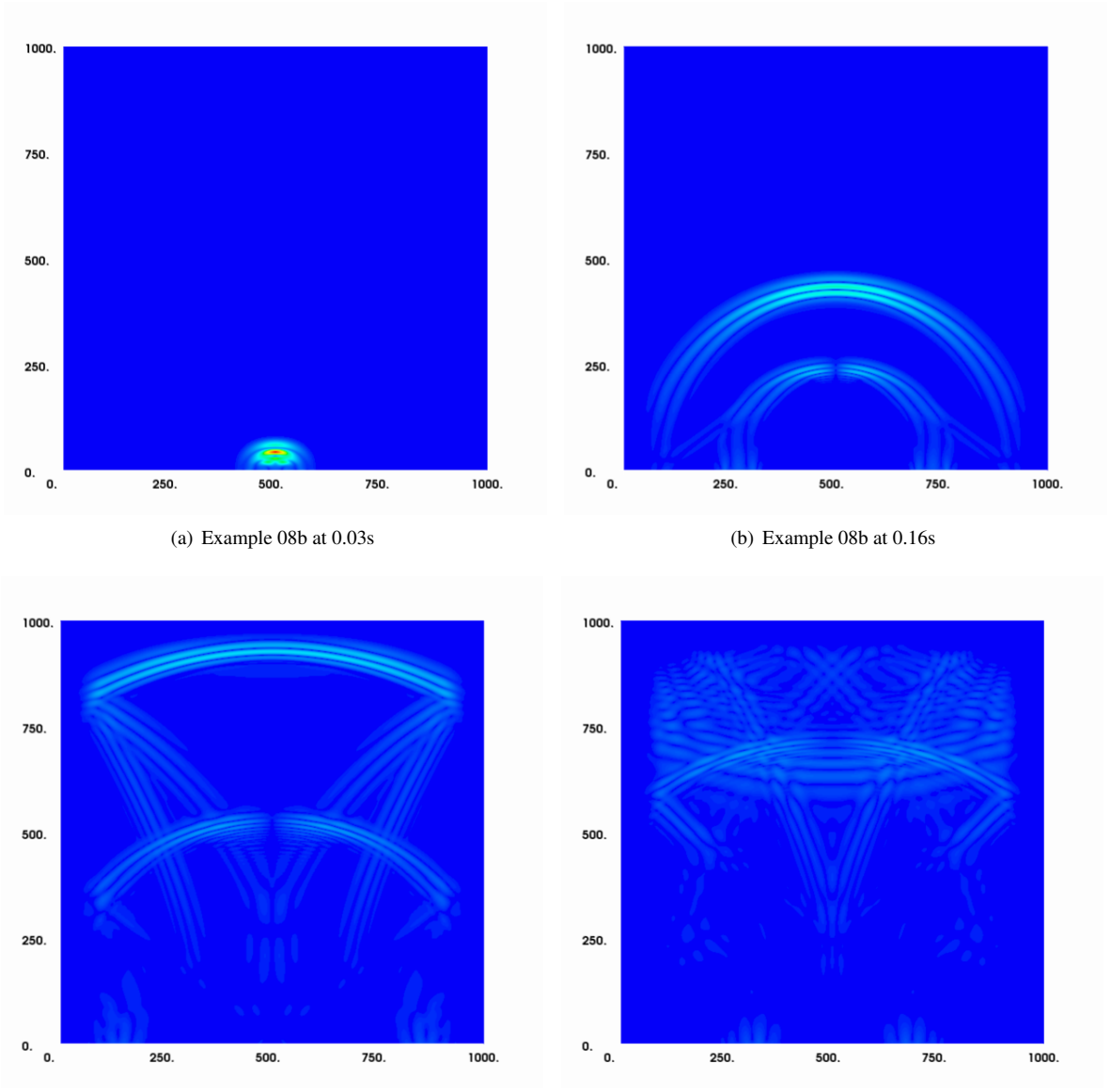


Figure 6.3: Absorbing boundary conditions for example08b.py



(a) Example 08b at 0.03s

(b) Example 08b at 0.16s

(c) Example 08b at 0.33s

(d) Example 08b at 0.44s

Figure 6.4: Results of Example 08b at various times.

6.7 Pycad example

The scripts referenced in this section are; example08c.py

To make the problem more interesting we will now introduce an interface to the middle of the domain. In fact we will use the same domain as we did fora different set of material properties on either side of the interface.

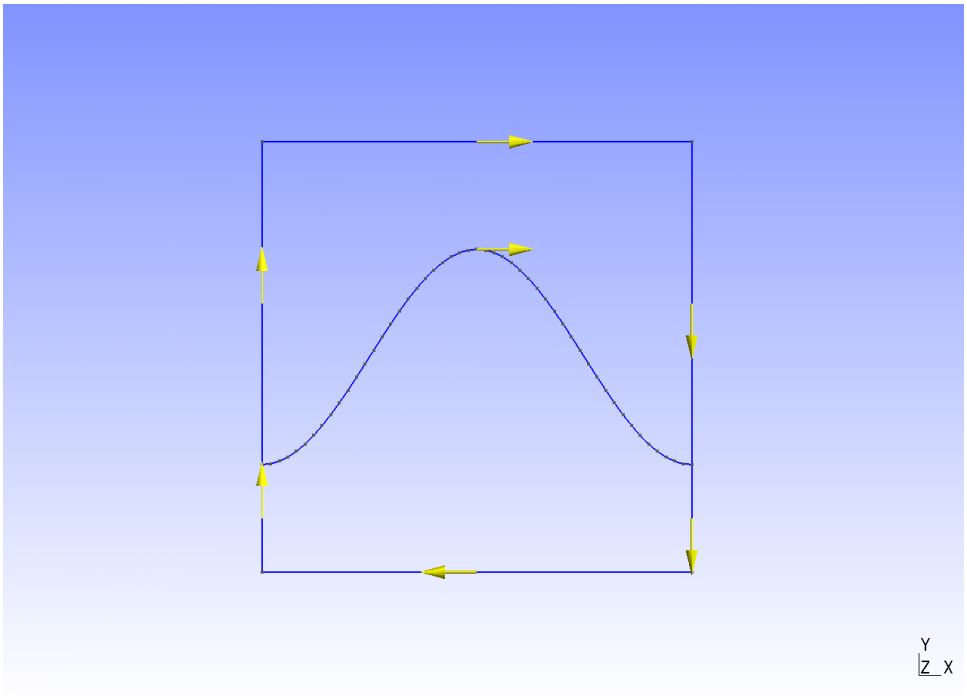


Figure 6.5: Domain geometry for example08c.py showing line tangents.

It is simple enough to slightly modify the scripts of the previous sections to accept this domain. Multiple material parameters must now be deined and assigned to specific tagged areas. Again this is done via

```
lam=Scalar(0,Function(domain))
lam.setTaggedValue("top",lam1)
lam.setTaggedValue("bottom",lam2)
mu=Scalar(0,Function(domain))
mu.setTaggedValue("top",mu1)
mu.setTaggedValue("bottom",mu2)
rho=Scalar(0,Function(domain))
rho.setTaggedValue("top",rho1)
rho.setTaggedValue("bottom",rho2)
```

Don't forget that the source boundary must also be tagged and added so it can be referenced

```
# Add the subdomains and flux boundaries.
```

```
d.addItems(PropertySet("top",tblock),PropertySet("bottom",bblock),\
                                        PropertySet("linetop",l30))
```

It is now possible to solve the script as in the previous examples (Figure 6.6).

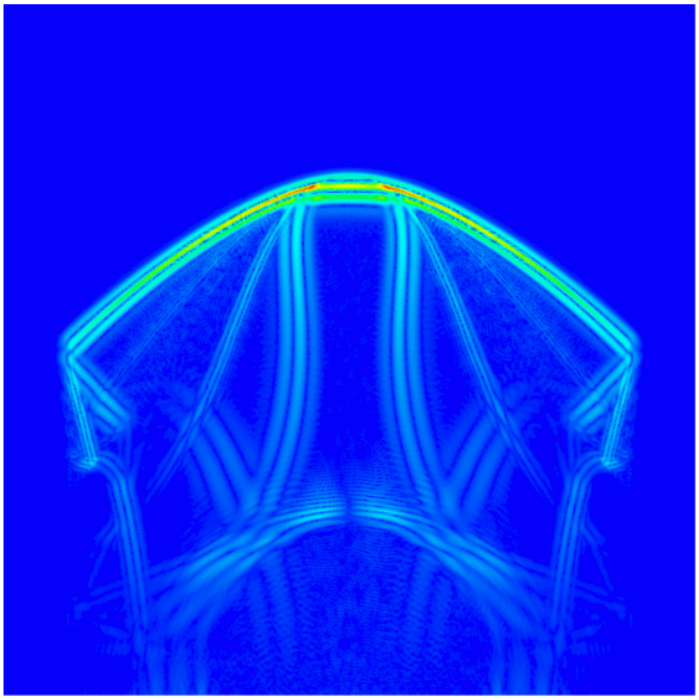


Figure 6.6: Modelling results of example08c.py at 0.2601 seconds. Notice the refraction of the wave front about the boundary between the two layers.

CHAPTER

SEVEN

3D Seismic Wave Propagation

7.1 3D pycad

The scripts referenced in this section are; example09m.py

This example explains how to build a 3D layered domain using pycad. As simple as this example sounds, there are a few import concepts that one must remember so that the model will function correctly.

- There must be no duplication of any geometric features whether they are points, lines, loops, surfaces or volumes.
- All objects with dimensions greater then a line have a normal defined by the right hand rule (RHR). It is important to consider which direction a normal is oriented when combining primitives to form higher order shapes.

The first step as always is to import the external modules. To build a 3D model and mesh we will need pycad, some GMesh interfaces, the Finley domain builder and some additional tools.

```
##########################################################EXTERNAL MODULES
from esys.pycad import * #domain constructor
from esys.pycad.gmsh import Design #Finite Element meshing package
from esys.finley import MakeDomain #Converter for escript
from esys.escript import mkDir, getMPISizeWorld
import os
```

After carrying out some routine checks and setting the `save_path` we then specify the parameters of the model. This model will be 2000 by 2000 meters on the surface and extend to a depth of 500 meters. An interface or boundary between two layers will be created at half the total depth or 250 meters. This type of model is known as a horizontally layered model or a layer cake model.

```
##################################################ESTABLISHING PARAMETERS
#Model Parameters
xwidth=2000.0*m   #x width of model
ywidth=2000.0*m   #y width of model
depth=500.0*m   #depth of model
intf=depth/2.   #Depth of the interface.
```

We now start to specify the components of our model starting with the vertexes using the `Point` primitive. These are then joined by lines in a regular manner taking note of the right hand rule. Finally, the lines are turned into loops and then planar surfaces. [1]

```
##################################################DOMAIN CONSTRUCTION
# Domain Corners
p0=Point(0.0,     0.0,       0.0)
#..etc..
l45=Line(p4, p5)


# Join line segments to create domain boundaries and then surfaces
ctop=CurveLoop(l01, l12, l23, l30);     stop=PlaneSurface(ctop)
cbot=CurveLoop(-l67, -l56, -l45, -l74); sbot=PlaneSurface(cbot)
```

With the top and bottom of the domain taken care of, it is now time to focus on the interface. Again the vertexes of the planar interface are created. With these, vertical and horizontal lines (edges) are created joining the interface with itself and the top and bottom surfaces.

```
# for each side
ip0=Point(0.0,     0.0,       intf)
#..etc..
linte_ar=[]; #lines for vertical edges
linhe_ar=[]; #lines for horizontal edges
linte_ar.append(Line(p0,ip0))
#..etc..
linhe_ar.append(Line(ip3,ip0))
```

[1] Some code has been emmitted here for simlpicity. For the full script please refer to the script referenced at the beginning of this section.

Consider now the sides of the domain. One could specify the whole side using the points first defined for the top and bottom layer. This would define the whole domain as one volume. However, there is an interface and we wish to define each layer individually. Therefore, there will be 8 surfaces on the sides of our domain. We can do this operation quite simply using the points and lines that we had defined previously. First loops are created and then surfaces making sure to keep a normal for each layer which is consistent with upper and lower surfaces of the layer. For example, all surface normals must face outwards from or inwards towards the centre of the volume.

```
cintfa_ar=[]; cintfb_ar=[] #curveloops for above and below interface on sides
cintfa_ar.append(CurveLoop(linte_ar[0],linhe_ar[0],-linte_ar[2],-l01))
#..etc..
cintfb_ar.append(CurveLoop(linte_ar[7],l45,-linte_ar[1],-linhe_ar[3]))

sintfa_ar=[PlaneSurface(cintfa_ar[i]) for i in range(0,4)]
sintfb_ar=[PlaneSurface(cintfb_ar[i]) for i in range(0,4)]

sintf=PlaneSurface(CurveLoop(*tuple(linhe_ar)))
```

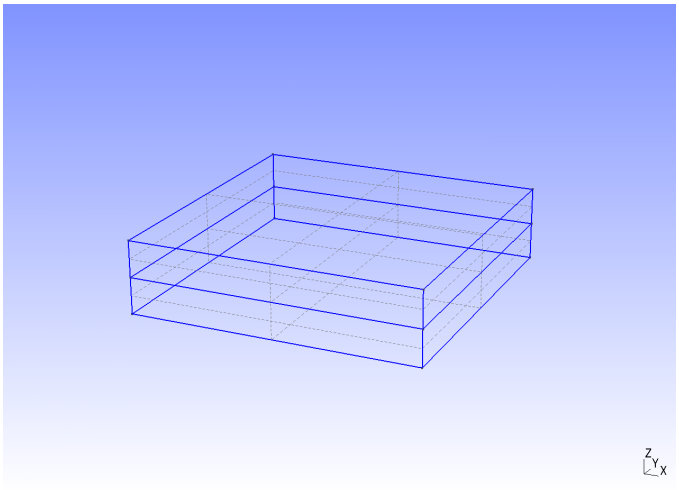
Assuming all is well with the normals, the volumes can be created from our surface arrays. Note the use here of the `*tuple` function. This allows us to pass an list array as an argument list to a function. It must be placed at the end of the function arguments and there cannot be more than one per function call.

```
vintfa=Volume(SurfaceLoop(stop,-sintf,*tuple(sintfa_ar)))
vintfb=Volume(SurfaceLoop(sbot,sintf,*tuple(sintfb_ar)))

# Create the volume.
#sloop=SurfaceLoop(stop,sbot,*tuple(sintfa_ar+sintfb_ar))
#model=Volume(sloop)
```

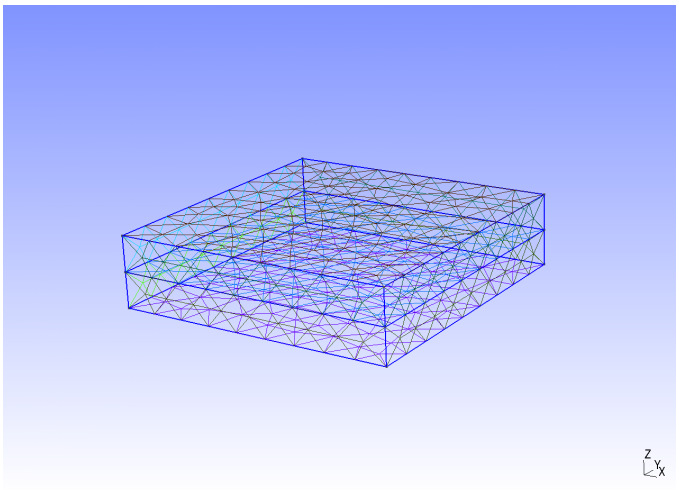
The final steps are designing the mesh, tagging the volumes and the interface and outputting the data to file so it can be imported by an *escript* solution script.

```
################################################EXPORTING MESH FOR ESCRIPT
# Create a Design which can make the mesh
d=Design(dim=3, element_size=5.0*m)
d.addItems(PropertySet('vintfa',vintfa))
d.addItems(PropertySet('vintfb',vintfb))
d.addItems(sintf)

d.setScriptFileName(os.path.join(save_path,"example09m.geo"))
```

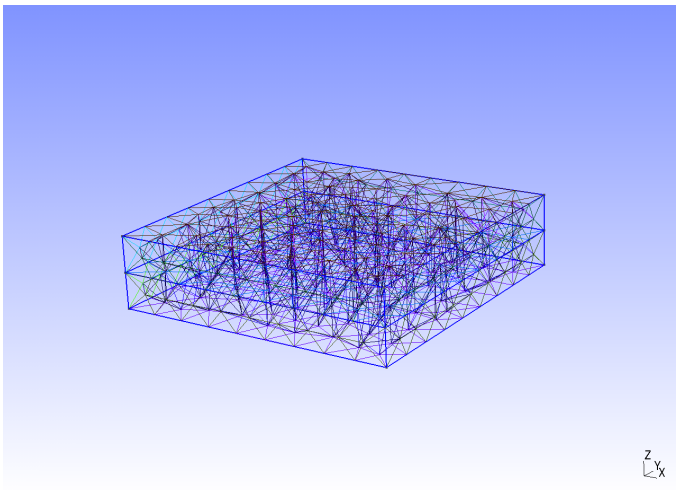
```
d.setMeshFileName(os.path.join(save_path,"example09m.msh"))
#
#  make the finley domain:
#
domain=MakeDomain(d)
# Create a file that can be read back in to python with
# mesh=ReadMesh(fileName)
domain.write(os.path.join(save_path,"example09m.fly"))
```



(a) Gmesh view of geometry only.



(b) Gmesh view of a 200m 2D mesh on the domain surfaces.



(c) Gmesh view of a 200m 3D mesh on the domain volumes.

7.2 Layer Cake Models

Whilst this type of model seems simple enough to construct for two layers, specifying multiple layers can become cumbersome. A function exists to generate layer cake models called `layer_cake`. A detailed description of its arguments and returns is available in the API and the function can be imported from the pycad.extras module.

```
from esys.pycad.extras import layer_cake
intfaces=[10,30,50,55,80,100,200,250,400,500]

domaindes=Design(dim=3,element_size=element_size,order=2)
cmplx_domain=layer_cake(domaindes,xwidth,ywidth,intfaces)
cmplx_domain.setScriptFileName(os.path.join(save_path,"example09lc.geo"))
cmplx_domain.setMeshFileName(os.path.join(save_path,"example09lc.msh"))
dcmplx=MakeDomain(cmplx_domain)
dcmplx.write(os.path.join(save_path,"example09lc.fly"))
```

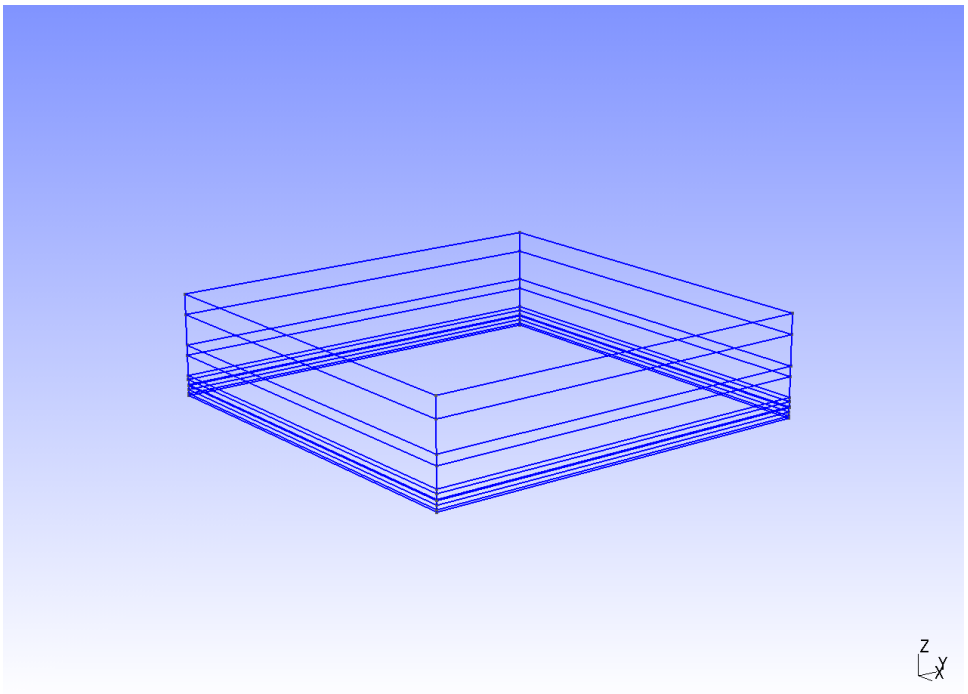


Figure 7.1: Example geometry using layer cake function.

7.3 Troubleshooting Pycad

There are some techniques which can be useful when trying to troubleshoot problems with pycad. As mentioned earlier it is important to ensure the correct directionality of your primitives when constructing more complicated domains. If it remains too difficult to establish the tangent of a line or curveloop, or the normal of a surface, these values can be checked by importing the geometry to gmesh and applying the appropriate visualisation options.

7.4 3D Seismic Wave Propagation

Adding an extra dimension to our wave equation solution script should be relatively simple. Apart from the changes to the domain and therefore the parameters of the model, there are only a few minor things which must be modified to make the solution appropriate for 3d modelling.

7.5 Applying a function to a domain tag

The scripts referenced in this section are; example09a.py

To apply a function or data object to a domain requires a two step process. The first step is to create a data object with an on/off mask based upon the tagged value. This is quite simple and can be achieved by using a scalar data object based upon the domain. In this case we are using the `FunctionOnBoundary` function space because the tagged value `'stop'` is effectively a specific subsurface of the boundary of the whole domain.

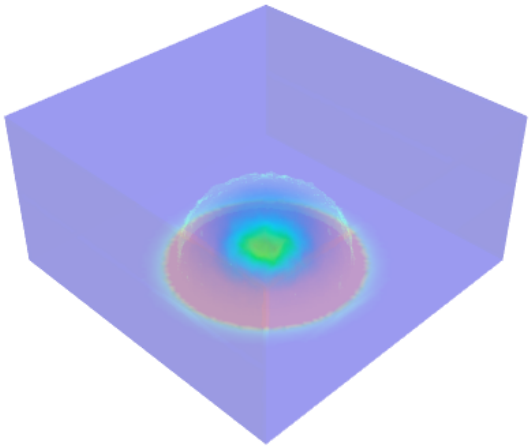
```
stop=Scalar(0.0,FunctionOnBoundary(domain))
stop.setTaggedValue("stop",1.0)
```

Now the data object `stop` has the value 1.0 on the surface `'stop'` and zero everywhere else. To apply our function to the boundary only on `stop` we now mulitply it by `stop`

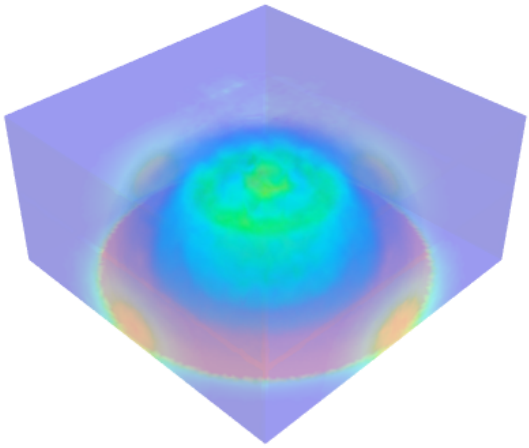
```
xb=FunctionOnBoundary(domain).getX()
yx=(cos(length(xb-xc)*3.1415/src_length)+1)*whereNegative(length(xb-xc)-src_length)
src_dir=numpy.array([0.,0.,1.0]) # defines direction of point source as down
mypde.setValue(y=source[0]*yx*src_dir*stop) #set the source as a function on the boundary
```

7.6 Mayavi2 with 3D data.

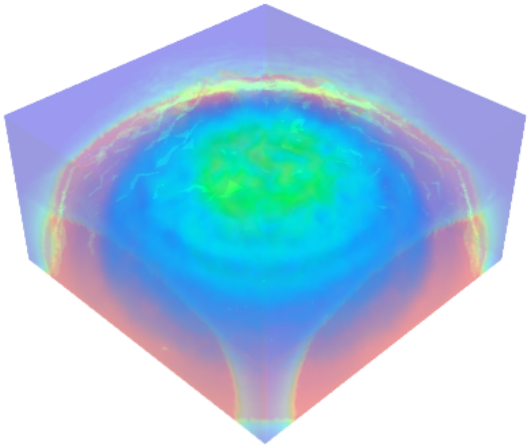
There are a variety of visualisation options available when using VTK data. The types of visualisation are often data and interpretation specific and thus consideration must be given to the type of output saved to file. Whether that is scalar, vector or tensor data.



(a) Example 9 at time step 201.



(b) Example 9 at time step 341.



(c) Example 9 at time step 440.

CHAPTER

EIGHT

Potential Fields - Newtonian Gravitation

8.1 Newtonian Potential

In this chapter the gravitational potential field is developed for *escript*. Gravitational fields are present in many modelling scenarios, including geophysical investigations, planetary motion and attraction and micro-particle interactions. Gravitational fields also present an opportunity to demonstrate the saving and visualisation of vector data for Mayavi, and the construction of variable sized meshes.

The gravitational potential U at a point P due to a region with a mass distribution of density $\rho(P)$, is given by Poisson's equation [1]

$$\nabla^2 U(P) = -4\pi\gamma\rho(P) \tag{8.1}$$

where γ is the gravitational constant. Consider now the *escript* general form, Equation 8.1 requires only two coefficients, A and Y, thus the relevant terms of the general form are reduced to;

$$-\left(A_{jl}u_{,l}\right)_{,j} = Y \tag{8.2}$$

One recognises that the LHS side is equivalent to

$$-\nabla A \nabla u \tag{8.3}$$

and when $A = \delta_{jl}$, Equation 8.3 is equivalent to

$$-\nabla^2 u$$

Thus Poisson's Equation 8.1 satisfies the general form when

$$A = \delta_{jl} \text{ and } Y = 4\pi\gamma\rho \tag{8.4}$$

The boundary condition is the last parameter that requires consideration. The potential U is related to the mass of a sphere by

$$U(P) = -\gamma\frac{m}{r^2} \tag{8.5}$$

where m is the mass of the sphere and r is the distance from the center of the mass to the observation point P. Plainly, the magnitude of the potential is governed by an inverse-square distance law. Thus, in the limit as r increases;

$$\lim_{r\to\infty} U(P) = 0 \tag{8.6}$$

Provided that the domain being solved is large enough, and the source mass is contained within a central region of the domain, the potential will decay to zero. This is a dirichlet boundary condition where $U = 0$.

This boundary condition can be satisfied when there is some mass suspended in a free-space. For geophysical models where the mass of interest may be an anomaly inside a host rock, the anomaly can be isolated by subtracting the density of the host rock from the model. This creates a fictitious free space model that will satisfy the analytic boundary conditions. The result is that $Y = 4\pi\gamma\Delta\rho$, where $\Delta\rho = \rho - \rho_0$ and ρ_0 is the baseline or host density. This of course means that the true gravity response is not being modelled, but the response due to an anomalous mass with a density contrast $\Delta\rho$.

For this example we have set all of the boundaries to zero but only one boundary point needs to be set for the problem to be solvable. The normal flux condition is also zero by default. Note, that for a more realistic and complicated models it may be necessary to give careful consideration to the boundary conditions of the model, which can have an influence upon the solution.

Setting the boundary condition is relatively simple using the `q` and `r` variables of the general form. First `q` is defined as a masking function on the boundary using

```
q=whereZero(x[1]-my)+whereZero(x[1])+whereZero(x[0])+whereZero(x[0]-mx)
mypde.setValue(q=q,r=0)
```

This identifies the points on the boundary and `r` is simply ser to `r=0.0`. This is a Dirichlet boundary condition.

8.2 Gravity Pole

The scripts referenced in this section are; example10a.py

A gravity pole is used in this example to demonstrate the vector characteristics of gravity, and also to demonstrate how this information can be exported for visualisation to Mayavi or an equivalent using the VTK data format.

The solution script for this section is very simple. First the domain is constructed, then the parameters of the model are set, and finally the steady state solution is found. There are quite a few values that can now be derived from the solution and saved to file for visualisation.

The potential U is related to the gravitational response $\vec{g}$ via

$$\vec{g} = \nabla U \tag{8.7}$$

$\vec{g}$ is a vector and thus, has a a vertical component g_z where

$$g_z = \vec{g} \cdot \hat{z} \tag{8.8}$$

Finally, there is the magnitude of the vertical component g of g_z

$$g = |g_z| \tag{8.9}$$

These values are derived from the *escript* solution `sol` to the potential U using the following commands

```
g_field=grad(sol) #The gravitational acceleration g.
g_fieldz=g_field*[0,1] #The vertical component of the g field.
gz=length(g_fieldz) #The magnitude of the vertical component.
```

This data can now be simply exported to a VTK file via

```
# Save the output to file.
saveVTK(os.path.join(save_path,"ex10a.vtu"),\
        grav_pot=sol,g_field=g_field,g_fieldz=g_fieldz,gz=gz)
```

It is quite simple to visualise the data from the gravity solution in Mayavi2. With Mayavi2 open go to File, Load data, Open file ... as in Figure 8.1 and select the saved data file. The data will have then been loaded and is ready for visualisation. Notice that under the data object in the Mayavi2 navigation tree the 4 values saved to the VTK file are available (Figure 8.2). There are two vector values, `gfield` and `gfieldz`. Note that to plot both of these on the same chart requires that the data object be imported twice.

The point scalar data `grav_pot` is the gravitational potential and it is easily plotted using a surface module.

To visualise the cell data a filter is required that converts to point data. This is done by right clicking on the data object in the explorer tree and selecting the cell to point filter as in Figure 8.3.

The settings can then be modified to suit personal taste. An example of the potential and gravitational field vectors is illustrated in Figure 8.4.

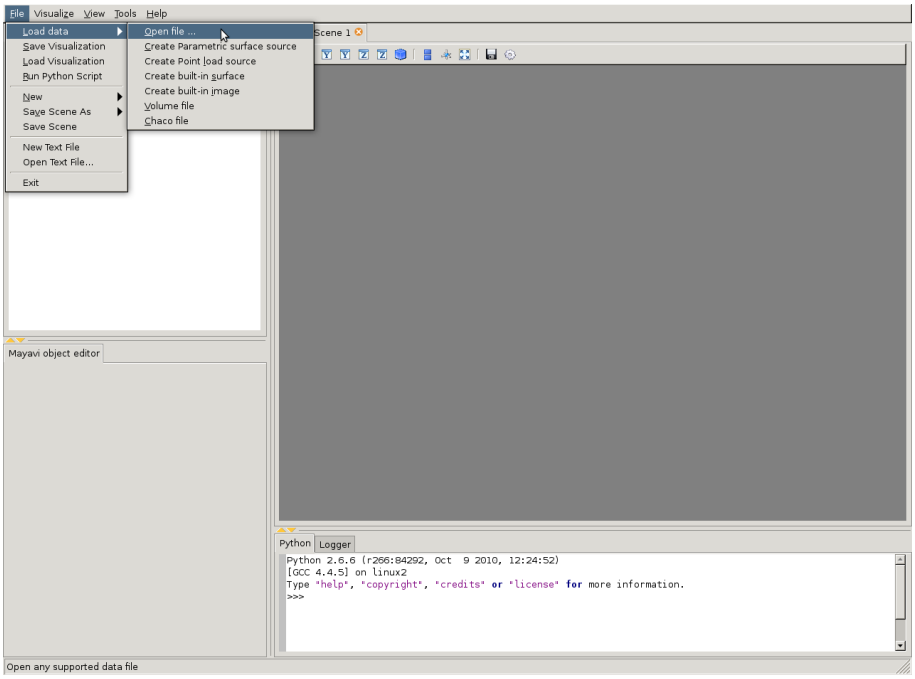


Figure 8.1: Open a file in Mayavi2

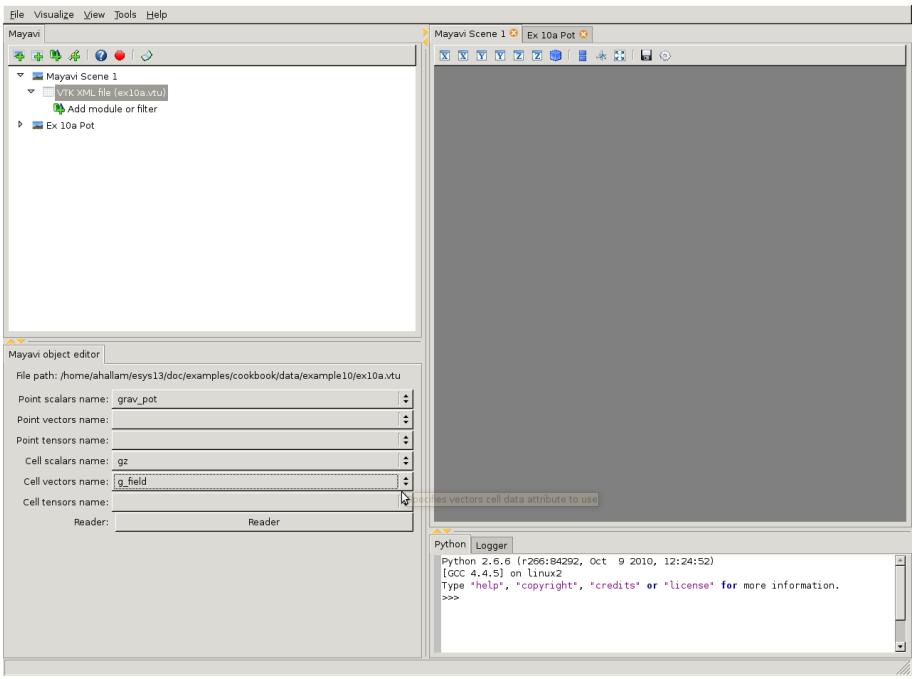


Figure 8.2: The 4 types of data in the imported VTK file.

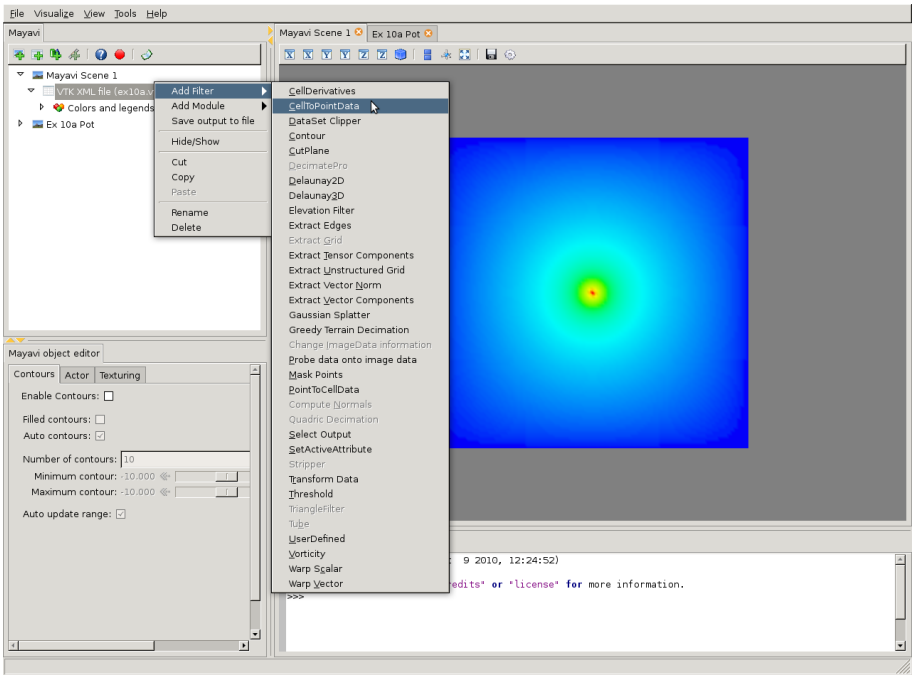


Figure 8.3: Converting cell data to point data.

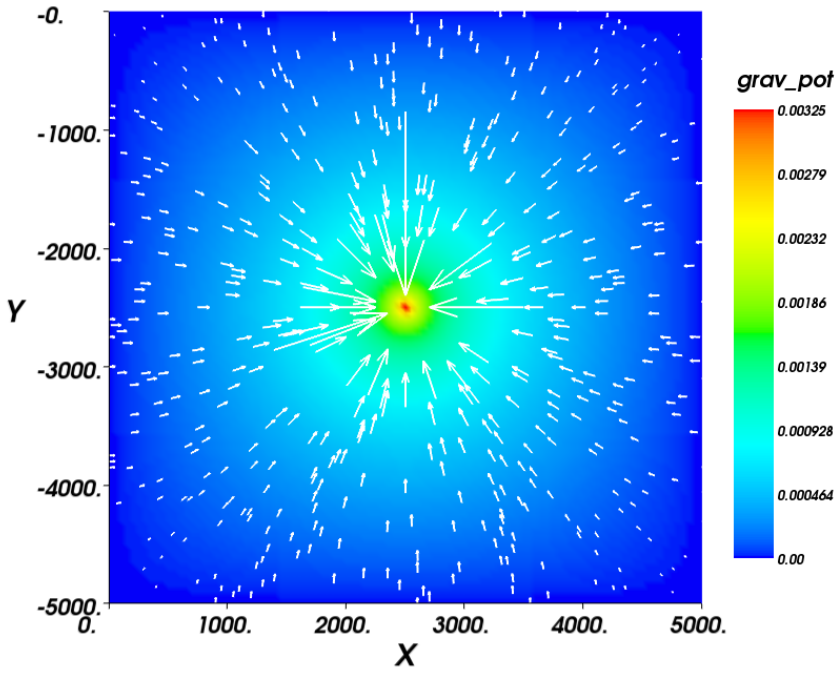


Figure 8.4: Newtonian potential with $\vec{g}$ field directionality. The magnitude of the field is reflected in the size of the vector arrows.

8.3 Gravity Well

The scripts referenced in this section are; example10b.py

Let us now investigate the effect of gravity in three dimensions. Consider a volume which contains a spherical mass anomaly and a gravitational potential which decays to zero at the base of the model.

The script used to solve this model is very similar to that used for the gravity pole in the previous section, but with an extra spatial dimension. As for all the 3D problems examined in this cookbook, the extra dimension is easily integrated into the *escript* solution script.

The domain is redefined from a rectangle to a Brick;

```
domain = Brick(l0=mx,l1=my,n0=ndx, n1=ndy,l2=mz,n2=ndz)
```

the source is relocated along z;

```
x=x-[mx/2,my/2,mz/2]
```

and, the boundary conditions are updated.

```
q=whereZero(x[2]-inf(x[2]))
```

No modifications to the PDE solution section are required. Thus the migration from a 2D to a 3D problem is almost trivial.

Figure 8.5 illustrates the strength of a PDE solution. Three different visualisation types help define and illuminate properties of the data. The cut surfaces of the potential are similar to a 2D section for a given x or y and z. The iso-surfaces illuminate the 3D shape of the gravity field, as well as its strength which is illustrated by the colour. Finally, the streamlines highlight the directional flow of the gravity field in this example.

The boundary conditions were discussed previously, but not thoroughly investigated. It was stated, that in the limit as the boundary becomes more remote from the source, the potential will reduce to zero. Figure 8.6 is the solution to the same gravity problem but with a slightly larger domain. It is obvious in this case that the previous domain size was too small to accurately represent the solution. The profiles in Figure 8.7 further demonstrates the affect the domain size has upon the solution. As the domain size increases, the profiles begin to converge. This convergence is a good indicator of an appropriately dimensioned model for the problem, and and sampling location.

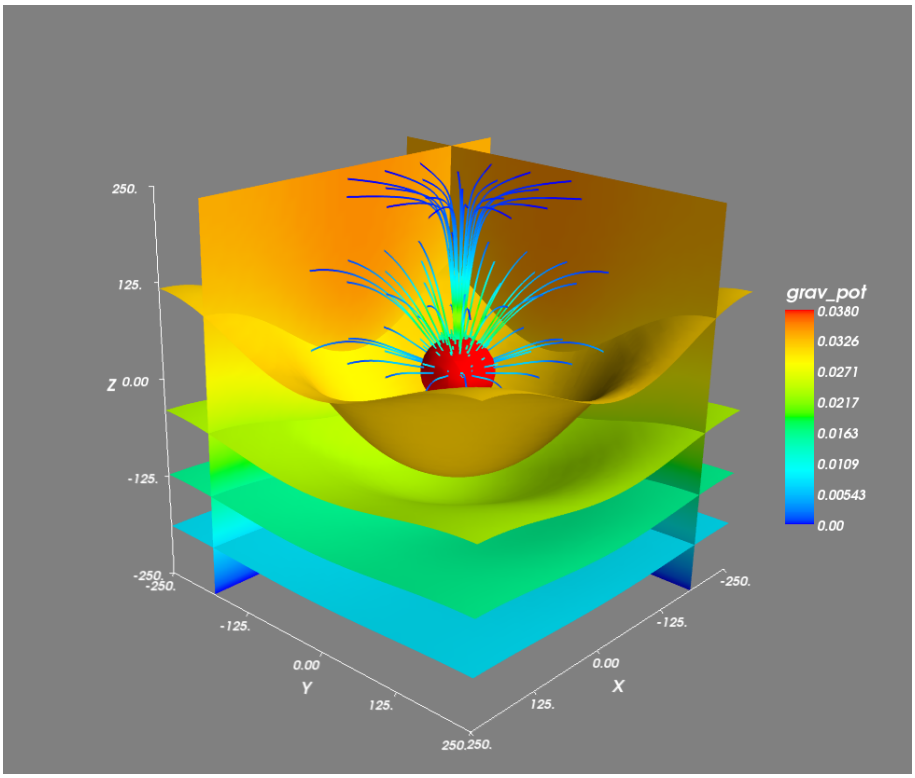


Figure 8.5: Gravity well with iso surfaces and streamlines of the vector gravitational potential —small model.

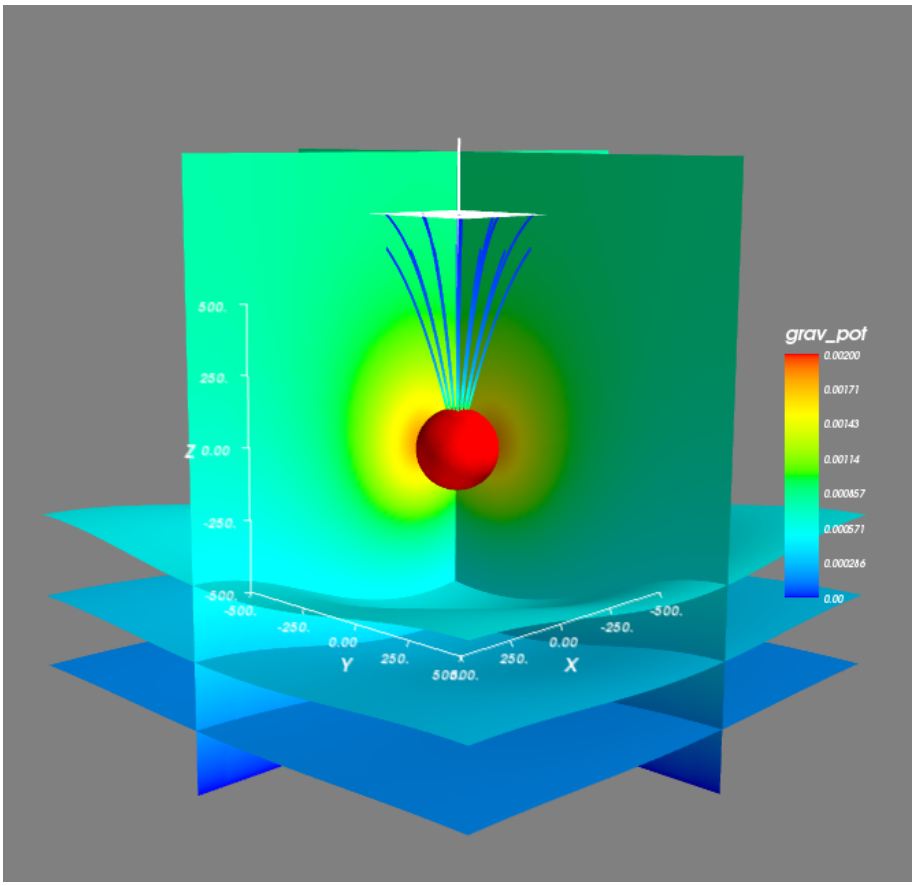


Figure 8.6: Gravity well with iso surfaces and streamlines of the vector gravitational potential —large model.

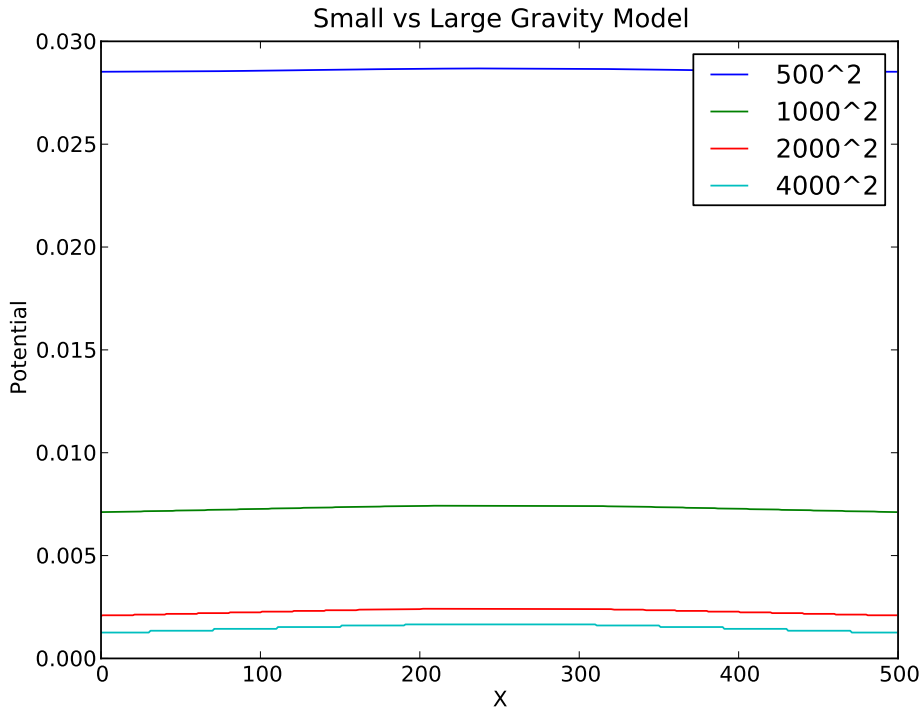


Figure 8.7: Profile of the graviational provile along x where $y = 0, z = 250$ for various sized domains.

8.4 Variable mesh-element sizes

The scripts referenced in this section are; example10m.py

We saw in a previous section that the domain needed to be sufficiently large for the boundary conditions to be satisfied. This can be troublesome when trying to solve problems that require a dense mesh, either for solution resolution of stability reasons. The computational cost of solving a large number of elements can be prohibitive.S

With the help of Gmsh, it is possible to create a mesh for *escript*, which has a variable element size. Such an approach can significantly reduce the number of elements that need to be solved, and a large domain can be created that has sufficient resolution close to the source and extends to distances large enough that the infinity clause is satisfied.

To create a variable size mesh, multiple meshing domains are required. The domains must share points, boundaries and surfaces so that they are joined; and no sub-domains are allowed to overlap. Whilst this initially seems complicated, it is quite simple to implement.

This example creates a mesh which contains a high resolution sub-domain at its center. We begin by defining two curve loops which describe the large or big sub-domain and the smaller sub-domain which is to contain the high resolution portion of the mesh.

```
##################################################BIG DOMAIN
#ESTABLISHING PARAMETERS
width=10000.    #width of model
depth=10000.    #depth of model
bele_size=500.  #big element size
#DOMAIN CONSTRUCTION
p0=Point(0.0,    0.0)
p1=Point(width, 0.0)
p2=Point(width, depth)
p3=Point(0.0,   depth)
# Join corners in anti-clockwise manner.
l01=Line(p0, p1)
l12=Line(p1, p2)
l23=Line(p2, p3)
l30=Line(p3, p0)

cbig=CurveLoop(l01,l12,l23,l30)

################################################SMALL DOMAIN
```

```
#ESTABLISHING PARAMETERS
xwidth=2000.0   #x width of model
zdepth=2000.0   #y width of model
sele_size=10.   #small element size
#TRANSFORM
xshift=width/2-xwidth/2
zshift=depth/2-zdepth/2
#DOMAIN CONSTRUCTION
p4=Point(xshift, zshift)
p5=Point(xwidth+xshift, zshift)
p6=Point(xwidth+xshift, zdepth+zshift)
p7=Point(xshift,    zdepth+zshift)
# Join corners in anti-clockwise manner.
l45=Line(p4, p5)
l56=Line(p5, p6)
l67=Line(p6, p7)
l74=Line(p7, p4)


csmall=CurveLoop(l45,l56,l67,l74)
```

The small sub-domain curve can then be used to create a surface.

```
ssmall=PlaneSurface(csmall)
```

However, so that the two domains do not overlap, when the big sub-domain curveloop is used to create a surface it must contain a hole. The hole is defined by the small sub-domain curveloop.

```
sbig=PlaneSurface(cbig,holes=[csmall])
```

The two sub-domains now have a common geometry and no over-laping features as per Figure 8.8. Notice, that both domains have a normal in the same direction.

The next step, is exporting each sub-domain individually, with an appropriate element size. This is carried out using the `esys.pycad` Design command.

```
# Design the geometry for the big mesh.
d1=Design(dim=2, element_size=bele_size, order=1)
d1.addItems(sbig)
d1.addItems(PropertySet(l01,l12,l23,l30))
d1.setScriptFileName(os.path.join(save_path,"example10m_big.geo"))
MakeDomain(d1)
```

```
# Design the geometry for the small mesh.
d2=Design(dim=2, element_size=sele_size, order=1)
d2.addItems(ssmall)
d2.setScriptFileName(os.path.join(save_path,"example10m_small.geo"))
MakeDomain(d2)
```

Finally, a system call to Gmsh is required to merge and then appropriately mesh the two sub-domains together.

```
# Join the two meshes using Gmsh and then apply a 2D meshing algorithm.
# The small mesh must come before the big mesh in the merging call!!@!!@!
sp.call("gmsh -2 "+
        os.path.join(save_path,"example10m_small.geo")+" "+
        os.path.join(save_path,"example10m_big.geo")+" -o "+
        os.path.join(save_path,"example10m.msh"),shell=True)
```

The "-2" option is responsible for the 2D meshing, and the "-o" option provides the output path. The resulting mesh is depicted in Figure 8.9

To use the Gmsh "*.msh" file in the solution script, the mesh reading function "ReadGmsh" is required. It can be imported via;

```
from esys.finley import ReadGmsh
```

To read in the file the function is called

```
domain=ReadGmsh(os.path.join(mesh_path,'example10m.msh'),2) # create the domain
```

where the integer argument is the number of domain dimensions.

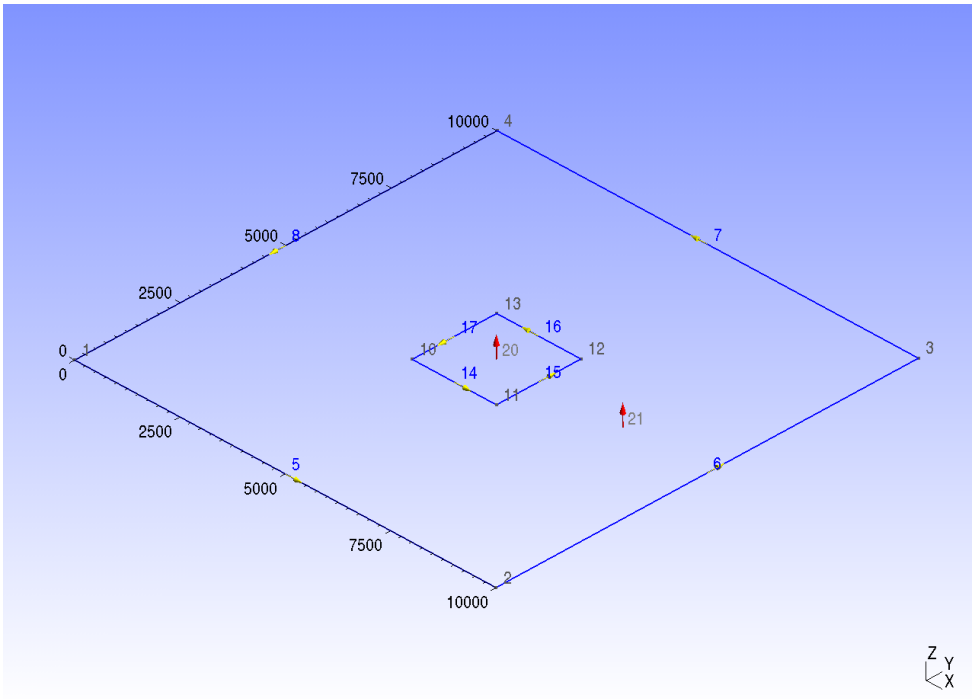


Figure 8.8: Geometry of two surfaces for a single domain.

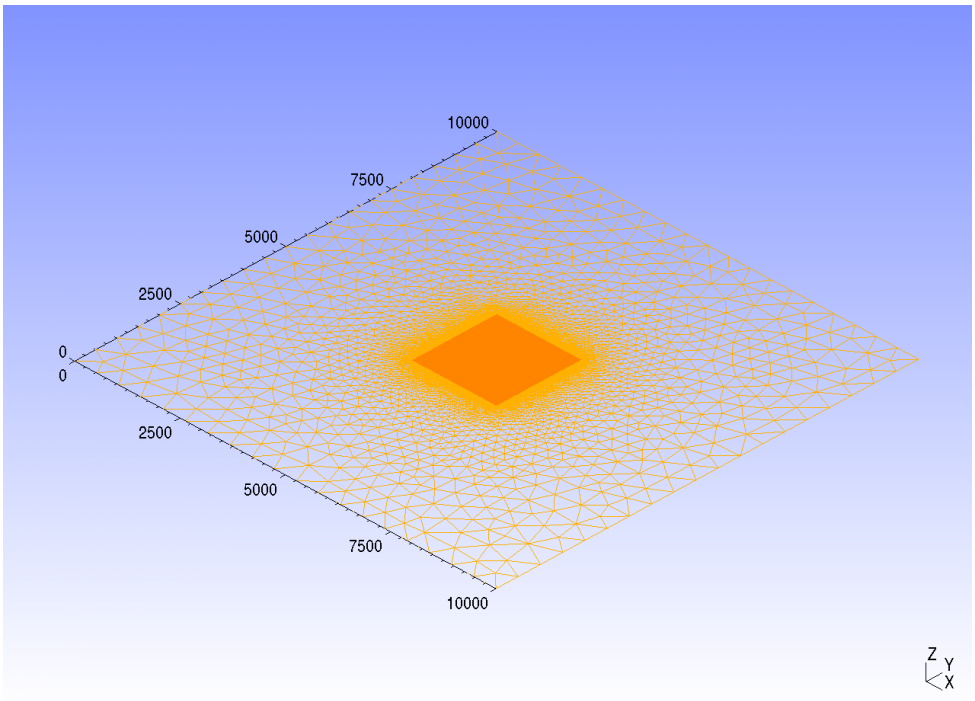


Figure 8.9: Mesh of merged surfaces, showing variable element size. Elements range from 10m in the centroid to 500m at the boundary.

8.5 Unbounded problems

With a variable element-size, it is now possible to solve the potential problem over a very large mesh. To test the accuracy of the solution, we will compare the *escript* result with the analytic solution for the vertical gravitational acceleration g_z of an infinite horizontal cylinder.

For a horizontal cylinder with a circular cross-section with infinite strike, the analytic solution is give by

$$g_z = 2\gamma\pi R^2 \Delta\rho \frac{z}{(x^2 + z^2)} \tag{8.10}$$

where γ is the gravitational constant (as defined previously), R is the radius of the cylinder, $\Delta\rho$ is the density contrast and x and z are geometric factors, relating the observation point to the center of the source via the horizontal and vertical displacements respectively.

The accuracy of the solution was tested using a square domain. For each test the dimensions of the domain were modified, being set to 5, 10, 20 and 40 Km. The results are compared with the analytic solution and are depicted in Figure 8.10 and Figure 8.11. Clearly, as the domain size increases, the *escript* approximation becomes more accurate at greater distances from the source. The same is true at the anomaly peak, where the variation around the source diminishes with an increasing domain size.

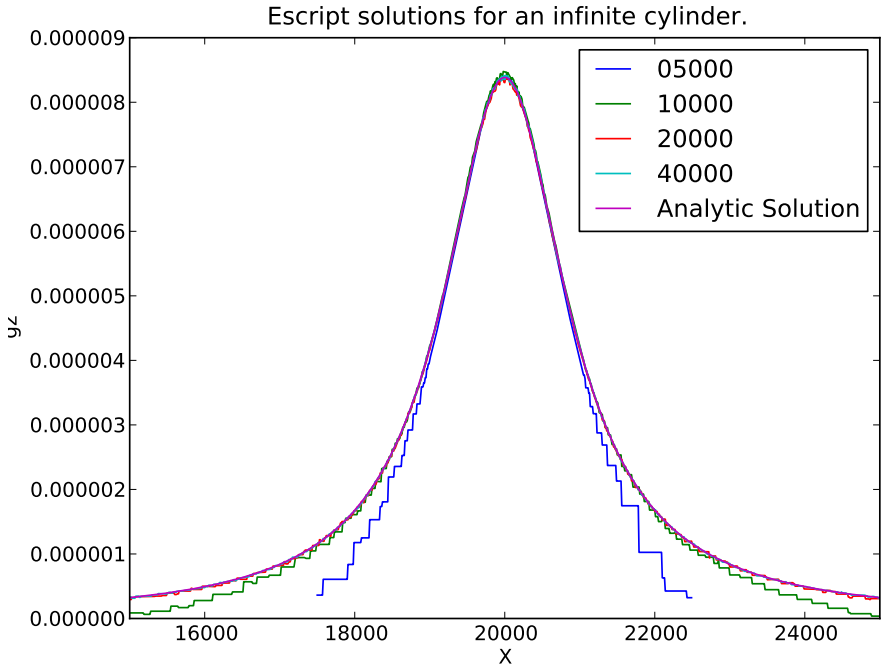


Figure 8.10: Solution profile 1000.0 meters from the source as the domain size increases.

There is a methodology which can help establish an appropriate zero mass region to a domain.

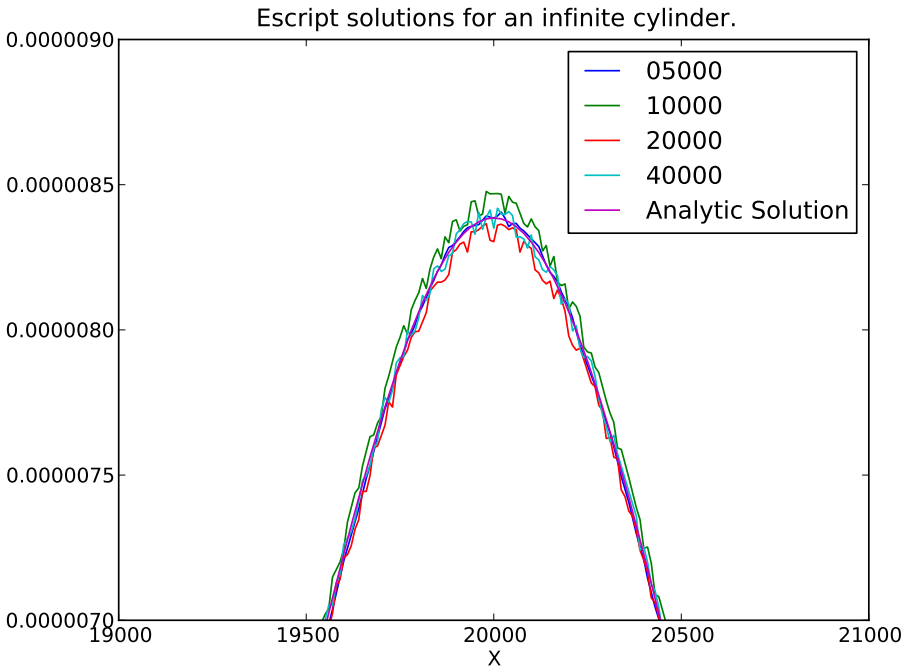


Figure 8.11: Magnification of Figure 8.10.

CHAPTER

NINE

Potential Fields - Electrical Resistivity

In this chapter we will investigate the effects of a current flow and resistivity in a medium. This type of problem is related to the DC resistivity method of geophysical prospecting. Currents are injected into the ground at the surface and measurements of the potential are taken at various potential-dipole locations along or adjacent to the survey line. From these measurements of the potential it is possible to infer an approximate apparent resistivity model of the subsurface.

The following theory comes from a tutorial by Loke [5]. We know from Ohm's law that the current flow in the ground is given in vector form by

$$\vec{J} = \sigma \vec{E} \tag{9.1}$$

where $\vec{J}$ is current density, $\vec{E}$ is the electric field intensity and σ is the conductivity. We can relate the potential to the electric field intensity by

$$E = -\nabla \Phi \tag{9.2}$$

where Φ is the potential. We now note that the current density is related to the potential via

$$\vec{J} = -\sigma \nabla \Phi \tag{9.3}$$

Geophysical surveys predominantly use current sources which individually act as point poles. Considering our model will contain volumes, we can normalise the input current and approximate the current density in a volume ΔV by

$$\nabla \vec{J} = \left(\frac{I}{\Delta V} \right) \delta(x - x_s)\delta(y - y_s)\delta(z - z_s) \tag{9.4}$$

$$-\nabla \cdot [\sigma(x,y,z)\nabla\phi(x,y,z)] = \left(\frac{I}{\Delta V}\right)\delta(x-x_s)\delta(y-y_s)\delta(z-z_s) \tag{9.5}$$

This form is quite simple to solve in *escript*.

9.1 3D Current-Dipole Potential

The scripts referenced in this section are; example11m.py; example11c.py

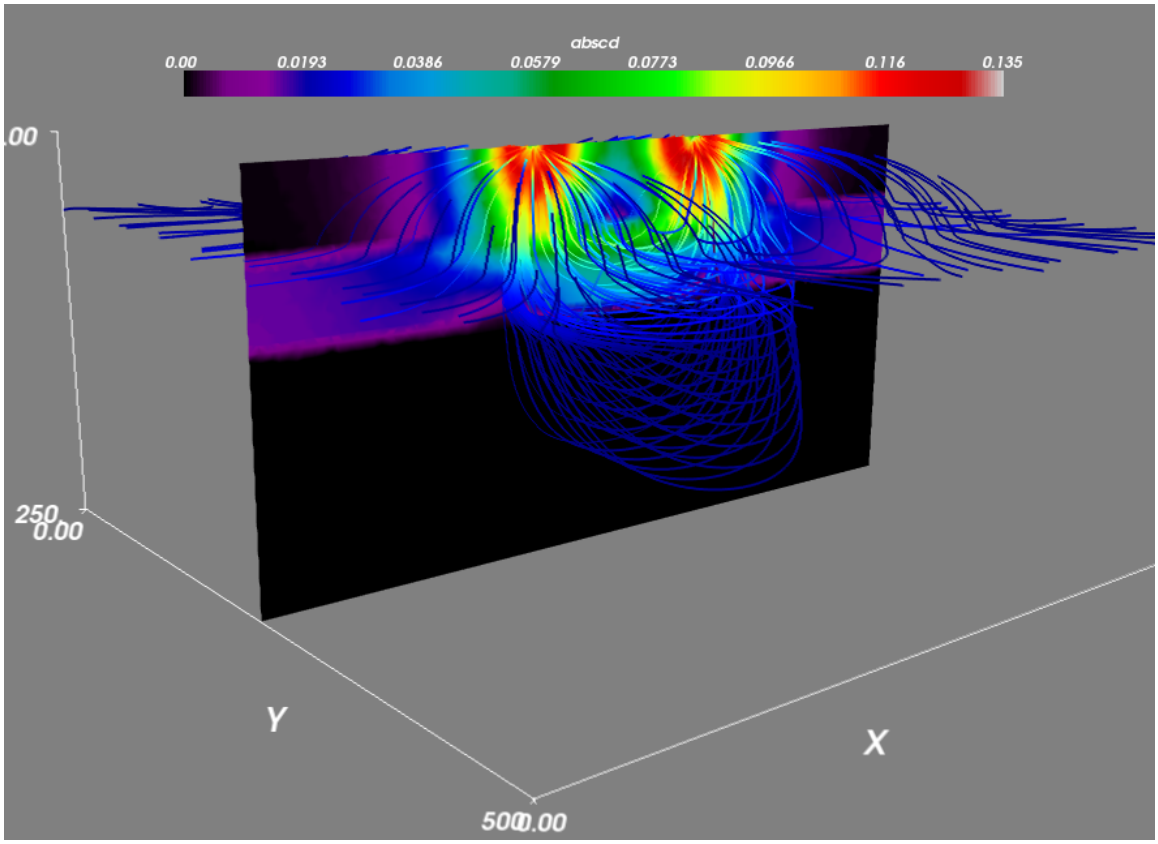


Figure 9.1: Current Density Model for layered medium.

9.2 Frequency Dependent Resistivity - Induced Polarisation

With a more complicated resistivity model it is possible to calculate the chargeability or IP effect in the model. A recent development has been the Fractal model for complex resistivity [3, 4].

The model is calculated over many frequencyies and transformed to the time domain using a discrete fourier transform.

References

[1] Richard J. Blakely. *Potential Theory in Gravity and Magnetic Applications*. Cambridge University Press, 1995.

[2] C. Cerjan. A nonreflecting boundary condition for discrete acoustic and elastic wave equations. *Geophysics*, 50, 1985. doi: 10.1190/1.1441945.

[3] V.J.da C. Farias, C.H. de M. Maranho, B.R.P. da Rocha, and N. de P.O. de Andrade. Induced polarization forward modelling using finite element method and the fractal model. *Applied Mathematical Modelling*, 34: 1849–1860, 2010.

[4] Mark Honig and Bulent Tezkan. 1d and 2d cole-cole-inversion of time-domain induced-polarization data. *Geophysical Prospecting*, 55:117–133, 2007.

[5] M. H. Loke. Tutorial: 2-d and 3-d electrical imaging surveys. Online, July 2004.

Index