`esys.downunder`: Inversion with *escript*

Release - 3.4
(r4488)

Cihan Altinay, Vince Boros, Lutz Gross, Azadeh Salehi

June 28, 2013

The University of Queensland
School of Earth Sciences
St. Lucia, QLD 4072, Australia.

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

Contents

Overview

The `esys.downunder` module for *python* is designed to perform the inversion of geophysical data such as gravity and magnetic anomalies using a parallel supercomputer. The solution approach bases entirely on the finite element method and is therefore different from the usual approach based on Green's functions and linear algebra techniques. The module is implemented on top of the *escript* solver environment for *python* and is distributed as part of the *escript* package through https://launchpad.net/escript-finley. We refer to the *escript* documentation [6] for installation instructions and to [7] for a basic introduction to *escript*.

This document is split into two parts: Part I provides a tutorial-style introduction to running inversions with the `esys.downunder` module. Users with minimal or no programming skills should be able to follow the tutorial which demonstrates how to run inversions of gravity anomaly data, magnetic anomaly data and the combination of both. The scripts and data files used in the examples are provided with the *escript* distribution.

Part II gives more details on the mathematical methods used and the module infrastructure. It is the intention of this part to give users a deeper understanding of how `esys.downunder` is implemented and also to open the door for experienced *python* programmers to build their own inversion programs using `esys.downunder` components and the *escript* infrastructure.

The development project of `esys.downunder` is part of the AuScope Inversion Lab. The work is funded under Australian Geophysical Observing System, see http://auscope.org.au/site/agos.php, through the Education Investment Fund of the Australian Commonwealth (2011-2014) and under the AuScope Sustainability Funding (2011-12) with the support of the School of Earth Sciences at the University of Queensland, see http://www.earth.uq.edu.au/.

Part I

Inversion Cookbook

CHAPTER
ONE

Gravity Inversion

1.1 Introduction

In this part of the documentation we give an introduction on how to use the `esys.downunder` module and the inversion driver functions to perform the inversion of gravity and magnetic data. The driver functions enable geologists and geophysicists to apply the `esys.downunder` module quickly and in an easy way without requiring detailed knowledge of the theory behind inversion or programming skills. However, users who are interested in specializing or extending the inversion capacity are referred to Part II of this manual. It is beyond the intention of this manual to give a detailed introduction to geophysical inversion, in particular to the appropriate preprocessing of data sets.

The `esys.downunder` module described here is designed to calculate estimations for the 3-D distribution of density and/or susceptibility from 2-D gravity and magnetic data measured in ground or airborne surveys. This process is generally called inversion of geophysical data. Following the standard assumption it is assumed that the data are measured as perturbation of an expected gravity and/or magnetic field of the Earth. In this context measured gravity and magnetic data are in fact describing anomalies in the gravity and magnetic field. As a consequence the inversion process provides corrections to an average density (typically $2670 kg/m^3$) and susceptibility (typically 0). So in the following we will always assume that given data are anomalies and therefore not in all cases explicitly use the terms gravity anomalies or density corrections but just use the terms gravity and density.

In this chapter we will give a detailed introduction into usage of the driver functions for inversion of gravity data. In Chapter 2 we will discuss the inversion of magnetic data using `esys.downunder`.

To run the examples discussed you need to have *escript* (version 3.3.1 or newer) installed on your computer. Moreover, if you want to visualize the results you need to have access to a data plotting software which is able to process *VTK* input files, e.g. *mayavi* or *VisIt*. As *mayavi* can be easily obtained and installed for most platforms the tutorial includes commands to visualize output files using *mayavi*. However, it is pointed out that *VisIt* is the preferred visualization tool for *escript* as it can deal with very large data sets more efficiently.

1.2 How does it work?

The execution of the inversion is controlled by a script which, in essence, is a text file and can be edited using any text editor. The script contains a series of statements which are executed by an interpreter which is an executable program reading the text file and executing the statements line-by-line. In the case of `esys.downunder` the interpreter is *python*. In order to be able to process the statements in each line of the script certain rules (called syntax) need to be obeyed. There is a large number of online tutorials for *python* available[1]. We also refer to the *escript* cook book [7] and user's guide [6] which is in particular useful for users who like to dive deeper into `esys.downunder`. For this part of the manual no *python* knowledge is required but it is recommended that users acquire some basic knowledge on *python* as they progress in their work with `esys.downunder`.

[1] e.g. `http://www.tutorialspoint.com/python` and `http://doc.pyschools.com`

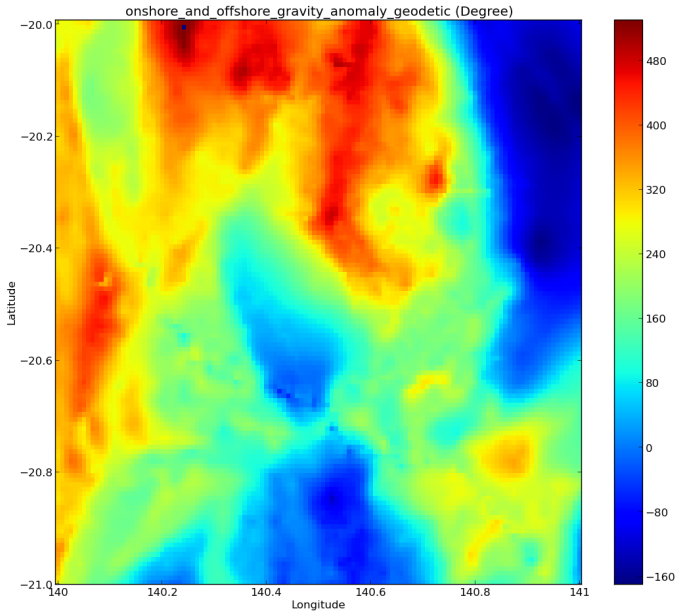


Figure 1.1: Gravity Anomaly Data in $mgal$ from Western Queensland, Australia (file `data/QLDWestGravity.nc`). Data obtained from Geoscience Australia.

The following script 1.1[2] is a simple example to run an inversion for gravity data:

Python Program 1.1

```
# Header:
from esys.downunder import *
from esys.weipa import *
from esys.escript import unitsSI as U

# Step 1: set up domain
dom=DomainBuilder()
dom.setVerticalExtents(depth=40.*U.km, air_layer=6.*U.km, num_cells=25)
dom.setFractionalPadding(pad_x=0.2, pad_y=0.2)
dom.fixDensityBelow(depth=40.*U.km)

# Step 2: read gravity data
source0=NetCdfData(NetCdfData.GRAVITY, 'GravitySmall.nc')
dom.addSource(source0)

# Step 3: set up inversion
inv=GravityInversion()
inv.setSolverTolerance(1e-4)
inv.setSolverMaxIterations(50)
inv.setup(dom)

# Step 4: run inversion
inv.getCostFunction().setTradeOffFactorsModels(10.)
rho = inv.run()

# Step 5: write reconstructed density to file
saveVTK("result.vtu", density=rho)
```

[2]The script is similar to `grav_netcdf.py` within the *escript* example file directory.

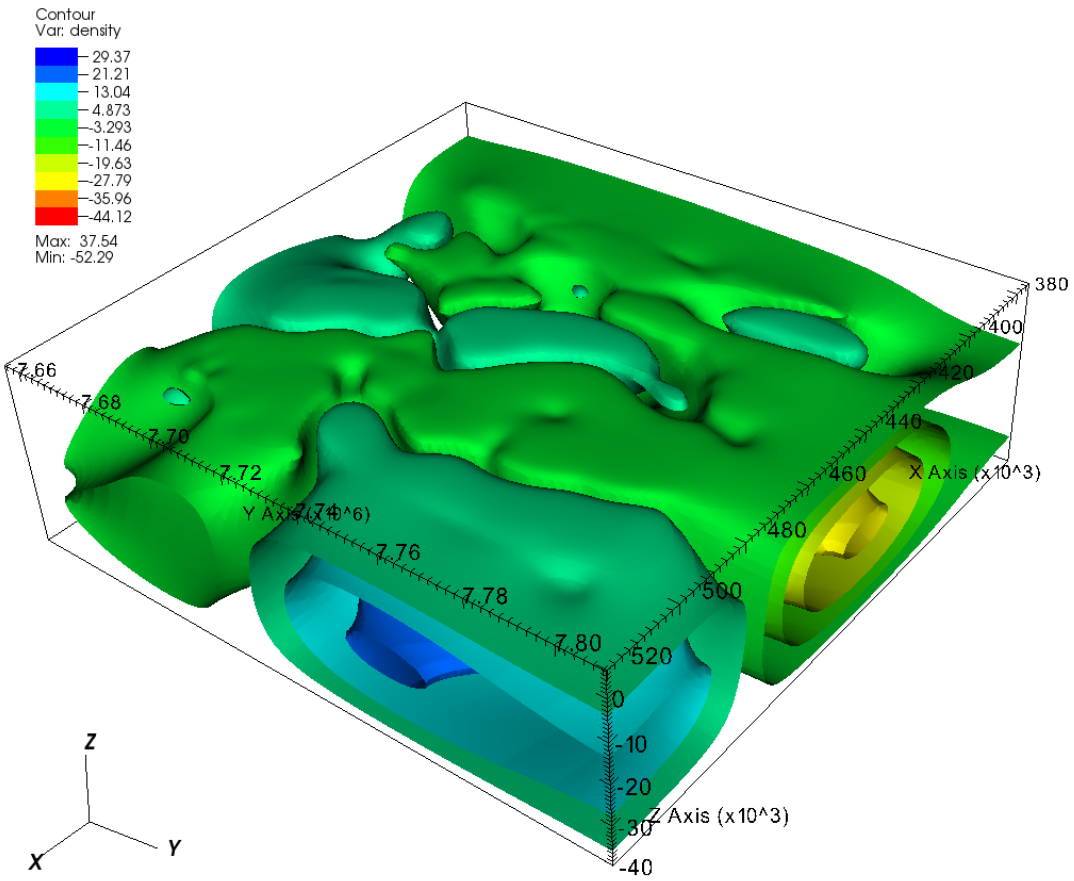


Figure 1.2: 3-D contour plot of the density distribution obtained by inversion of file `data/QLDWestGravity.nc` (with $\mu = 10$). Colours represent values of density where high values are represented by blue and low values are represented by red.

The result, in this case the density distribution, is written to an external file for further processing. You can copy and paste the text of the script into a file of any name, let's say for further reference we use the file name `grav.py`. It is recommended to use the extension `.py` to identify the file as a *python* script. We will discuss the statements of the script later in this chapter.

Obviously the inversion needs to be fed with some gravity data. You can find example data from western Queensland, Australia in two resolutions in the *escript* example directory. In this case the data are loaded from the file `GravitySmall.nc` which is given in the *netCDF* file format. After you have copied this file into the directory in which you have saved the script `grav.py` you can run the program using the command line

```
run-escript grav.py
```

We are running `grav.py` through the *escript* start-up command since *escript* is used as a back end for the inversion algorithm[3]. Obviously it is assumed that you have an installation of *escript* available on your computer, see https://launchpad.net/escript-finley.

After the execution has successfully completed you will find the result file `result.vtu` in the directory from where you have started the execution of the script. The file has the *VTK* format and can be imported easily into many visualization tools. One option is the *mayavi* package which is available on most platforms. You can invoke the visualization using the commands

```
mayavi2 -d result.vtu -m SurfaceMap
```

from the command line. Figure 1.2 shows the result of this inversion as a contour plot[4], while the gravity anomaly data is shown in Figure 1.1. We will discuss data later in Section 1.4.

Let us take a closer look at the script[5]. Besides the header section one can separate the script into five steps:

1. set up domain on which the inversion problem is solved
2. load the data
3. set-up the inversion problem
4. run the inversion
5. further processing of the result. Here we write the reconstructed density distribution to a file.

In the following we will discuss the steps of the scripts in more detail. Before we do this it is pointed out that the header section, following *python* conventions, makes all required packages available to access within the script. At this point we will not discuss this in more details but emphasize that the header section is a vital part of the script. It is is required in each `esys.downunder` inversion script and should not be altered except if additional modules are needed.

1.3 Creating the Inversion Domain

First step in Script 1.1 is the definition of the domain over which the inversion is performed. We think of the domain as a block with orthogonal, plain faces. Figure 1.3 shows the set-up for a two-dimensional domain (see also Figure 6.1 for 3-D). The lateral coordinates along the surface of the Earth are denoted by x and y (only x-direction is used in 2-D). The z direction defines the vertical direction where the part above the surface has positive coordinates and the subsurface negative coordinates. The height of the section above the surface, which is assumed to be filled with air, needs to be set by the user. The inversion assumes that the density in the section is known to be zero[6]. The density below the surface is unknown and is calculated through the inversion. The user needs to specify the depth below the surface in which the density is to be calculated. The lateral extension of the domain is defined by the data sets fed into the inversion. It is chosen large enough to cover all data sets (in case more than one is used). In order to reduce the impact of the boundary a padding zone around the data sets can be introduced.

The reconstruction of the gravity field from an estimated density distribution is the key component of the inversion process. To do this `esys.downunder` uses the finite element method (FEM). We need to introduce a

[3] Please see the *escript* user's guide [6] on how to run your script in parallel using threading and/or MPI.
[4] These plots were generated by *VisIt* using the higher resolution data.
[5] In *python* lines starting with '#' are comments and are not processed further.
[6] Always keeping in mind that these are not absolute values but anomalies.

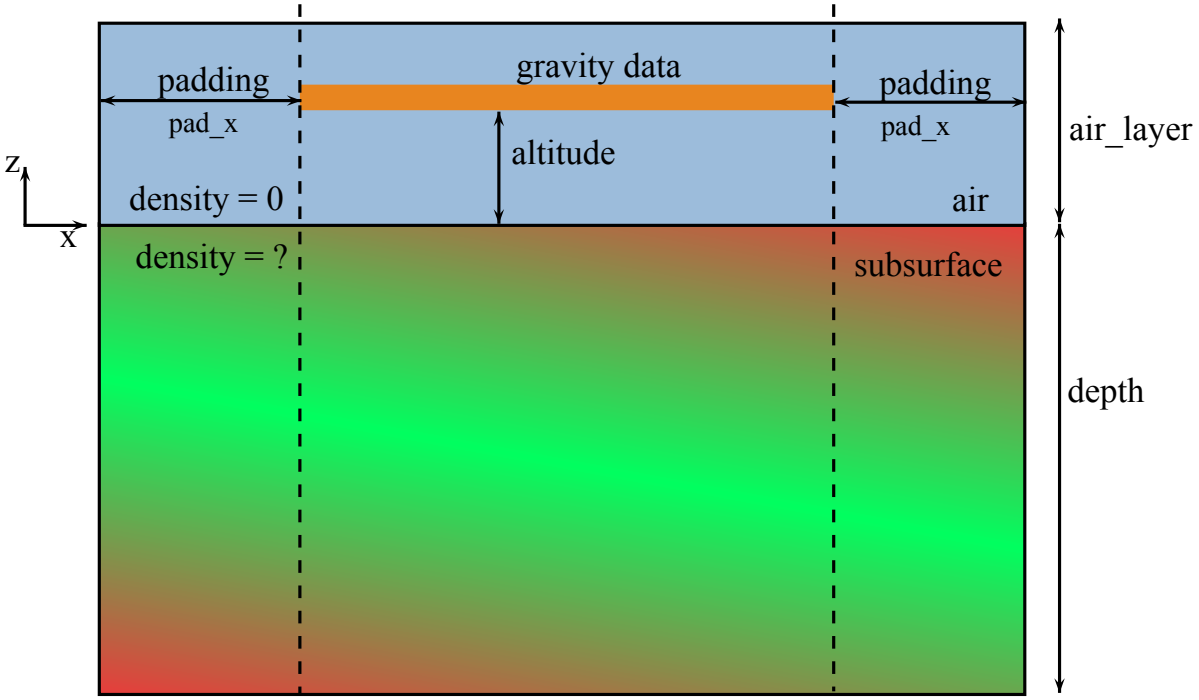


Figure 1.3: 2-D domain set-up for gravity inversion

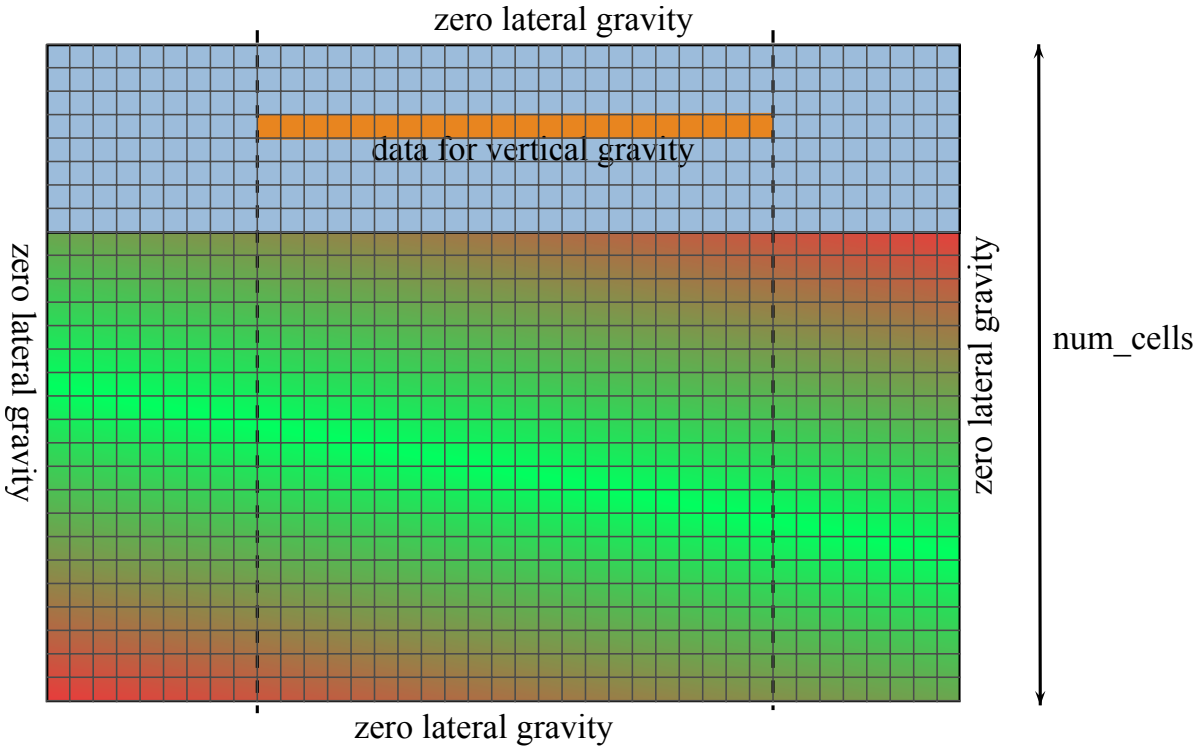


Figure 1.4: Cell distribution and boundary conditions for a 2-D domain

grid over the domain, see Figure 1.4. The number of vertical cells is set by the user while the number of horizontal cells is derived from the grid spacing of the gravity data set(s). It is assumed that gravity field data given are constant across a cell. To be able to reconstruct the gravity field some assumptions on the values of the gravity field on the domain boundary have to be made. `esys.downunder` assumes that on all faces the lateral gravity field component equals zero. No assumptions on the horizontal components are made[7][8].

In script 1.1 the statement

```
dom=DomainBuilder()
```

creates something like a factory to build a domain. We then define the features of the domain we would like to create:

```
dom.setVerticalExtents(depth=40.*U.km, air_layer=6.*U.km, num_cells=25)
dom.setFractionalPadding(pad_x=0.2, pad_y=0.2)
```

Here we specify the depth of the domain to $40km$, the thickness of the air layer above the surface to $6km$ and the number of vertical cells to 25. We also introduce a lateral padding of 20% of the expansion of the gravity data on each side of the data and in both lateral directions.

In some cases it can be appropriate to assume that the density below a certain depth is zero[9]. The statement

```
dom.fixDensityBelow(depth=40.*U.km)
```

introduces this constraint. As in the case discussed here if the depth for zero density is not less than the depth of the domain no constraint at depth is applied to the density.

`esys.downunder` uses the metre-kilogram-second based International System of Units (SI)[10]. So all values must be converted to appropriate units. This does not apply to geographic coordinates which in `esys.downunder` are given in fractional degrees (as a floating point number) to represent longitude and latitude. In the script we have used the expression

```
depth=40.*U.km
```

to define the depth of the domain to $40km$. The expression `U.km` denotes the unit km (kilometer) and ensures appropriate conversion of the value 40 into the base unit m (meter). It is recommended to add units to values (where present) in order to make sure that the final values handed into `esys.downunder` is given with the appropriate units. The physical units module of *escript*, which we have imported here under the name `U` in the script header, defines a large number of physical units and constants, please see [6] and [1].

1.4 Loading Gravitational Data

In practice gravity acceleration is measured in various ways, for instance by airborne surveys [16]. `esys.downunder` assumes that all data supplied as input are already appropriately pre-processed. In particular, corrections for

- free-air, to remove effects from altitude above ground;
- latitude, to remove effects from ellipsoidicity of the Earth;
- terrain, to remove effects from topography

must have been applied to the data. In general, data prepared in such a form are called Bouguer anomalies [16].

To load gravity data into `esys.downunder` the data are given on a plane parallel to the surface of the Earth at a constant altitude, see diagram 1.3. The data need to be defined over a rectangular grid covering a subregion of the Earth surface. The grid uses a geographic coordinate system with latitudes and longitudes assumed to be given in the Clarke 1866 geodetic system. Figure 1.1 shows an example of such a data set from the western parts of Queensland, Australia. The data set covers a rectangular region between 140^o and 141^o east and between 20^o

[7] It is assumed that the gravity potential equals zero on the top and bottom surface, see Section 9.1 for details

[8] Most inversion codes use Green's functions over an unbounded domain to reconstruct the gravity field. This approach makes the assumption that the gravity field (or potential) is converging to zero when moving away from the region of interest. The boundary conditions used here are stronger in the sense that the lateral gravity component is enforced to be zero in a defined distance of the region of interest but weaker in the sense that no constraint on the horizontal component is applied.

[9] As we are in fact calculating density corrections this means that the density is assumed to be the average density.

[10] see http://en.wikipedia.org/wiki/International_System_of_Units

and 21^o south. Notice that latitude varies between -90^o to 90^o where negative signs refer to places in the southern hemisphere and longitude varies between -180^o to 180^o where negative signs refer to places west of Greenwich. The colour at a location represents the value of the vertical Bouguer gravity anomaly at this point at the surface of the Earth. Values in this data set range from -160 $mgal$ to about 500 $mgal$[11] over a 121×121 grid.

In general, a patch of gravity data needs to be defined over a plane `NX` $\times$ `NY` where `NX` and `NY` define the number of grid lines in the longitude (`X`) and the latitude (`Y`) direction, respectively. The grid is spanned from an origin with spacing `DELTA_X` and `DELTA_Y` in the longitude and the latitude direction, respectively. The gravity data for all grid points need to be given as an `NX` $\times$ `NY` array. If available, measurement errors can be associated with the gravity data. The values are given as an `NX` $\times$ `NY` array matching the shape of the gravity array. Note that data need not be available on every single point of the grid, see Section 1.7.2 for more information on this.

Currently, two data file formats are supported, namely *ER Mapper Raster* [2] files and *netCDF* [12] files. In the examples of this chapter we use *netCDF* files and refer to Section 1.7.1 and Section 6.3.1 for more information on using ER Mapper Raster files. If you have data in any other format you have the option of writing a suitable reader (for advanced users, see Chapter 6) or, assuming you are able to read the data in *python*, refer to the example script `create_netcdf.py` which shows how to create a file in the *netCDF* file format [12] compatible with `esys.downunder` from a data array.

In script 1.1 we use the statement

```
source0=NetCdfData(NetCdfData.GRAVITY, 'GravitySmall.nc')
```

to load the gravity data stored in `GravitySmall.nc` in the *netCDF* format. Within the script the data set is now available under the name `source0`. We need to link the data set to the `DomainBuilder` using

```
dom.addSource(source0)
```

At the time the domain for the inversion is built the `DomainBuilder` will use the information about origin, extent and spacing along with the other options provided to build an appropriate domain. As at this point a flat Earth is assumed geographic coordinates used to represent data in the input file are mapped to a (local) Cartesian coordinate system. This is achieved by projecting the geographic coordinates into the *Universal Transverse Mercator* (UTM) coordinate system[12].

There are a few optional arguments you can add when constructing a data source. While Section 6.3 has a detailed description of all arguments it is worth noting a few. Firstly, it is important to know that data slices are assumed to be at altitude 0 by default. This can be easily changed though:

```
source0=NetCdfData(NetCdfData.GRAVITY, 'GravitySmall.nc', altitude=2.5*U.km)
```

Another important setting is the scale or unit of the measurements. The default is dependent on the data type and for gravity anomalies a scale of $\frac{\mu m}{sec^2}$ (or 0.1 $mgal$) is assumed. For instance to change the default scale to $mgal$ (which is $10^{-5}\frac{m}{sec^2}$), you could use:

```
source0=NetCdfData(NetCdfData.GRAVITY, 'GravitySmall.nc', scale_factor=U.mgal)
```

Finally, it is possible to specify measurement errors (i.e. uncertainties) alongside the data. Since these can never be zero, a value of 2 units is used if nothing else has been specified. The error value is assumed to be given in the same units as the data so the default value translates to an error of 0.2 $mgal$. There are two possibilities to specify the error, namely by providing a constant value which is applied to all data points:

```
source0=NetCdfData(NetCdfData.GRAVITY, 'GravitySmall.nc', error=1.7)
```

or, if the information is available in the same *netCDF* file under the name `errors`, provide `esys.downunder` with the appropriate variable name:

```
source0=NetCdfData(NetCdfData.GRAVITY, 'GravitySmall.nc', error="errors")
```

It is important to keep an eye on the complexity of the inversion. A good measure is the total number of cells being used. Assume we have given a data set on a 20×20 grid and we add lateral padding of, say, 20% to each side of the data, the lateral number of cells becomes $(20 \cdot 1.4) \times (20 \cdot 1.4) = 1.4^2 \cdot 20^2 \approx 2 \cdot 10^2 = 800$. If we use 20 cells in the vertical direction we end up with a total number of $800 \times 20 = 16,000$ cells. This size can be

[11] The unit $mgal$ means milli gal (galileo) with $1\ gal = 0.01\frac{m}{sec^2}$.

[12] See e.g. http://en.wikipedia.org/wiki/Universal_Transverse_Mercator_coordinate_system

easily handled by a modern desktop PC. If we increase the grid size of the data to 40×40 points and use 40 cells in the vertical extent we get a total of $(2 \cdot 40^2) \cdot 40 = 128,000$ cells, a problem size which is considerably larger but can still be handled by a desktop computer. Taking this one step further, if the amount of data is increased to 200×200 points and we use 200 cells in the vertical extent the domain will contain $16,000,000$ (16 million) cells. This scenario requires a computer with enough memory and (a) fast processor(s) to run the inversion. This estimate of complexity growth applies to the case where the increase of data grid size is driven by an increase of resolution where it is recommended to increase the vertical resolution in synch with the lateral resolution. Note that if more than one data set is used the target resolution will be the resolution of the finest data set (see also Section 1.7.3). In other cases the expansion of the region of interest drives an increase of data grid size and the increase of total number of cells is less dramatic as the vertical number of cells can remain constant while keeping a balanced resolution in vertical and lateral direction.

1.5 Setting up the Inversion and Running it

We are now at step three of script 1.1 in which the actual inversion is set up. First we create an empty inversion under the name `inv`:

```
inv=GravityInversion()
```

As indicated by the name we can use `inv` to perform an inversion of gravity data[13]. The inversion is an iterative process which sequentially calculates updates to the density distribution in an attempt to improve the match of the gravity field produced by the density distribution with the data. Termination of the iteration is controlled by the tolerance which is set by the user:

```
inv.setSolverTolerance(1e-4)
```

Here we set the tolerance to 10^{-4}, i.e. the iteration is terminated if the maximum density correction is less than or equal to 10^{-4} relative to the maximum value of estimated density anomaly. In case the iteration does not converge a maximum number of iteration steps is set:

```
inv.setSolverMaxIterations(50)
```

If the maximum number of iteration steps (here 50) is reached the iteration process is aborted and an error message is printed. In this case you can try to rerun the inversion with a larger value for the maximum number of iteration steps. If even for a very large number of iteration steps no convergence is achieved, it is very likely that the inversion has not been set up properly.

The statement

```
inv.setup(dom)
```

links the inversion with the domain and the data. At this step – as we are solving a gravity inversion problem – only gravitational data attached to the domain builder `dom` are considered. Internally a cost function J is created which is minimized during the inversion iteration. It is a combination of a measure of the data misfit of the gravity field from the given density distribution and a measure of the smoothness of the density distribution. The latter is often called the regularization term. By default the gradient of density is used as the regularization term, see also Section 1.7.4. Obviously, the result of the inversion is sensitive to the weighting between the misfit and the regularization. This trade-off factor μ for the misfit function is set by the following statement:

```
inv.getCostFunction().setTradeOffFactorsModels(0.1)
```

Here we set $\mu = 0.1$. The statement `inv.setup` must appear in the script before setting the trade-off factor. A small value for the trade-off factor μ will give more emphasis to the regularization component and create a smoother density distribution. A large value of the trade-off factor μ will emphasize the misfit more and typically creates a better fit to the data and a rougher density distribution. It is important to keep in mind that the regularization reduces noise in the date and in fact gives the problem a unique solution. Consequently, the trade-off factor μ may not be chosen too large in order control the noise on the solution and ensure convergence in the iteration process.

We can now actually run the inversion:

[13] `GravityInversion` is a driver with a simplified interface which is provided for convenience. See Part II for more details on how to write inversion scripts with more general functionality, e.g. constraints.

```
rho = inv.run()
```

The answer as calculated during the inversion is returned and can be accessed under the name `rho`. As pointed out earlier the iteration process may fail in which case the execution of the script is aborted with an error message.

1.6 Taking a Look

In the final step of script 1.1 the calculated density distribution is written to an external file. A popular file format used by several visualization packages such as *VisIt* [17] and *mayavi* [10] is the *VTK* file format. The result of the inversion which has been named `rho` can be written to the file `result.vtu` by adding the statement

```
saveVTK("result.vtu", density=rho)
```

at the end of script. The inversion solution is tagged with the name `density` in the result file, however any other name for the tag could be used. As the format is text-based (as opposed to binary) *VTK* files tend to be very large and take compute time to create, in particular when it comes to large numbers of cells ($> 10^6$). For large problems it is more efficient to use the *SILO* file format [15]. *SILO* files tend to be smaller and are faster generated and read. It is the preferred format to import results into the visualization program *VisIt* [17] which is particularly suited for the visualization of large data sets. Inversion results can directly exported into *SILO* files using the statement

```
saveSilo("result.silo", density=rho)
```

replacing the `saveVTK(...)` statement. Similar to *VTK* files the result `rho` is tagged with the name `density` so it can be identified in the visualization program.

Another useful output option is the *Voxet* format which is understood by the *GOCAD* [5] geologic modelling software. In order to write inversion results to *Voxet* files use the statement

```
saveVoxet("result.vo", density=rho)
```

Unlike the other output formats *Voxet* data consists of a header file with the file extension `.vo` and separate *property* files without file extension. The call to `saveVoxet(...)` above would produce the files `result.vo` and `result_density`.

Figures 1.5 and 1.6 show two different styles of visualization generated in *VisIt* using the result of the inversion of the gravity anomalies shown in Figure 1.1. The inversions have been performed with different values for the model trade-off factor μ. The visualization shows clearly the smoothing effect of lower values for the trade-off factors. For larger values of the trade-off factor the density distribution becomes rougher showing larger details. Computational costs are significantly higher for larger trade-off factors. Moreover, noise in the data has a higher impact on the result. Typically several runs are required to adjust the value for the trade-off factor to the datasets used.

For some analysis tools it is useful to process the results in form of Comma-separated Values (*CSV*)[14]. Such a file can be created using the statement

```
saveDataCSV("result.csv", x=rho.getFunctionSpace().getX(), density=rho)
```

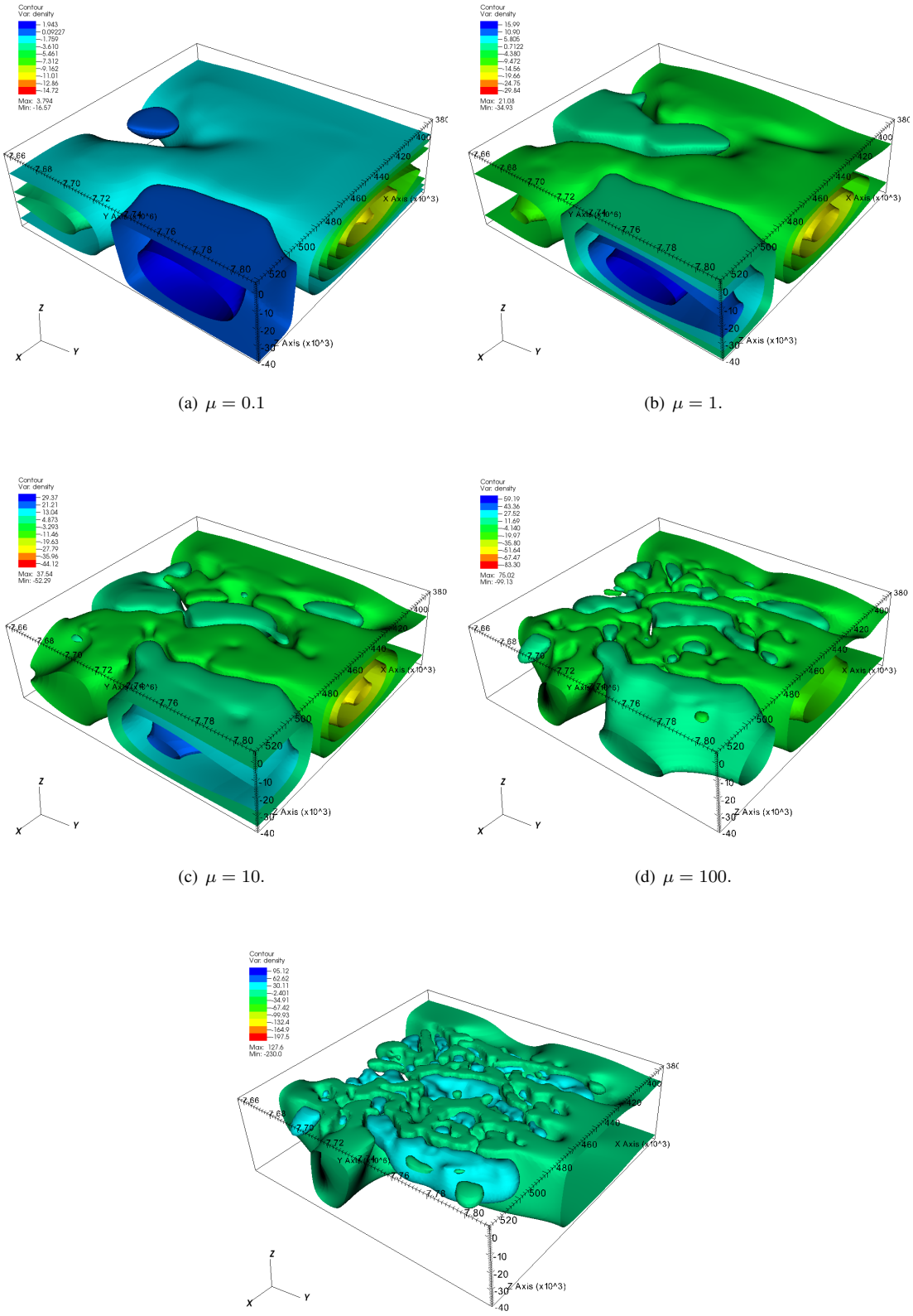
in the script. This will create a `result.csv` with columns separated by a comma. Each row contains the value of the density distribution and the three coordinates of the corresponding location in the domain. There is a header specifying the meaning of the corresponding column. Notice that rows are not written in a particular order and therefore, if necessary, the user has to apply appropriate sorting of the rows. Columns are written in alphabetic order of their corresponding tag names. For the interested reader: the statement `rho.getFunctionSpace()` returns the type used to store the density data `rho`. The `getX()` method returns the coordinates of the sampling points used for the particular type of representation, see [6] for details.

1.7 Remarks

1.7.1 ER Mapper Raster Files

The `esys.downunder` module can read data stored in ER Mapper Raster files. A data set in this format consists of two files, a header file whose name usually ends in `.ers` and the actual data file which has the same filename

[14] see http://en.wikipedia.org/wiki/Comma-separated_values



(a) $\mu = 0.1$

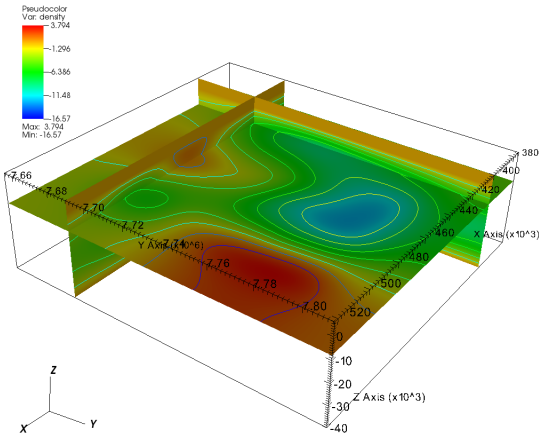
(b) $\mu = 1.$

(c) $\mu = 10.$

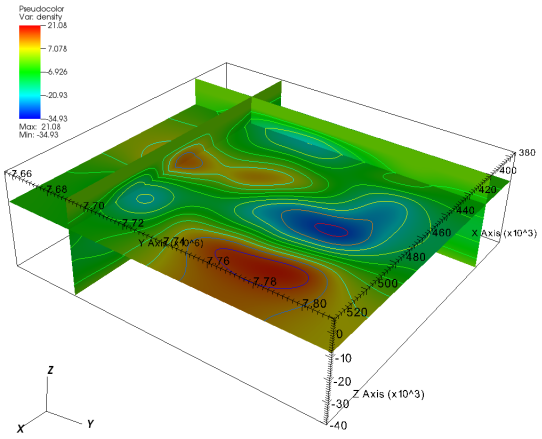
(d) $\mu = 100.$

(e) $\mu = 1000.$

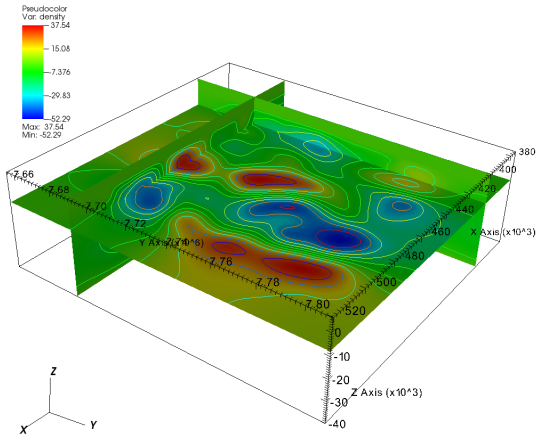
Figure 1.5: 3-D contour plots of gravity inversion results with data from Figure 1.1 for various values of the model trade-off factor μ. Visualization has been performed in *VisIt*.



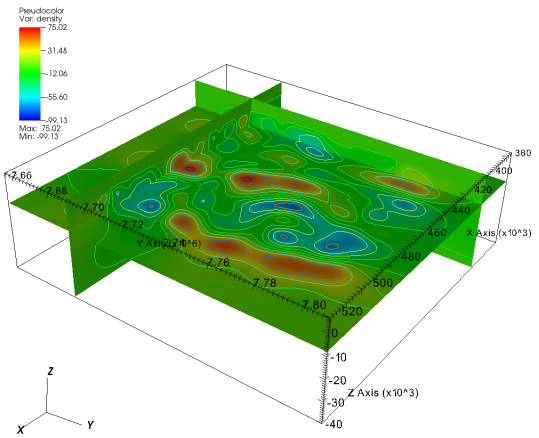
(a) $\mu = 0.1$



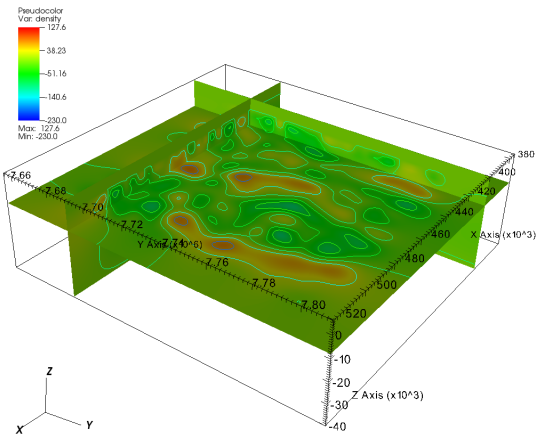
(b) $\mu = 1.$



(c) $\mu = 10.$



(d) $\mu = 100.$



(e) $\mu = 1000.$

Figure 1.6: 3-D slice plots of gravity inversion results with data from Figure 1.1 for various values of the model trade-off factor μ. Visualization has been performed *VisIt*.

as the header file but without any file extension. These files are usually produced by a commercial software package and the contents can be quite diverse. Therefore, it is not guaranteed that every data set is supported by `esys.downunder` but the most common types of raster data should work[15].

The interface for loading ER Mapper files is very similar to the *netCDF* interface described in Section 1.4. To load gravity data stored in the file pair[16] `GravitySmall.ers` (the header) and `GravitySmall` (the data) without changing any of the defaults use:

```
source0=ErMapperData(ErMapperData.GRAVITY, 'GravitySmall.ers')
```

If your data set does not follow the default naming convention you can specify the name of the data file explicitly:

```
source0=ErMapperData(ErMapperData.GRAVITY, 'GravitySmall.ers',
                     datafile='GravityData')
```

Please note that there is no way for the reader to determine if the two files really form a pair so make sure to pass the correct filenames when constructing the reader object. The same optional arguments explained in sections 1.4 and 1.7.2 are available for ER Mapper data sets. However, due to the limitation of the file format only a constant error value is supported.

1.7.2 Data With Holes

As described previously in this chapter input data is always given in the form of a rectangular grid with constant cell size in each dimension. However, there are cases when this is not necessarily the case. Consider an onshore data set which includes parts of the offshore region as in Figure 1.7. The valid data in this example has a value range of about -600 to 600 and the inversion is to be run based on these values only, disregarding the offshore region. In order to achieve that, the offshore region is *masked* by using a constant value which is not found within the onshore area. Figure 1.7 clearly shows this masked area in dark blue since a mask value of -1000 was used.

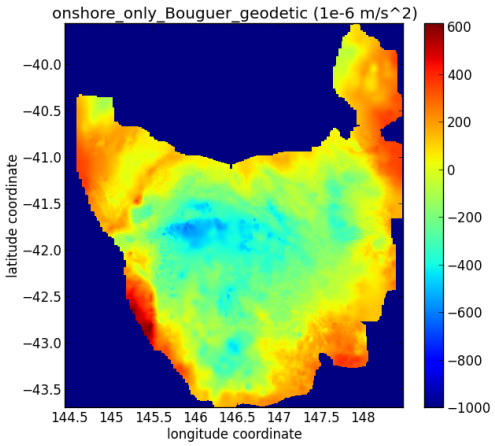


Figure 1.7: Plot of a rectangular gridded onshore data set that includes offshore regions which have a value (here -1000) not found within the real data (Bouguer anomalies in Tasmania, courtesy Geoscience Australia)

The *netCDF* conventions supported in `esys.downunder` include a standard way of specifying such a mask value. The example script `create_netcdf.py` demonstrates how this is accomplished in an easy way with any data. If, for any reason, the mask value in the input file is invalid it can be overridden via the `null_value` argument when constructing the `NetCdfData` object:

```
source0=NetCdfData(NetCdfData.GRAVITY, 'data0.nc', null_value=-1000)
```

In this example, all data points that have a value of -1000 are ignored and not used in the inversion. Please note that the special value *NaN* (not-a-number) is sometimes used for the purposes of masking in data sets. Areas marked with this value are always disregarded in `esys.downunder`.

[15] If your data does not load please contact us through https://launchpad.net/escript-finley.
[16] These files are available in the example directory.

1.7.3 Multiple Data Sets

It is possible to run a single inversion using more than one input data set, possibly in different file formats. To do so, simply create the data sources and add them to the domain builder:

```
source0=NetCdfData(NetCdfData.GRAVITY, 'data0.nc')
source1=ErMapperData(ErMapperData.GRAVITY, 'data1.ers')
dom.addSource(source0)
dom.addSource(source1)
```

However, there are some restrictions when combining data sets:

- Due to the coordinate transformation all data sets must be located in the same UTM zone. If a single dataset crosses UTM zones only the zone of the central longitude is used when projecting. For example, if one data set lies mostly in zone 51 but contains areas of zone 52, it is transformed using zone 51. In this case more data from zone 51 can be added, but not from any other zone.
- All data sets should have the same spatial resolution but this is not enforced. Combining data with different resolution is currently considered experimental but works best when the resolutions are multiples of each other. For example if the first data set has a resolution (or cell size) of 100 metres and the second has a cell size of 50 metres then the target domain will have a cell size of 50 metres (the finer resolution) and each point of the coarse data will occupy two cells (in the respective dimension).

1.7.4 Regularization Term

The `GravityInversion` class supports the following form for the regularization:

$$\int w^{(0)} \cdot \rho^2 + w_0^{(1)} \rho_{,0}^2 + w_1^{(1)} \rho_{,1}^2 + w_2^{(1)} \rho_{,2}^2 \, dx \tag{1.1}$$

where the integral is calculated across the entire domain. ρ represents the density distribution where $\rho_{,0}$ $\rho_{,1}$ and $\rho_{,2}$ are the spatial derivatives of ρ with respect to the two lateral and the vertical direction, respectively. $w^{(0)}$, $w_0^{(1)}$, $w_1^{(1)}$ and $w_2^{(1)}$ are weighting factors[17]. By default these are $w^{(0)} = 0$, $w_0^{(1)} = w_1^{(1)} = w_2^{(1)} = 1$. Other weighting factors can be set in the inversion set-up. For instance to set $w^{(0)} = 10$, $w_0^{(1)} = w_1^{(1)} = 0$ and $w_2^{(1)} = 100$ use the statement:

```
inv.setup(dom, w0=10, w1=[0,0,100])
```

It is pointed out that the weighting factors are rescaled in order to improve numerical stability. Therefore the relative size of the weighting factors is relevant and using

```
inv.setup(dom, w0=0.1, w1=[0,0,1])
```

would lead to the same regularization as the statement above.

[17] A more general form, e.g. spatially variable values for the weighting factors, is supported, see Part II

CHAPTER
TWO

Magnetic Inversion

Magnetic data report the observed magnetic flux density over a region above the surface of the Earth. Similar to the gravity case the data are given as deviation from an expected background magnetic flux density B^b of the Earth. Example data in units of nT (nano Tesla) are shown in Figure 2.1. It is the task of the inversion to recover the susceptibility distribution k from the magnetic data collected. The approach for inverting magnetic data is almost identical to the one used for gravity data. In fact the `esys.downunder` script 2.1 used for the magnetic inversion is very similar to the script 1.1 for gravity inversion.

Python Program 2.1

```
# Header:
from esys.downunder import *
from esys.weipa import *
from esys.escript import unitsSI as U


# Step 1: set up domain
dom=DomainBuilder()
dom.setVerticalExtents(depth=40.*U.km, air_layer=6.*U.km, num_cells=25)
dom.setFractionalPadding(pad_x=0.2, pad_y=0.2)
B_b = [2201.*U.Nano*U.Tesla,  31232.*U.Nano*U.Tesla, -41405.*U.Nano*U.Tesla]
dom.setBackgroundMagneticFluxDensity(B_b)
dom.fixSusceptibilityBelow(depth=40.*U.km)

# Step 2: read magnetic data
source0=NetCdfData(NetCdfData.MAGNETIC, 'MagneticSmall.nc', scale_factor=U.Nano * U.Tesla)
dom.addSource(source0)

# Step 3: set up inversion
inv=MagneticInversion()
inv.setSolverTolerance(1e-4)
inv.setSolverMaxIterations(50)
inv.fixMagneticPotentialAtBottom(False)
inv.setup(dom)

# Step 4: run inversion
inv.getCostFunction().setTradeOffFactorsModels(0.1)
k = inv.run()

# Step 5: write reconstructed susceptibility to file
saveVTK("result.vtu", susceptibility=k)
```

The structure of the script is identical to the gravity case. Following the header section importing the necessary modules the domain of the inversion is defined in step one. In step two the data are read and added to the domain

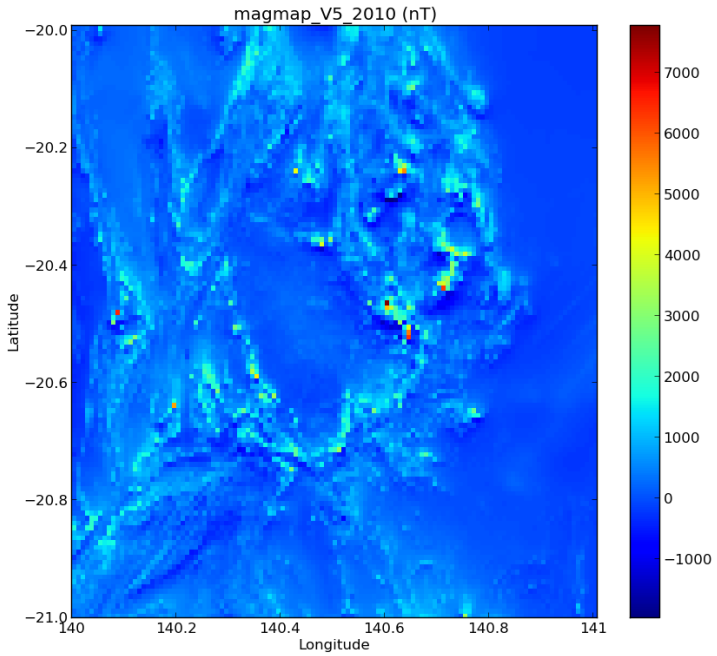


Figure 2.1: Magnetic anomaly data in nT from Western Queensland, Australia (file `data/QLDWestMagnetic.nc`). Data obtained from Geoscience Australia.

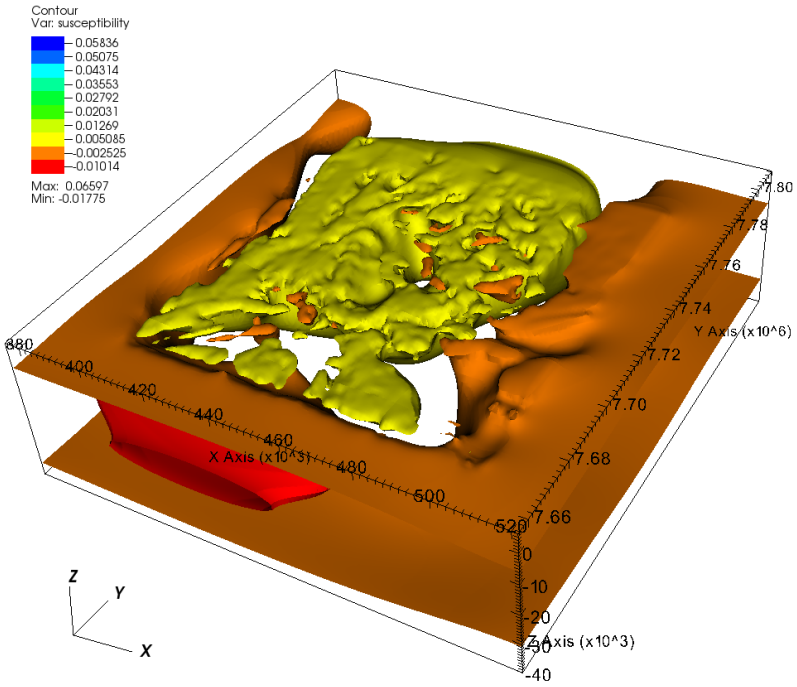


Figure 2.2: Contour plot of the susceptibility from a three-dimensional magnetic inversion (with $\mu = 0.1$). Colours represent values of susceptibility where high values are represented by blue and low values are represented by red.

builder. Step three sets up the inversion and step four runs it. Finally in step five the result is written to the result file, here `result.vtu` in the *VTK* format. Results are shown in Figure 2.2.

Although scripts for magnetic and gravity inversion are largely identical there are a few small differences which we are going to highlight now. The magnetic inversion requires data about the background magnetic flux density over the region of interest which is added to the domain by the statements

```
B_b = [2201.*U.Nano*U.Tesla, 31232.*U.Nano*U.Tesla,  -41405.*U.Nano*U.Tesla]
dom.setBackgroundMagneticFluxDensity(B_b)
```

Here it is assumed that the background magnetic flux density is constant across the domain and is given as the list

```
B_b= [ B_E,  B_N, B_V ]
```

in units of Tesla (T) where `B_N`, `B_E` and `B_V` refer to the north, east and vertical component of the magnetic flux density, respectively. Values for the magnetic flux density can be obtained by the International Geomagnetic Reference Field (IGRF) [3] (or the Australian Geomagnetic Reference Field (AGRF) [9] via `http://www.ga.gov.au/oracle/geomag/agrfform.jsp`). Similar to the gravity case susceptibility below a certain depth can be set to zero via the statement

```
dom.fixSusceptibilityBelow(depth=40.*U.km)
```

where here the susceptibility below $40km$ is prescribed (this has no effect as the depth of the domain is $40km$)[1].

Magnetic data are read and added to the domain with the following statements:

```
source0=NetCdfData(NetCdfData.MAGNETIC, 'MagneticSmall.nc', \
                   scale_factor=U.Nano * U.Tesla)
dom.addSource(source0)
```

The first argument `NetCdfData.MAGNETIC` identifies the data read from file `MagneticSmall.nc` (second argument) as magnetic data.The argument `scale_factor` specifies the units (here nT) of the magnetic flux density data in the file. If scalar data are given it is assumed that the magnetic flux density anomalies are measured in direction of the background magnetic flux density[2].

Finally the inversion is created and run:

```
inv=MagneticInversion()
inv.fixMagneticPotentialAtBottom(False)
k = inv.run()
```

The result for the susceptibility is named `k`. In this case the magnetic potential is not fixed at the bottom of the domain. The magnetic potential is still set zero at the top of the domain.

We then write the result to a *VTK* file using

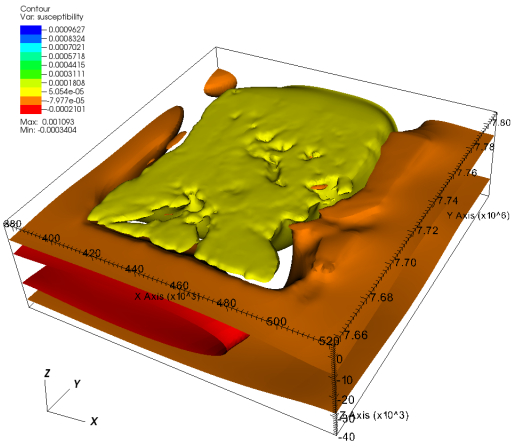
```
saveVTK("result.vtu", susceptibility=k)
```

where the result of the inversion is tagged with the name `susceptibility` as an identifier for the visualization software.

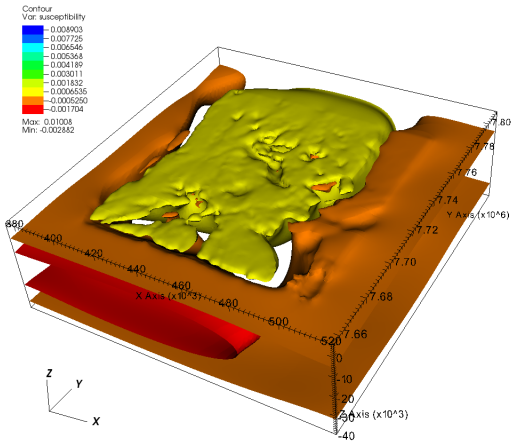
Figures 2.3 and 2.4 show results from the inversion of the magnetic data shown in Figure 2.1. In Figure 2.3 surface contours are used to represent the susceptibility while Figure 2.4 uses contour lines on a lateral plane intercept and two vertical plane intercepts. The images show the strong impact of the trade-off factor μ on the result. Larger values give more emphasis to the misfit term in the cost function leading to rougher susceptibility distributions. The result for $\mu = 0.1$ seems to be the most realistic.

[1] Notice that the method called is different from the one in the case of gravity inversion.

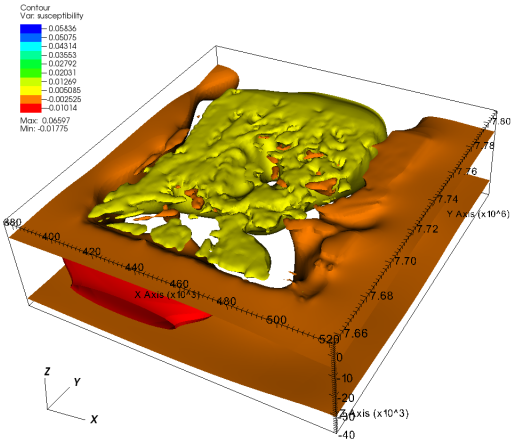
[2] The default for `scale_factor` for magnetic data is nT.



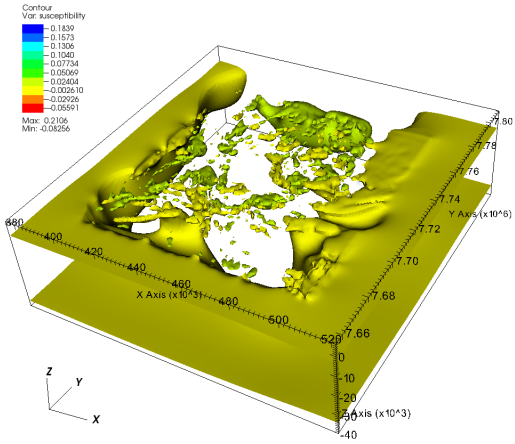
(a) $\mu = 0.001$



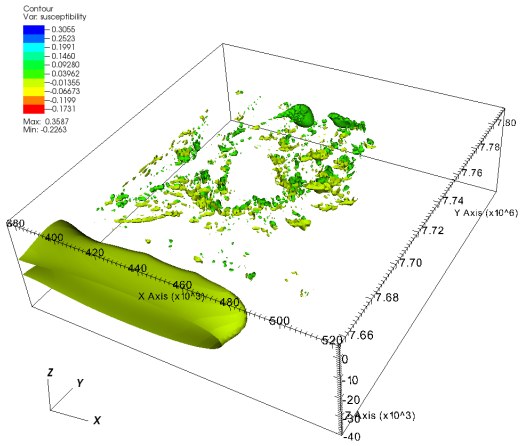
(b) $\mu = 0.01$



(c) $\mu = 0.1$

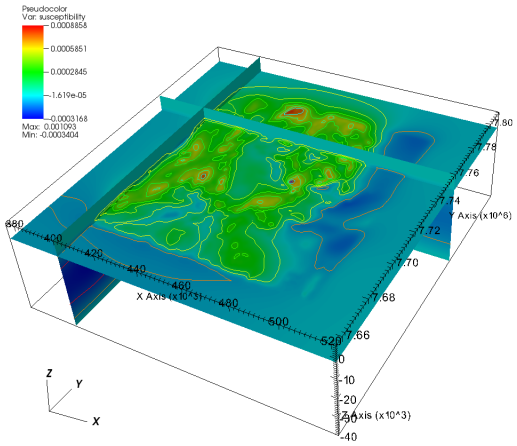


(d) $\mu = 1.$

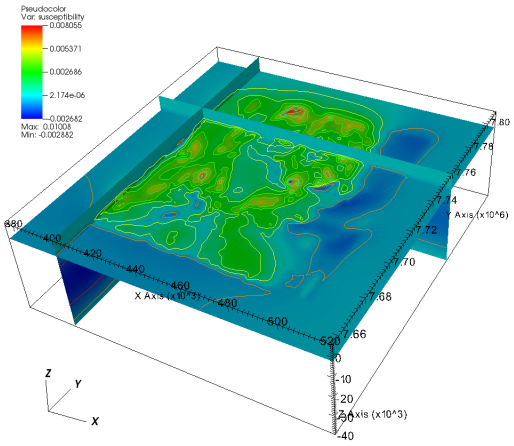


(e) $\mu = 10.$

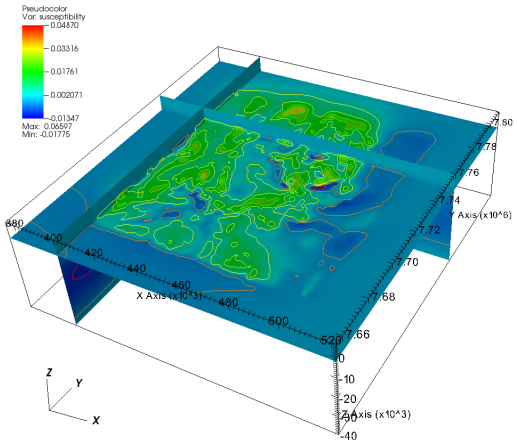
Figure 2.3: 3-D contour plots of magnetic inversion results with data from Figure 2.1 for various values of the model trade-off factor μ. Visualization has been performed in *VisIt*.



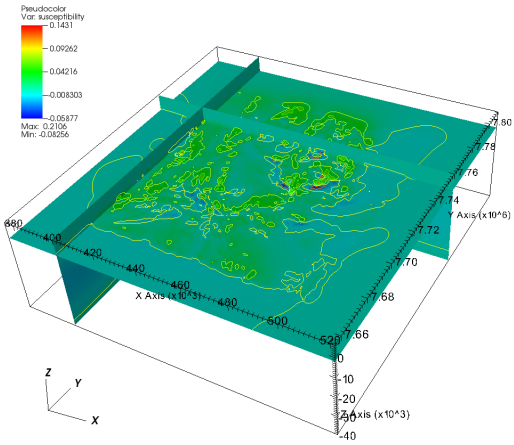
(a) $\mu = 0.001$



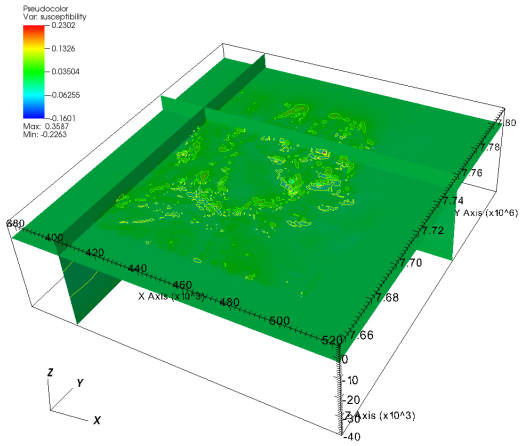
(b) $\mu = 0.01$



(c) $\mu = 0.1$



(d) $\mu = 1.$



(e) $\mu = 10.$

Figure 2.4: 3-D slice plots of magnetic inversion results with data from Figure 2.1 for various values of the model trade-off factor μ. Visualization has been performed *VisIt*.

Part II

Reference Guide

CHAPTER
THREE

Inversion Drivers

Our task in the inversion is to find the geological structure within a given three-dimensional region Ω from given geophysical observations. The structure is described by a *level set function* m. This function can be a scalar function or may have several components, see Chapter 7 for more details. Its values are dimensionless and should be between zero and one. However, the latter condition is not enforced. Through a mapping (see Chapter 8) the values of the level set function are mapped onto physical parameter p^f. The physical parameter feeds into one or more forward models which return a prediction for the observations, see Chapter 9. An inversion may consider several forward models at once which we call *joint inversion*.

The level set function describing the actual geological structure is given as the function which minimizes a particular *cost function* J. This cost function is a composition of the difference of the predicted observations to the actual observations for the relevant forward models, and the regularization term which controls the smoothness of the level set function. In general the cost function J takes the form

$$J(m) = J^{reg}(m) + \sum_f \mu_f^{data} \cdot J^f(p^f) \tag{3.1}$$

where $J^f(p)$ is a measure of the defect of the observations predicted for the parameter p^f against the observations for forward model f, and $J^{reg}(m)$ is the regularization term. The weighting factors μ_f^{data} are dimensionless, non-negative trade-off factors. Potentially, values for the trade-off factors are altered during the inversion process in order to improve the balance between the regularization term and the data defect terms[1]. The physical parameter p^f depends on the level set function m in a known form:

$$p^f = M_f(m) \tag{3.2}$$

where M_f is a given mapping. For the case of gravity inversion the M_f is a simple linear function mapping the level set function m with dimensionless values to physical density anomaly values ρ. (see Chapter 8). In its simplest from the mapping is given as $\rho = \rho_0 \cdot m$ where ρ_0 is a reference density. It is pointed out that the inversion techniques applied do not constrain limits to the values of the level set function although there is the notion that its values are between zero and one. However, limits can be enforced to physical parameters using appropriate mappings.

The level set function m and consequently the physical parameters p^f are defined over a three dimensional domain Ω which represented by an *escript* `Domain` object, see [6]. The domain builder methods provide functions to build appropriate domains from field data sets, see Section 6.2. In general the domain is a rectangular three-dimensional domain where the third dimension $x_2 = z$ represents depth. The $z = 0$ surface defines the surface of the earth where $z < 0$ is defining the subsurface region and $z > 0$ is defining the region above the surface, see Figure 3.2. In general physical parameters such as density and susceptibility anomaly are known above the surface, typically assumed to be zero. For subregions where a physical parameter is known it is assumed that the corresponding level set function as the value zero. If required, non-zero values for the physical parameters can be set using appropriate mapping.

[1] The current version does not support an automated selection of trade-off factors

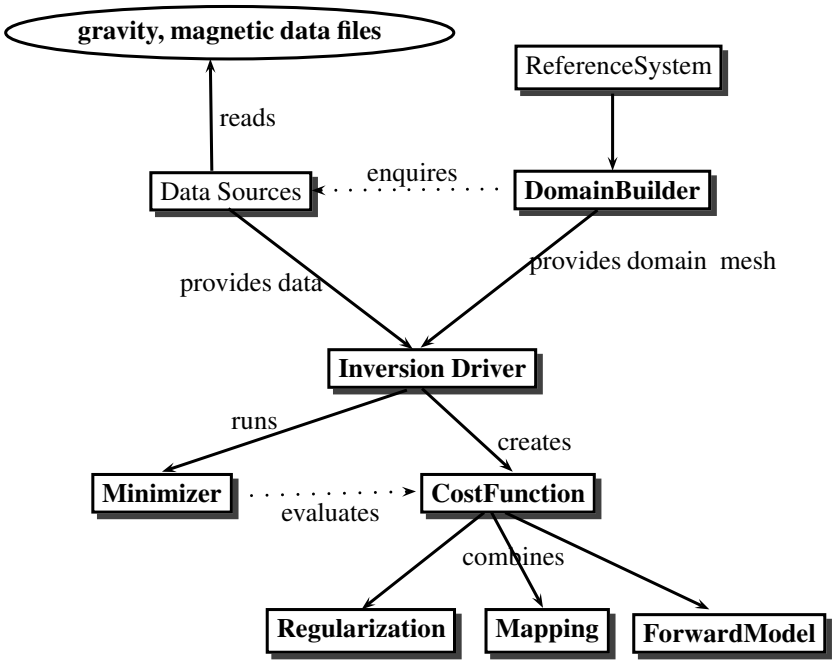


Figure 3.1: Class dependencies

3.1 Class Dependencies

For simplification of usage `esys.downunder` provides predefined classes that drive inversion for particular problems. The usage of this classes is being discussed in Part I. More details are shown in Section 3.3. It is the role of the driver class to orchestration an inversion. New inversions can easily be implemented by modifying the available drivers.

As illustrated in Figure 3.1 the driver class uses geophysical data as managed through the `DataSource` class (see Chapter 6) and an *escript* domain to define an appropriate costs function to be minimized. The driver class also run the minimization solver. The *escript* domain [6] is created using the `DomainBuilder`, see Chapter 6.2, which builds an appropriate domain and mesh based on the geophysical data used in the inversion. Based on the inversion to be performed (gravity, magnetic, joint) the driver class builds an appropriate cost function J including the regularization term J^{reg}, see `Regularization` class in Chapter 7, the forward models, see Chapter 9 and the required mappings, see `Mapping` class in Chapter 8, to connect the level set function with physical parameters. Finally the driver class calls the solver to minimize the cost function, see Chapter 4.

The driver classes cover commonly used cases for the convenience of users. In fact, more general cases can be implemented in an easy way. Script `nodriver.py` is an example on how to implement an inversion without using one of the driver classes.

3.2 Domains

3.2.1 Cartesian Domain

For the Cartesian domain Ω we assume a flat Earth in the form

$$\Omega = [x_0^{min}, x_0^{max}] \times [x_1^{min}, x_1^{max}] \times [x_2^{min}, x_2^{max}] \tag{3.3}$$

and use the Universal Transverse Mercator (UTM) coordinate system[2] where x_0 represents the easting, x_1 the northing and x_2 the altitude. In this way, all three coordinates can be given in meters with minimal distortion when visualizing the domain. The origin in vertical direction (altitude 0) corresponds to sea level. A proper inversion set up requires a buffer zone in all dimensions. Figure 3.2 depicts these as areas shaded in red (padding area) and blue

[2]See e.g. http://en.wikipedia.org/wiki/Universal_Transverse_Mercator_coordinate_system.

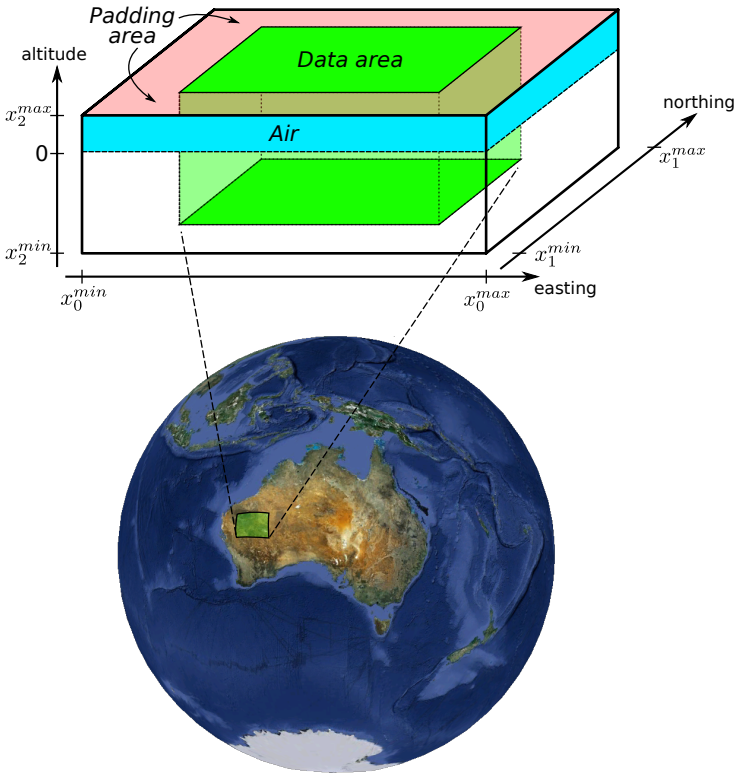


Figure 3.2: Illustration of domain extents, mapping and padding area

(air buffer). While the inversion results contain values for the entire domain the buffer zone should be disregarded when performing any analysis. In other words, only the region labeled *data area* in Figure 3.2 contains useful information. Both the thickness of the air layer and the amount of padding in the x_0/x_1 dimension is configurable when setting up an inversion.

3.3 Driver Classes

The inversion minimizes an appropriate cost function J to find the physical parameter distribution (or more precisely the level set function) which gives the best fit to measured data. A particular inversion case (gravity, magnetic or joint) is managed through an instance of a specialization of the `InversionDriver` class. The task of the class instance is to set up the appropriate cost function, to manage solution parameters and to run the optimization process.

3.3.1 Template

class InversionDriver
: template for inversion drivers.

getCostFunction()
: returns the cost function of the inversion. This will be an instance of the `InversionCostFunction` class, see Section 5. Use this method to access or alter attributes or call methods of the underlying cost function.

getSolver()
: returns the instance of the solver class used to minimize the cost function, see Chapter 4. Use this method to modify solver options.

getDomain()
returns the domain of the inversion as an *escript* `Domain` object.

setSolverMaxIterations([maxiter=*None*])
sets the maximum number of iteration steps for the solver used to minimize the cost function. The default value is 200. If the maximum number is reached, the iteration will be terminated and `MinimizerMaxIterReached` is thrown.

setSolverTolerance([m_tol=*None*] [, J_tol=*None*])
set the tolerance for the solver used to minimize the cost function. If `m_tol` is set the iteration is terminated if the relative change of the level set function is less than or equal `m_tol`, see condition 4.3. If `J_tol` is set the iteration is terminated if the change of the cost function relative to the initial value is less than or equal `J_tol`, , see condition 4.4. If both tolerances are set both stopping criteria need to be met. By default `tol`=1e-4 and `J_tol`=*None*.

getLevelSetFunction()
returns the level set function as solution of the optimization problem. This method can only be called if the optimization process as been completed. If the iteration failed the last available approximation of the solution is returned.

run()
this method runs the optimization solver and returns the physical parameter(s) from the output of the inversion. Notice that the `setup` method must be called before the first call of `run`. The call can fail as the maximum number of iterations is reached in which case a `MinimizerMaxIterReached` exception is thrown or as there is an incurable break down in the iteration in which case a `MinimizerIterationIncurableBreakDown` exception is thrown.

3.3.2 Gravity Inversion Driver

For examples of usage please see Chapter 1.

class GravityInversion([solverclass=None] [, fixGravityPotentialAtBottom=False])
Driver class to perform an inversion of Gravity (Bouguer) anomaly data. This class is a sub-class of `InversionDriver`. The class uses the standard `Regularization` for a single level set function, see Chapter 7, `DensityMapping` mapping, see Section 8.1, and the gravity forward model `GravityModel`, see Section 9.1. `solverclass` set the solver class to be used for inversion, see Chapter 4. By default the limited-memory Broyden-Fletcher-Goldfarb-Shanno (*L-BFGS*) [13] solver is used. If `fixGravityPotentialAtBottom` is set [, fixGravityPotentialAtBottom=False]to *True* the gravity potential at the bottom is set to zero.

fixGravityPotentialAtBottom([status=*True*])
If `status` is *True* the gravity potential at the bottom is set to zero. Otherwise the gravity potential at the top is set to zero only.

setup(domainbuilder [, rho0=*None*] [, drho=*None*] [, z0=*None*] [, beta=*None*] [, w0=*None*] [, w1=*None*] [, rho_at_depth=*None*])
sets up the inversion from an instance `domainbuilder` of a `DomainBuilder`, see Section 6.2. Only gravitational data attached to the `domainbuilder` are considered in the inversion. `rho0` defines a reference density anomaly (default is 0), `drho` defines a density anomaly (default is $2750\frac{kg}{m^3}$), `z0` defines the depth weighting reference depth (default is *None*), and `beta` defines the depth weighting exponent (default is *None*), see `DensityMapping` in Section 8.1. `w0` and `w1` define the weighting factors $\omega^{(0)}$ and $\omega^{(1)}$, respectively (see Equation 7.1). By default `w0`=*None* and `w1`=1 are used. `rho_at_depth` sets the value for density at depth. This is only used if density is fixed below a certain depth, see `Domain Builder` in Section 6.2.

setInitialGuess([rho=*None*])
sets an initial guess for the density anomaly. By default zero is used.

3.3.3 Magnetic Inversion Driver

For examples of usage please see Chapter 2.

class MagneticInversion([solverclass=None])

Driver class to perform an inversion of magnetic anomaly data. This class is a sub-class of `InversionDriver`. The class uses the standard `Regularization` class for a single level set function, see Chapter 7, `SusceptibilityMapping` mapping, see Section 8.2, and the linear magnetic forward model `MagneticModel`, see Section 9.2. `solverclass` set the solver class to be used for inversion, see Chapter 4. By default the limited-memory Broyden-Fletcher-Goldfarb-Shanno (*L-BFGS*) [13] solver is used.

fixMagneticPotentialAtBottom([status=*True*])

If `status` is *True* the magnetic potential at the bottom is set to zero. Otherwise the magnetic potential at the top is set to zero only.

setup(domainbuilder [, k0=*None*] [, dk=*None*] [, z0=*None*] [, beta=*None*] [, w0=*None*] [, w1=*None*] [, k_at_depth=*None*])

sets up the inversion from an instance `domainbuilder` of a `DomainBuilder`, see Section 6.2. Only magnetic data attached to the `domainbuilder` are considered in the inversion. `k0` defines a reference susceptibility anomaly (default is 0), `dk` defines a susceptibility anomaly scale (default is 1), `z0` defines the depth weighting reference depth (default is *None*), and `beta` defines the depth weighting exponent (default is *None*), see `SusceptibilityMapping` in Section 8.2. `w0` and `w1` define the weighting factors $\omega^{(0)}$ and $\omega^{(1)}$, respectively (see equation 7.1). By default `w0`=*None* and `w1`=1 are used. `k_at_depth` sets the value for susceptibility at depth. This is only used if susceptibility is fixed below a certain depth, see `Domain Builder` in Section 6.2.

setInitialGuess([k=*None*])

sets an initial guess for the susceptibility anomaly. By default zero is used.

3.3.4 Gravity and Magnetic Joint Inversion Driver

For examples of usage please see Chapter **??**.

class JointGravityMagneticInversion([solverclass=None])

Driver class to perform a joint inversion of Gravity (Bouguer) and magnetic anomaly data. This class is a sub-class of `InversionDriver`. The class uses the standard `Regularization` for two level set functions with cross-gradient correlation, see Chapter 7, `DensityMapping` and `SusceptibilityMapping` mappings, see Section 8, the gravity forward model `GravityModel`, see Section 9.1 and the linear magnetic forward model `MagneticModel`, see Section 9.2. `solverclass` set the solver class to be used for inversion, see Chapter 4. By default the limited-memory Broyden-Fletcher-Goldfarb-Shanno (*L-BFGS*) [13] solver is used.

fixGravityPotentialAtBottom([status=*True*])

If `status` is *True* the gravity potential at the bottom is set to zero. Otherwise the gravity potential at the top is set to zero only.

fixMagneticPotentialAtBottom([status=*True*])

If `status` is *True* the magnetic potential at the bottom is set to zero. Otherwise the magnetic potential at the top is set to zero only.

setup(domainbuilder [, rho0=*None*] [, drho=*None*] [, rho_z0=*None*] [, rho_beta=*None*] [, k0=*None*] [, dk=*None*] [, k_z0=*None*] [, k_beta=*None*] [, w0=*None*] [, w1=*None*] [, w_gc=*None*] [, rho_at_depth=*None*] [, k_at_depth=*None*])

sets up the inversion from an instance `domainbuilder` of a `DomainBuilder`, see Section 6.2. Gravity and magnetic data attached to the `domainbuilder` are considered in the inversion. `rho0` defines a reference density anomaly (default is 0), `drho` defines a density anomaly (default is $2750 \frac{kg}{m^3}$), `rho_z0` defines the depth weighting reference depth for density (default is *None*), and `rho_beta` defines the depth weighting exponent for

density (default is *None*), see `DensityMapping` in Section 8.1. `k0` defines a reference susceptibility anomaly (default is 0), `dk` defines a susceptibility anomaly scale (default is 1), `k_z0` defines the depth weighting reference depth for susceptibility (default is *None*), and `k_beta` defines the depth weighting exponent for susceptibility (default is *None*), see `SusceptibilityMapping` in Section 8.2. `w0` and `w1` define the weighting factors $\omega^{(0)}$ and $\omega^{(1)}$, respectively (see Equation 7.1). `w_gc` sets the weighting factor $\omega^{(c)}$ for the cross gradient term. By default `w0`=*None*, `w1`=1 and `w_gc`=1 are used. `k_at_depth` sets the value for susceptibility at depth. This is only used if susceptibility is fixed below a certain depth, see `Domain Builder` in Section 6.2. `rho_at_depth` sets the value for density at depth. This is only used if density is fixed below a certain depth, see `Domain Builder` in Section 6.2.

setInitialGuess([rho=*None*,] [k=*None*])

sets initial guesses for density and susceptibility anomaly. By default zero is used for both.

CHAPTER
FOUR

Minimization Algorithms

We need to find the level set function m minimizing the cost function J as defined in Equation 3.1. The physical parameters p^f and the data defects are linked through state variables u^f which is given as a solution of a partial differential equation (PDE) with coefficients depending on p_f. This PDE (or – in case of several forward models – this set of PDEs) defines a constraint in the minimization problem. In the context of our applications it can be assumed that the PDE is of 'maximum rank', i.e. for a given value of the level set function m there is a unique value for the state variables u^f as the solution of the forward model. So from a mathematical point of view the state variable u^f can be eliminated from the problem and the minimization problem becomes in fact a minimization problem for the level set function m alone where the physical parameters which are of most interest in applications can easily be derived from the solution. However one needs to keep in mind that each evaluation of the cost function requires the solution of a PDE (an additional PDE solution is required for the gradient).

In some application cases the optimization problem to be solved defines a quadratic programming problem which would allow to use a special case of solvers. However, for the general scenarios we are interested in here we cannot assume this simplification and need to be able to solve for a general cost function. The method used here is the limited-memory Broyden-Fletcher-Goldfarb-Shanno (*L-BFGS*) method, see [13]. This method is a quasi-Newton method. To implement the method one needs to provide the evaluation of the cost function J and its gradient ∇J and dual product $< \cdot, \cdot >$[1] such that for $\alpha \to 0$

$$J(m + \alpha p) = J(m) + \alpha \cdot < p, \nabla J(m) > + o(\alpha) \tag{4.1}$$

where p is an increment to the level set function[2]. Notice that if m is the unknown solution one has $\nabla J(m) = 0$. Moreover, an approximation of the inverse of the Hessian operator $\nabla\nabla J(m)$ needs to be provided for a given value of m. In the implementation we don't need to provide the entire operator but for a given gradient difference g an approximation $h = Hg$ of $\nabla\nabla J(m))^{-1} g$ needs to be calculated. This is an approximative solution of the equation $\nabla\nabla J(m))h = g$ which in variational form is given as

$$< p, \nabla\nabla J(m)\ h >=< p, g > \tag{4.2}$$

for all level set increments p. In the cases relevant here, it is a possible approach to make the approximation $\nabla\nabla J(m) \approx \nabla\nabla J^{reg}(m)$ which can easily be constructed and the resulting variational Equation 4.2 can easily be solved, see Chapter 7.

L-BFGS is an iterative method which calculates a sequence of level set approximations m_k which converges towards the unknown level set function minimizing the cost function. This iterative process is terminated if certain stopping criteria are fulfilled. A criterion commonly used is

$$\|m_k - m_{k-1}\| \leq \|m_k\| \cdot m\_tol \tag{4.3}$$

in order to terminate the iteration after m_k has been calculated. In this condition $\|.\|$ denotes a norm and m_tol defines a relative tolerance. A typical value for m_tol is 10^{-4}. Alternatively one can check for changes in the cost

[1] A dual product $< \cdot, \cdot >$ defines a function which assigns a pair (p, g) of a level set increment p and a gradient g a real number $< p, g >$. It is linear in the first and linear in the second argument.

[2] o denotes the little o notation see http://en.wikipedia.org/wiki/Big_O_notation

function:

$$| J(m_k) - J(m_k) | \leq | J(m_k) - J(m_0) | \cdot J\_tol \tag{4.4}$$

where J_tol is a given tolerance and m_0 is the initial guess of the solution. As this criterion depends on the initial guess it is not recommended.

4.1 Solver Classes

class MinimizerLBFGS([J=*None*][, m_tol=1e-4][, J_tol=*None*][, imax=300])
constructs an instance of the limited-memory Broyden-Fletcher-Goldfarb-Shanno *L-BFGS* solver. `J` sets the cost function to be minimized, see Section 4.2. If not present the cost function needs to be set using the `setCostFunction` method. `m_tol` and `J_tol` specify the tolerances for the stopping criteria (see description of `setTolerance` below). `imax` sets the maximum number of iteration steps (see `setMaxIterations` below).

setTolerance([m_tol=1e-4][, J_tol=*None*])
sets the tolerances for the stopping criteria. `m_tol` sets the tolerance for the unknown level set Function 4.3. If `m_tol`=*None* the tolerance on level set function is not checked. `J_tol` sets the tolerance for the cost function, see Equation 4.4. If `J_tol`=*None* tolerance on the cost function is not checked (recommended). If `m_tol` and `J_tol` are both set criterion 4.3 and criterion 4.4 are both applied to terminate the iteration.

setMaxIterations(imax)
sets the maximum number of iterations before the minimizer terminates. If the maximum number of iteration is reached a `MinimizerMaxIterReached` exception is thrown.

getResult()
returns the solution of the optimization problem. This method only returns something useful if the optimization process has completed. If the iteration failed the last available approximation of the solution is returned.

getOptions()
returns the solver options as a dictionary which contains all available option names and their current value.

setOptions([truncation=30][, restart=60][, initialHessian=1])
sets solver options. `truncation` defines number of gradients to keep in memory. `restart` defines the number of iteration steps after which the iteration is restarted. `initialHessian` can be used to provide an estimate for the scale of the inverse Hessian. If the cost function provides such an estimate this option is ignored.

run(m0)
runs the solution algorithm and returns an approximation of the solution. `m0` is the initial guess to use. This method may throw an exception if the maximum number of iterations is reached or a solver breakdown occurs.

logSummary()
writes some statistical information to the logger.

4.2 `CostFunction` Class Template

class CostFunction()
template of cost function J

getDualProduct(m, g)
returns the dual product $< m, g >$ of `m` and `g`.

getValue(m, *args)
: returns the value $J(m)$ using pre-calculated values `args` for m.

getGradient(m, *args)
: returns the gradient ∇J of J at m using pre-calculated values `args` for m.

getArguments(m)
: returns pre-calculated values that are shared in the calculation of $J(m)$ and $\nabla J(m)$.

getNorm(m)
: returns the norm $\|m\|$ of $\mathtt{m}$.

updateHessian()
: notifies the class that the Hessian operator needs to be updated in case the class provides an approximation for the inverse of the Hessian.

getInverseHessianApproximation(m, g, *args)
: returns an approximative evaluation p of the inverse of the Hessian operator of the cost function for a given gradient g at location m.

4.3 The L-BFGS Algorithm

The L-BFGS algorithm looks as follows (see also [13]):

Algorithm 4.1 *L-BFGS*

begin
  Input: initial guess x_0 and integer m;
  Allocate m vector pairs $\{s_i, g_i\}$;
  $k \leftarrow 0$;
  $H_0 \leftarrow I$;
  while ¬converged do
    $p_k \leftarrow$ twoLoop$(\{s, g\})$;
    $\alpha_k \leftarrow$ lineSearch(m_k, p_k); See Section 4.3.1
    $m_{k+1} \leftarrow m_k + \alpha_k p_k$;
    if $k > truncation$
      then
        Discard the vector pair $\{s_{k-truncation}, g_{k-truncation}\}$ from storage;
    fi
    $s_k \leftarrow m_{k+1} - m_k$;
    $g_k \leftarrow \nabla J_{k+1} - \nabla J_k$;
    $k \leftarrow k + 1$;
  od
where
funct twoLoop$(\{s, g\})$ $\equiv$
  $\lceil q \leftarrow \nabla J_k$;
  for $i = k-1, k-2, \ldots, k-truncation$ do
    $\rho_i \leftarrow \frac{1}{<s_i, g_i>}$;
    $\alpha_i \leftarrow \rho_i < s_i, q >$;
    $q \leftarrow q - \alpha_i \cdot g_i$;
  od
  $r \leftarrow Hq$;
  for $i = k - truncation, k - truncation + 1, \ldots, k-1$ do
    $\beta \leftarrow \rho_i \cdot < r, g_i >$;
    $r \leftarrow r + s_i \cdot (\alpha_i - \beta)$
  od
  $r \rfloor$.

end

4.3.1 Line Search

The line search procedure minimizes the function $\phi(\alpha)$ for a given level set function m and search direction p, see [14], [11] with

$$\phi(\alpha) = J(m + \alpha \cdot p) \tag{4.5}$$

Notice that

$$\phi'(\alpha) =< p, \nabla J(m + \alpha \cdot p) > \ . \tag{4.6}$$

Algorithm 4.2 *Line Search that satisfies strong Wolfe conditions*

```
begin
  Input: α_max > 0, 0 < c_1 < c_2 < 1;
  i ← 1;
  while 1 do
        if φ(α_i) > φ(0) + c_1 α_i φ'(0) ∨
          (φ(α_i) ≥ φ(α_{i-1}) ∧ i > 1)
          then
                α_* ← zoom(α_{i-1}, α_i);
                break;
        fi
        if ‖φ'(α_i)‖ ≤ −c_2 φ'(0)
          then
                α_* ← α_i;
                break;
        fi
        if φ'(α_i) ≥ 0
          then
                α_* ← zoom(α_i, α_{i-1});
                break;
        fi
        α_{i+1} = 2α_i;
        i ← i + 1;
  od
where
funct zoom(α_lo, α_hi) ≡
  while 1 do
        ⌈α_j ← α_lo + (α_hi − α_lo)/2;
         if φ(α_j) > φ(0) + c_1 α_j φ'(0) ∨ φ(α_j) ≥ φ(α_lo)
           then
                α_hi ← α_j;
          else
                if ‖φ'(α_j)‖ ≤ −c_2 φ'(0)
                  then
                        α_* ← α_j;
                        break;
                fi
                if φ'(α_j)(α_hi − α_lo) ≥ 0
                  then
                        α_hi ← α_lo;
                fi
                α_lo = α_j;
         fi
        od
  ⌋.
end
```

CHAPTER
FIVE

Cost Function

The general form of the cost function minimized in the inversion is given in the form (see also Chapter 3)

$$J(m) = J^{reg}(m) + \sum_f \mu_f^{data} \cdot J^f(p^f) \tag{5.1}$$

where m represents the level set function, J^{reg} is the regularization term, see Chapter 7, and J^f are a set of cost functions for forward models, (see Chapter 9) depending on physical parameters p^f. The physical parameters p^f are known functions of the level set function m which is the unknown to be calculated by the optimization process. μ_f^{data} are trade-off factors. It is pointed out that the regularization term includes additional trade-off factors The `InversionCostFunction` is class to define cost functions of an inversion. It is pointed out that the `InversionCostFunction` class implements the `CostFunction` template class, see Chapter 4.

In the simplest case there is a single forward model using a single physical parameter which is derived form single-values level set function. The following script snippet shows the creation of the `InversionCostFunction` for the case of a gravity inversion:

```
p=DensityMapping(...)
f=GravityModel(...)
J=InversionCostFunction(Regularization(...), \
                        mappings=p, \
                        forward_models=f)
```

The argument `...` refers to an appropriate argument list.

If two forward models are coming into play using two different physical parameters the `mappings` and `forward_models` are defined as lists in the following form:

```
p_rho=DensityMapping(...)
p_k=SusceptibilityMapping(...)
f_mag=MagneticModel(...)
f_grav=GravityModel(...)

J=InversionCostFunction(Regularization(...), \
                        mappings=[p_rho, p_k], \
                        forward_models=[(f_mag, 1), (f_grav,0)])
```

Here we define a joint inversion of gravity and magnetic data. `forward_models` is given as a list of a tuple of a forward model and an index which referring to parameter in the `mappings` list to be used as an input. The magnetic forward model `f_mag` is using the second parameter (=`p_k`) in `mappings` list. In this case the physical parameters are defined by a single-valued level set function. It is also possible to link physical parameters to components of a level set function:

```
p_rho=DensityMapping(...)
p_k=SusceptibilityMapping(...)
```

```
f_mag=MagneticModel(...)
f_grav=GravityModel(...)

J=InversionCostFunction(Regularization(numLevelSets=2,...), \
                        mappings=[(p_rho,0), (p_k,1)], \
                        forward_models=[[(f_mag, 1), (f_grav,0)]])
```

The `mappings` argument is now a list of pairs where the first pair entry specifies the parameter mapping and the second pair entry specifies the index of the component of the level set function to be used to evaluate the parameter. In this case the level set function has two components, where the density mapping uses the first component of the level set function while the susceptibility mapping uses the second component.

5.1 `InversionCostFunction` API

The `InversionCostFunction` implements a `CostFunction` class used to run optimization solvers, see Section 4.2. Its API is defined as follows:

class InversionCostFunction(regularization, mappings, forward_models)
: Constructor for the inversion cost function. `regularization` sets the regularization to be used, see Chapter 7. `mappings` is a list of pairs where each pair comprises of a physical parameter mapping (see Chapter 8) and an index which refers to the component of level set function defined by the `regularization` to be used to calculate the corresponding physical parameter. If the level set function has a single component the index can be omitted. If in addition there is a single physical parameter the mapping can be given instead of a list. `forward_models` is a list of pairs where the first pair component is a forward model (see Chapter 9) and the second pair component refers to the physical parameter in the `mappings` list providing the physical parameter for the model. If a single physical parameter is present the index can be omitted. If in addition a single forward model is used this forward model can be assigned to `forward_models` in replacement of a list.

getDomain()
: returns the *escript* domain of the inversion, see [6].

getNumTradeOffFactors()
: returns the total number of trade-off factors. The count includes the trade-off factors μ_f^{data} for the forward models and (hidden) trade-off factors in the regularization term, see Definition 5.1.

getForwardModel([idx=*None*])
: returns the forward model with index `idx`. If the cost function contains one model only argument `idx` can be omitted.

getRegularization()
: returns the regularization component of the cost function, see `regularization` in Chapter 7.

setTradeOffFactorsModels([mu=*None*])
: sets the trade-off factors μ_f^{data} for the forward model components. If a single model is present `mu` must be a floating point number. Otherwise `mu` must be a list of floating point numbers. It is assumed that all numbers are positive. The default value for all trade-off factors is one.

getTradeOffFactorsModels()
: returns the values of the trade-off factors μ_f^{data} for the forward model components.

setTradeOffFactorsRegularization([mu=*None*], [mu_c=*None*])
: sets the trade-off factors for the regularization component of the cost function. `mu` defines the trade-off factors for the level-set variation part and `mu_c` sets the trade-off factors for the cross-gradient variation part. This method is a shortcut for calling `setTradeOffFactorsForVariation` and `setTradeOffFactorsForCrossGradient` for the underlying the regularization. Please see `Regularization` in Chapter 7 for more details on the arguments `mu` and `mu_c`.

setTradeOffFactors([mu=*None*])

sets the trade-off factors for the forward model and regularization terms. `mu` is a list of positive floats. The length of the list is the total number of trade-off factors given by the method `getNumTradeOffFactors`. The first part of `mu` defines the trade-off factors μ_f^{data} for the forward model components while the remaining entries define the trade-off factors for the regularization components of the cost function. By default all values are set to one.

getProperties(m)

returns the physical properties from a given level set function `m` using the mappings of the cost function. The physical properties are returned in the order in which they are given in the `mappings` argument in the class constructor.

createLevelSetFunction(*props)

returns the level set function corresponding to set of given physical properties. This method is the inverse of the `getProperties` method. The arguments `props` define a tuple of values for the physical properties where the order needs to correspond to the order in which the physical property mappings are given in the `mappings` argument in the class constructor. If a value for a physical property is given as *None* the corresponding component of the returned level set function is set to zero. If no physical properties are given all components of the level set function are set to zero.

getNorm(m)

returns the norm of a level set function `m` as a floating point number.

getArguments(m)

returns pre-computed values for the evaluation of the cost function and its gradient for a given value `m` of the level set function. In essence the method collects pre-computed values for the underlying regularization and forward models[1].

getValue(m[, *args])

returns the value of the cost function for a given level set function `m` and corresponding pre-computed values `args`. If the pre-computed values are not supplied `getArguments` is called.

getGradient(m[, *args])

returns the gradient of the cost function at level set function `m` using the corresponding pre-computed values `args`. If the pre-computed values are not supplied `getArguments` is called. The gradient is represented as a tuple (Y, X) where in essence Y represents the derivative of the cost function kernel with respect to the level set function and X represents the derivative of the cost function kernel with respect to the gradient of the level set function, see Section 5.2 for more details.

getDualProduct(m, g)

returns the dual product of a level set function `m` with a gradient `g`, see Section 5.2 for more details. This method uses the dual product of the regularization.

getInverseHessianApproximation(m, g [, *args])

returns an approximative evaluation of the inverse of the Hessian operator of the cost function for a given gradient `g` at a given level set function `m` using the corresponding pre-computed values `args`. If no pre-computed values are present `getArguments` is called. In the current implementation contributions to the Hessian operator from the forward models are ignored and only contributions from the regularization and cross-gradient term are used.

[1] Using pre-computed values can significantly speed up the optimization process when the value of the cost function and its gradient are needed for the same level set function.

5.2 Gradient calculation

In this section we briefly discuss the calculation of the gradient and the Hessian operator. If ∇ denotes the gradient operator (with respect to the level set function m) the gradient of J is given as

$$\nabla J(m) = \nabla J^{reg}(m) + \sum_f \mu_f^{data} \cdot \nabla J^f(p^f) \ . \tag{5.2}$$

We first focus on the calculation of ∇J^{reg}. In fact the regularization cost function J^{reg} is given through a cost function kernel K^{reg} in the form

$$J^{reg}(m) = \int_\Omega K^{reg} \ dx \tag{5.3}$$

where K^{reg} is a given function of the level set function m_k and its spatial derivative $m_{k,i}$. If n is an increment to the level set function then the directional derivative of J^{ref} in the direction of n is given as

$$< n, \nabla J^{reg}(m) >= \int_\Omega \frac{\partial K^{reg}}{\partial m_k} n_k + \frac{\partial K^{reg}}{\partial m_{k,i}} n_{k,i} \ dx \tag{5.4}$$

where $< ., . >$ denotes the dual product, see Chapter 4. Consequently, the gradient ∇J^{reg} can be represented by a pair of values Y and X

$$\begin{aligned} Y_k &= \frac{\partial K^{reg}}{\partial m_k} \\ X_{ki} &= \frac{\partial K^{reg}}{\partial m_{k,i}} \end{aligned} \tag{5.5}$$

while the dual product $< ., . >$ of a level set increment n and a gradient increment $g = (Y, X)$ is given as

$$< n, g >= \int_\Omega Y_k n_k + X_{ki} n_{k,i} \ dx \tag{5.6}$$

We also need to provide (an approximation of) the value p of the inverse of the Hessian operator $\nabla\nabla J$ for a given gradient increment $g = (Y, X)$. This means we need to (approximatively) solve the variational problem

$$< n, \nabla\nabla J p >= \int_\Omega Y_k n_k + X_{ki} n_{k,i} \ dx \tag{5.7}$$

for all increments n of the level set function. If we ignore contributions from the forward models the left hand side takes the form

$$< n, \nabla\nabla J^{reg} p >= \int_\Omega \frac{\partial Y_k}{\partial m_l} p_l n_k + \frac{\partial Y_k}{\partial m_{l,j}} p_{l,j} n_k + \frac{\partial X_{ki}}{\partial m_l} p_l n_{k,i} + \frac{\partial X_{ki}}{\partial m_{l,j}} p_{l,j} n_{k,i} \ dx \tag{5.8}$$

We follow the concept as outlined in section 5.2. Notice that equation 5.7 defines a system of linear PDEs which is solved using *escript* `LinearPDE` class. In the *escript* notation we need to provide

$$\begin{aligned} A_{kilj} &= \frac{\partial X_{ki}}{\partial m_{l,j}} \\ B_{kil} &= \frac{\partial X_{ki}}{\partial m_l} \\ C_{klj} &= \frac{\partial Y_k}{\partial m_{l,j}} \\ D_{kl} &= \frac{\partial Y_k}{\partial m_l} \end{aligned} \tag{5.9}$$

The calculation of the gradient of the forward model component is more complicated: the data defect J^f for forward model f is expressed using a cost function kernel K^f

$$J^f(p^f) = \int_\Omega K^f \ dx \tag{5.10}$$

In this case the cost function kernel K^f is a function of the physical parameter p^f, which again is a function of the level-set function, and the state variable u_k^f and its gradient $u_{k,i}^f$. For the sake of a simpler presentation the upper index f is dropped.

The gradient $\nabla_p J$ of the J with respect to the physical property p is given as

$$< q, \nabla_p J(p) >= \int_\Omega \frac{\partial K}{\partial u_k}\frac{\partial u_k}{\partial q} + \frac{\partial K}{\partial u_{k,i}}\left(\frac{\partial u_k}{\partial q}\right)_{,i} + \frac{\partial K}{\partial p} q \; dx \tag{5.11}$$

for any q as an increment to the physical parameter p. If the change of the state variable u_f for physical parameter p in the direction of q is denoted as

$$d_k = \frac{\partial u_k}{\partial q} \tag{5.12}$$

equation 5.11 can be written as

$$< q, \nabla_p J(p) >= \int_\Omega \frac{\partial K}{\partial u_k} d_k + \frac{\partial K}{\partial u_{k,i}} d_{k,i} + \frac{\partial K}{\partial p} q \; dx \tag{5.13}$$

The state variable are the solution of PDE which in variational from is given

$$\int_\Omega F_k \cdot r_k + G_{li} \cdot r_{k,i} \; dx = 0 \tag{5.14}$$

for all increments r to the stat u. The functions F and G are given and describe the physical model. They depend of the state variable u_k and its gradient $u_{k,i}$ and the physical parameter p. The change d_k of the state u_f for physical parameter p in the direction of q is given from the equation

$$\int_\Omega \frac{\partial F_k}{\partial u_l} d_l r_k + \frac{\partial F_k}{\partial u_{l,j}} d_{l,j} r_k + \frac{\partial F_k}{\partial p} q r_k + \frac{\partial G_{ki}}{\partial u_l} d_l r_{k,i} + \frac{\partial G_{ki}}{\partial u_{l,j}} d_{l,j} r_{k,i} + \frac{\partial G_{ki}}{\partial p} q r_{k,i} \; dx = 0 \tag{5.15}$$

to be fulfilled for all functions r. Now let d_k^* be the solution of the variational equation

$$\int_\Omega \frac{\partial F_k}{\partial u_l} h_l d_k^* + \frac{\partial F_k}{\partial u_{l,j}} h_{l,j} d_k^* + \frac{\partial G_{ki}}{\partial u_l} h_l d_{k,i}^* + \frac{\partial G_{ki}}{\partial u_{l,j}} h_{l,j} d_{k,i}^* \; dx = \int_\Omega \frac{\partial K}{\partial u_k} h_k + \frac{\partial K}{\partial u_{k,i}} h_{k,i} \; dx \tag{5.16}$$

for all increments h_k to the physical property p. This problem is solved using *escript* `LinearPDE` class. In the *escript* notation we need to provide

$$\begin{aligned} A_{kilj} &= \frac{\partial G_{lj}}{\partial u_{k,i}} \\ B_{kil} &= \frac{\partial F_l}{\partial u_{k,i}} \\ C_{klj} &= \frac{\partial G_{lj}}{\partial u_k} \\ D_{kl} &= \frac{\partial F_l}{\partial u_k} \\ Y_k &= \frac{\partial K}{\partial u_k} \\ X_{ki} &= \frac{\partial K}{\partial u_{k,i}} \end{aligned} \tag{5.17}$$

Notice that these coefficient are transposed to the coefficients used to solve for the state variables in equation 5.14.

Setting $h_l = d_l$ in equation 5.13 and $r_k = d_k^*$ in equation 5.11 one gets

$$\int_\Omega \frac{\partial K}{\partial u_k} d_k + \frac{\partial K}{\partial u_{k,i}} d_{k,i} + \frac{\partial F_k}{\partial p} q d_k^* + \frac{\partial G_{ki}}{\partial p} q d_{k,i}^* \; dx = 0 \tag{5.18}$$

which is inserted into equation 5.13 to get

$$< q, \nabla_p J(p) >= \int_\Omega \left(\frac{\partial K}{\partial p} - \frac{\partial F_k}{\partial p} d_k^* - \frac{\partial G_{ki}}{\partial p} d_{k,i}^* \right) q \; dx \tag{5.19}$$

We need in fact the gradient of J^f with respect to the level set function which is given as

$$< n, \nabla J^f >= \int_\Omega \left(\frac{\partial K^f}{\partial p^f} - \frac{\partial F_k^f}{\partial p^f} d_k^{f*} - \frac{\partial G_{ki}^f}{\partial p^f} d_{k,i}^{f*} \right) \cdot \frac{\partial p^f}{\partial m_l} n_l \; dx \tag{5.20}$$

for any increment n to the level set function. So in summary we get

$$\begin{aligned} Y_k &= \frac{\partial K^{reg}}{\partial m_k} + \sum_f \mu_f^{data} \left(\frac{\partial K^f}{\partial p^f} - \frac{\partial F_l^f}{\partial p^f} d_l^{f*} - \frac{\partial G_{li}^f}{\partial p^f} d_{l,i}^{f*} \right) \cdot \frac{\partial p^f}{\partial m_k} \\ X_{ki} &= \frac{\partial K^{reg}}{\partial m_{k,i}} \end{aligned} \tag{5.21}$$

to represent ∇J as the tuple (Y, X). Contributions of the forward model to the Hessian operator are ignored.

CHAPTER
SIX

Data Sources

At the source of every inversion is data in the form of gravity anomaly or magnetic flux density values for at least a part of the region of interest. These usually come from surveys and are preprocessed to correct for various factors and distortions. This chapter provides an overview of the classes related to data input for inversions.

6.1 Overview

The inversion module comes with a number of classes that can read gridded (raster) data on a 2-dimensional plane from file or provide artificial values for testing purposes. These classes all derive from the abstract `DataSource` class and override methods that return information about the data and the values themselves. The `DomainBuilder` class is responsible for creating an *escript* domain with a suitable grid spacing and spatial extents that include all data sources attached to it (see Figure 6.1). Notice that in the figure there are cells in the region of interest that

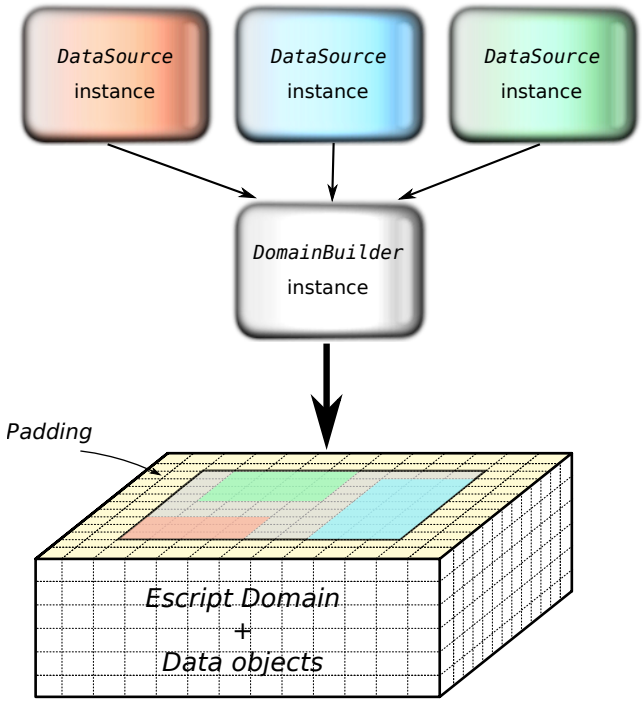


Figure 6.1: `DataSource` instances are added to a `DomainBuilder` which creates a suitable domain and `Data` objects for the inversion

are not covered by any data source instance. Ideally, all data sources used for an inversion have the same spatial resolution and are spatially adjacent so that all cells have a value but this is not a requirement.

6.2 Domain Builder

Every inversion requires one `DomainBuilder` instance which creates and holds a reference to the *escript* domain as well as associated `Data` objects for the input data used for the inversion. The class has the following public methods:

class DomainBuilder([dim=3])
Constructor for the domain builder. `dim` sets the dimensionality of the target domain and must be 2 or 3. By default a 3-dimensional domain is created.

addSource(source)
adds survey data `source` (a `DataSource` object) to the domain builder. The dimensionality of the data must be less than or equal to the domain dimensionality.

setVerticalExtents([depth=40000.][, air_layer=10000.][, num_cells=25])
sets the parameters for the vertical dimension of the domain. The parameter `depth` specifies the thickness in meters of the subsurface layer ($-x_2^{min}$ in Figure 3.2). The default value of 40 km is usually appropriate. Similarly, the `air_layer` parameter defines the buffer zone thickness above the surface (x_2^{max} in Figure 3.2) which should be a few kilometres to avoid artefacts in the inversion. The number of elements (or cells) in the vertical dimension is set with the `num_cells` parameter. Consider the size and resolution of your datasets, the total vertical length (=`depth`+`air_layer`) and available compute resources when setting this value.

setFractionalPadding([pad_x=*None*][, pad_y=*None*])
sets the amount of padding around the dataset as a fraction of the dataset side lengths if the reference coordinate system is Cartesian. For example, calling `setFractionalPadding(0.2, 0.1)` with a data source of size 10000×20000 meters will result in the padded data set size 14000×24000 meters (that is $10000 \times (1 + 2 \times 0.2)$ and $20000 \times (1 + 2 \times 0.1)$). By default no padding is applied and `pad_y` is ignored for 2-dimensional domains.

setFractionalPadding([pad_lat=*None*][, pad_lon=*None*])
sets the amount of padding around the dataset as a fraction of the dataset side lengths if the reference coordinate system is not Cartesian. For example, calling `setFractionalPadding(0.2, 0.1)` with a data source of size 10×20 degree will result in the padded data set size 14×24 degree (that is $10 \times (1 + 2 \times 0.2)$ and $20 \times (1 + 2 \times 0.1)$). By default no padding is applied and `pad_lon` is ignored for 2-dimensional domains.

setPadding([pad_x=*None*][, pad_y=*None*])
sets the amount of padding around the dataset in absolute length units. The final domain size will be the length in x (in y) of the dataset plus twice the value of `pad_x` (`pad_y`). The arguments must be non-negative. By default no padding is applied and `pad_y` is ignored for 2-dimensional domains. This function can be used for Cartesian reference coordinate system only.

setElementPadding([pad_x=*None*][, pad_y=*None*])
sets the amount of padding around the dataset in number of elements (cells), if the reference coordinate system is Cartesian. When the domain is constructed `pad_x` (`pad_y`) elements are added on each side of the x- (y-) dimension in case of a Cartesian reference system. The arguments must be non-negative integers. By default no padding is applied and `pad_y` is ignored for 2-dimensional domains.

setElementPadding([pad_lat=*None*][, pad_lon=*None*])
sets the amount of padding around the dataset in number of elements (cells) if the the reference coordinate system is not Cartesian. When the domain is constructed `pad_lat` (`pad_lon`) elements are added on each side of the latitudinal (longitudinal) dimension. The arguments must be non-negative integers. By default no padding is applied and `pad_lon` is ignored for 2-dimensional domains.

fixDensityBelow([depth=*None*])
defines the depth below which the density anomaly is fixed to zero. This method is only useful for inversions that involve gravity data.

fixSusceptibilityBelow([depth=*None*])
defines the depth below which the susceptibility anomaly is fixed to zero. This method is only useful for inversions that involve magnetic data.

getGravitySurveys()
returns a list of all gravity surveys added to the domain builder. See `getSurveys()` for more details.

getMagneticSurveys()
returns a list of all magnetic surveys added to the domain builder. See `getSurveys()` for more details.

getSurveys(datatype)
returns a list of surveys of type `datatype` available to this domain builder. In the current implementation each survey is a tuple of two `Data` objects, the first containing anomaly values and the second standard error values for the survey.

getDomain()
returns an *escript* domain (see [6]) suitable for running inversions on the attached data sources. The first time this method is called the target parameters (such as resolution, extents and number of elements) are computed, and the domain is created. Subsequent calls return the same domain instance so calls to `setPadding()`, `addSource()` and other methods that influence the domain will fail once `getDomain()` is called the first time.

setBackgroundMagneticFluxDensity(B)
sets the background magnetic flux density $B = (B_{North}, B_{East}, B_{Vertical})$ which is required for magnetic inversions. B_{East} is ignored for 2-dimensional magnetic inversions.

getBackgroundMagneticFluxDensity()
returns the background magnetic flux density B set via `setBackgroundMagneticFluxDensity()` in a form suitable for the inversion. There should be no need to call this method directly.

getSetDensityMask()
returns the density mask `Data` object which is non-zero for cells that have a fixed density value, zero otherwise. There should be no need to call this method directly.

getSetSusceptibilityMask()
returns the susceptibility mask `Data` object which is non-zero for cells that have a fixed susceptibility value, zero otherwise. There should be no need to call this method directly.

6.3 `DataSource` Class

Data sources added to a `DomainBuilder` must provide an implementation for a few methods as described in the class template `DataSource` from the `esys.downunder.datasources` module:

class DataSource()
Base constructor which initializes members and should therefore be invoked by subclasses. Subclasses may then use the member `logger` to print any output.

getDataExtents()
This method should be implemented to return a tuple of tuples ((x0, y0), (nx, ny), (dx, dy)), where (x0, y0) denote the UTM coordinates of the data origin, (nx, ny) the number of data points, and (dx, dy) the spacing of data points in a Cartesian reference system.

getDataType()

Subclasses must return `DataSource.GRAVITY` or `DataSource.MAGNETIC` depending on the type of data they provide.

getSurveyData(domain, origin, NE, spacing)
This method is called by the `DomainBuilder` to retrieve the actual survey data in the form of `Data` objects on the `domain`. Data sources are responsible to map or interpolate their data onto the domain which has been constructed to fit the data. The domain `origin`, number of elements `NE` and element `spacing` are provided as tuples or lists to aid with interpolation.

getUtmZone()
Must be implemented to return the UTM zone that corresponds to the location of this data set as returned by `getDataExtents`.

setSubsamplingFactor(factor)
Notifies the data source that data should be subsampled by `factor`. This method does not need to be overwritten. See `getSubsamplingFactor()` for an explanation.

getSubsamplingFactor()
Returns the subsampling factor which was set by `setSubsamplingFactor()` or 1 which indicates that no subsampling is requested. Data sources that support subsampling (or interleaving) of their data should use this method to query the subsampling factor before returning surveys via `getSurveyData`. If supported, the factor should be applied in all dimensions. For example, a 2-dimensional dataset with 300 x 150 data points should be reduced to 150 x 75 data points when the subsampling factor equals 2. Subsampling becomes important when the survey data resolution is too fine or when using data with varying resolution in one inversion. Note that data sources may choose to ignore the subsampling factor if they don't support it.

The `esys.downunder.datasources` module contains the following helper functions:

LatLonToUTM(longitude, latitude[, wkt_string=*None*])
converts one or more (longitude,latitude) pairs to the corresponding (x,y) coordinates in the *Universal Transverse Mercator* (UTM) projection. This function requires the `pyproj` module for conversion and the `gdal` module to parse the `wkt_string` parameter if supplied. The `wkt_string` parameter may describe the coordinate system used for the input values as a *Well-known Text* (WKT) string.

6.3.1 ER Mapper Raster Data

ER Mapper files that contain 2-dimensional raster data may be used for inversions through the `ErMapperData` class which is derived from `DataSource`. Date are given in latitudinal and longitudinal coordinates and if a Cartesian reference coordinate system is used are mapped using the appropriate (UTM) projection. Generally, these datasets contain two files [2], a header file and a data file. The former usually has the `.ers` file extension and is a text file that describes the data format, size, coordinate system used etc. The data file usually has the same file name but no extension. Note, that the current implementation may not work with all *ER Mapper* datasets. For example, the only cell type understood is *IEEE4ByteReal* at the moment. To run inversions on a *ER Mapper* dataset use the following constructor:

class ErMapperData(data_type, headerfile[, datafile=*None*][, altitude=0.][, error=*None*][, scale_factor=*None*][, null_value=*None*])
Creates a new data source from *ER Mapper* data. The parameter `data_type` must be one of `DataSource.GRAVITY` or `DataSource.MAGNETIC` depending on the type of data, `headerfile` is the name of the header file while `datafile` specifies the name of the data file. The parameter `datafile` can be left blank if the name is identical to the header file except for the file extension. The `altitude` parameter can be used to shift a 2-dimensional slice of data vertically within a 3-dimensional domain. Use `error` to set the (constant) measurement error with the same units used by the measurements. By default a value of 2 units is assumed which equals $0.2\ mgal$ or $2\ nT$ depending on the data type. Since ER Mapper files do not store any information about data units or scale the `scale_factor` may be used to provide this information. If not set, gravity data is assumed to be given

in $\frac{\mu m}{sec^2}$ while magnetic data is assumed to be given in nT. Finally, the `null_value` parameter can be used to override the value for the areas to be ignored (see Section 1.7.2) which is usually provided in the ER Mapper header file.

6.3.2 NetCDF Data

The `NetCdfData` class from the `esys.downunder.datasources` module provides the means to use data from *netCDF* files [12] for inversion. Currently, files that follow the *Climate and Forecast (CF)*[1] and/or the *Cooperative Ocean/Atmosphere Research Data Service (COARDS)*[2] metadata conventions are supported. The example script `create_netcdf.py` demonstrates how a compatible file can be generated from within *python* (provided the `scipy` module is available). To plot such an input file including coordinates and legend using *matplotlib* [8] see the script `show_netcdf.py`. The interface to `NetCdfData` looks as follows:

class NetCdfData(datatype, filename[, altitude=0.][, data_variable=*None*][, error=*None*][, scale_factor=*None*][, null_value=*None*])

Creates a new data source from compatible *netCDF* data. The two required parameters are `datatype`, which must be one of `DataSource.GRAVITY` or `DataSource.MAGNETIC` depending on the type of data, and `filename` which is the name of the file containing the data. The `altitude` parameter can be used to shift a 2-dimensional slice of data vertically within a 3-dimensional domain. Set the `data_variable` parameter to the name of the *netCDF* variable that contains the measurements unless there is only one data variable in the file in which case the parameter can be left empty. Use `error` to set the (constant) measurement error with the same units used by the measurements or the name of the *netCDF* variable that contains this information. By default a value of 2 units is assumed which equals $0.2\ mgal$ or $2\ nT$ depending on the data type. The current implementation does not use the units attribute (if available) within the *netCDF* file. Use the `scale_factor` argument to provide this information instead. If not set, gravity data is assumed to be given in $\frac{\mu m}{sec^2}$ while magnetic data is assumed to be given in nT. Finally, the `null_value` parameter can be used to override the value for the areas to be ignored (see Section 1.7.2) which is usually provided in the *netCDF* file.

6.3.3 Synthetic Data

As a special case the `esys.downunder.datasources` module contains classes to generate input data for inversions by solving a forward model with user-defined reference data. The main purpose of using synthetic data is to test the capabilities of the inversion module or for tracking down problems.

The base class for synthetic data which is derived from `DataSource` has the following interface:

class SyntheticDataBase(datatype[, DIM=2] [, number_of_elements=10] [, length=1*U.km] [, B_b=*None*] [, data_offset=0] [, full_knowledge=*False*])

Base class to define reference data based on a given property distribution (density or susceptibility). Data are collected from a square region of vertical extent `length` on a grid with `number_of_elements` cells in each direction. The synthetic data are constructed by solving the appropriate forward problem. Data can be sampled with an offset from the surface at $z = 0$ or using the entire subsurface region.

The only additional method which needs to be implemented in subclasses is

getReferenceProperty([domain=*None*])

Returns the reference `Data` object that was used to generate the gravity or susceptibility anomaly data. The `domain` argument must be present when this method is called for the first time but not necessarily in subsequent calls.

Two synthetic data providers are currently available. The class `SyntheticData` defines synthetic gravity or magnetic anomaly data based on a harmonic

$$k = A \cdot sin\left(\pi \frac{n_D}{D}(z + \Delta z)\right) \cdot sin\left(\pi \frac{n_L}{L}(x - x_0)\right) \tag{6.1}$$

[1] http://cf-pcmdi.llnl.gov/documents/cf-conventions/latest-cf-conventions-document-1
[2] http://ferret.wrc.noaa.gov/noaa_coop/coop_cdf_profile.html

where A is the amplitude, n_D, n_L denote the number of oscillations in the vertical and lateral direction, respectively, while D and L is the depth and side length for the data, respectively. The second data provider class is `SyntheticFeatureData` which takes a list of `SourceFeature` objects that define the anomaly and is thus more generic. The constructors are defined as follows:

class SyntheticData(datatype[, n_length=1] [, n_depth=1] [, depth_offset=0.] [, depth=*None*] [, amplitude=*None*] [, DIM=2] [, number_of_elements=10] [, length=1*U.km] [, B_b=*None*] [, data_offset=0] [, full_knowledge=*False*] [, spherical=*False*])

The arguments `n_length`, `n_depth`, `depth_offset`, `depth`, and `amplitude` correspond to the respective symbols in Equation 6.1. The remaining arguments are passed to the parent class (`SyntheticDataBase`) and described there. If `depth` is not set the depth of the domain is used. The argument `amplitude` may be left unset as well in which case a default value is used ($200\frac{kg}{m^3}$ for gravity and 0.1 for magnetic data).

class SyntheticFeatureData(datatype, features[, DIM=2] [, number_of_elements=10] [, length=1*U.km] [, B_b=*None*] [, data_offset=0] [, full_knowledge=*False*] [, spherical=*False*])

The only new argument is `features` which is a list of `SourceFeature` objects to be included in the data preparation. All other arguments are passed on to the parent class (see there).

class SourceFeature()

A feature adds a density/susceptibility distribution to (parts of) a domain of a synthetic data source, for example a layer of a specific rock type or a simulated ore body. This base class is empty and only provides the skeleton for subclasses which need to implement the following two methods:

getValue()

Returns the value for the area covered by mask. It can be constant or a `Data` object with spatial dependency.

getMask(x)

Returns the mask of the region of interest for this feature. That is, mask is non-zero where the value returned by `getValue()` should be applied, zero elsewhere.

CHAPTER
SEVEN

Regularization

The general cost function J^{total} to be minimized has some of the cost function J^f measuring the defect of the result from the forward model with the data, and the cost function J^{reg} introducing the regularization into the problem and makes sure that a unique answer exists. The regularization term is a function of, possibly vector-valued, level set function m which represents the physical properties to be represented and is, from a mathematical point of view, the unknown of the inversion problem. It is the intention that the values of m are between zero and one and that actual physical values are created from a mapping before being fed into a forward model. In general the cost function J^{reg} is defined as

$$J^{reg}(m) = \frac{1}{2}\int_\Omega \left(\sum_k \mu_k \cdot (\omega_k^{(0)} \cdot m_k^2 + \omega_{ki}^{(1)} m_{k,i}^2) + \sum_{l<k} \mu_{lk}^{(c)} \cdot \omega_{lk}^{(c)} \cdot \chi(m_l, m_k) \right) \, dx \tag{7.1}$$

where summation over i is performed. The additional trade–off factors μ_k and $\mu_{lk}^{(c)}$ $(l < k)$ are between zero and one and constant across the domain. They are potentially modified during the inversion in order to improve the balance between the different terms in the cost function.

χ is a given symmetric, non-negative cross-gradient function. We use

$$\chi(a, b) = (a_{,i}a_{,i}) \cdot (b_{,j}b_{,j}) - (a_{,i}b_{,i})^2 \tag{7.2}$$

where summations over i and j are performed, see [4]. Notice that cross-gradient function is measuring the angle between the surface normals of contours of level set functions. So minimizing the cost function will align the surface normals of the contours.

The coefficients $\omega_k^{(0)}$, $\omega_{ki}^{(1)}$ and $\omega_{lk}^{(c)}$ define weighting factors which may depend on their location within the domain. We assume that for given level set function k the weighting factors $\omega_k^{(0)}$, $\omega_{ki}^{(1)}$ are scaled such that

$$\int_\Omega (\omega_k^{(0)} + \frac{\omega_{ki}^{(1)}}{L_i^2}) \, dx = \alpha_k \tag{7.3}$$

where α_k defines the scale which is typically set to one. L_i is the width of the domain in x_i direction. Similarly we set for $l < k$ we set

$$\int_\Omega \frac{\omega_{lk}^{(c)}}{L^4} \, dx = \alpha_{lk}^{(c)} \tag{7.4}$$

where $\alpha_{lk}^{(c)}$ defines the scale which is typically set to one and

$$\frac{1}{L^2} = \sum_i \frac{1}{L_i^2} \, . \tag{7.5}$$

In some cases values for the level set functions are known to be zero at certain regions in the domain. Typically this is the region above the surface of the Earths. This expressed using a a characteristic function q which varies with its location within the domain. The function q is set to zero except for those locations x within the domain

where the values of the level set functions is known to be zero. For these locations x q takes a positive value. for a single level set function one has

$$q(x) = \begin{cases} 1 & \text{if } m \text{ is set to zero at location } x \\ 0 & \text{otherwise} \end{cases} \tag{7.6}$$

For multi-valued level set function the characteristic function is set componentwise:

$$q_k(x) = \begin{cases} 1 & \text{if component } m_k \text{ is set to zero at location } x \\ 0 & \text{otherwise} \end{cases} \tag{7.7}$$

7.1 Usage

class Regularization(domain [, w0=*None*] [, w1=*None*] [, wc=*None*] [, location_of_set_m=Data()] [, numLevelSets=1] [, useDiagonalHessianApproximation=*False*] [, tol=1e-8] [, scale=*None*] [, scale_c=*None*])

initializes a regularization component of the cost function for inversion. `domain` defines the domain of the inversion. `numLevelSets` sets the number of level set functions to be found during the inversion. `w0`, `w1` and `wc` define the weighting factors $\omega^{(0)}$, $\omega^{(1)}$ and $\omega^{(c)}$, respectively. A value for `w0` or `w1` or both must be given. If more than one level set function is involved `wc` must be given. `location_of_set_m` sets the characteristic function q to define locations where the level set function is set to zero, see equation (7.6). `scale` and `scale_c` set the scales α_k in equation (7.3) and $\alpha_{lk}^{(c)}$ in equation (7.4), respectively. By default, their values are set to one. Notice that weighting factors are rescaled to meet the scaling conditions. `tol` sets the tolerance for the calculation of the Hessian approximation. `useDiagonalHessianApproximation` indicates to ignore coupling in the Hessian approximation produced by the cross-gradient term. This can speed-up an individual iteration step in the inversion but typically leads to more inversion steps.

7.2 Gradient Calculation

The cost function kernel is given as

$$K^{reg}(m) = \frac{1}{2}\sum_k \mu_k \cdot (\omega_k^{(0)} \cdot m_k^2 + \omega_{ki}^{(1)} m_{k,i}^2) + \sum_{l<k} \mu_{lk}^{(c)} \cdot \omega_{lk}^{(c)} \cdot \chi(m_l, m_k) \tag{7.8}$$

We need to provide the gradient of the cost function J^{reg} with respect to the level set functions m. The gradient is represented by two functions Y and X which define the derivative of the cost function kernel with respect to m and to the gradient $m_{,i}$, respectively:

$$\begin{aligned} Y_k &= \frac{\partial K^{reg}}{\partial m_k} \\ X_{ki} &= \frac{\partial K^{reg}}{\partial m_{k,i}} \end{aligned} \tag{7.9}$$

For the case of a single valued level set function m we get

$$Y = \mu \cdot \omega^{(0)} \cdot m \tag{7.10}$$

and

$$X_i = \mu \cdot \omega_i^{(1)} \cdot m_{,i} \tag{7.11}$$

For a two-valued level set function (m_0, m_1) we have

$$Y_k = \mu_k \cdot \omega_k^{(0)} \cdot m_k \text{ for } k = 0, 1 \tag{7.12}$$

and for X

$$\begin{aligned} X_{0i} &= \mu_0 \cdot \omega_{0i}^{(1)} \cdot m_{0,i} + \mu_{01}^{(c)} \cdot \omega_{01}^{(c)} \cdot ((m_{1,j} m_{1,j}) \cdot m_{0,i} - (m_{1,j} m_{0,j}) \cdot m_{1,i}) \\ X_{1i} &= \mu_1 \cdot \omega_{1i}^{(1)} \cdot m_{1,i} + \mu_{01}^{(c)} \cdot \omega_{01}^{(c)} \cdot ((m_{0,j} m_{0,j}) \cdot m_{1,i} - (m_{1,j} m_{0,j}) \cdot m_{0,i}) \end{aligned} \tag{7.13}$$

We also need to provide an approximation of the inverse of the Hessian operator as discussed in section 5.2. For the case of a single valued level set function m we get

$$\begin{aligned} A_{ij} &= \mu \cdot \omega_i^{(1)} \cdot \delta_{ij} \\ D &= \mu \cdot \omega^{(0)} \end{aligned} \tag{7.14}$$

For a two-valued level set function (m_0, m_1) we have

$$D_{kl} = \mu_k \cdot \omega_k^{(0)} \cdot \delta_{kl} \tag{7.15}$$

and

$$\begin{aligned} A_{0i0j} &= \mu_0 \cdot \omega_{0i}^{(1)} \cdot \delta_{ij} + \mu_{01}^{(c)} \cdot \omega_{01}^{(c)} \cdot ((m_{1,j'} m_{1,j'}) \cdot \delta_{ij} - m_{1,i} \cdot m_{1,j}) \\ A_{0i1j} &= \mu_{01}^{(c)} \cdot \omega_{01}^{(c)} \cdot (2 \cdot m_{0,i} \cdot m_{1,j} - m_{1,i} \cdot m_{0,j} - (m_{1,j'} m_{0,j'}) \cdot \delta_{ij}) \\ A_{1i0j} &= \mu_{01}^{(c)} \cdot \omega_{01}^{(c)} \cdot (2 \cdot m_{1,i} \cdot m_{0,j} - m_{0,i} \cdot m_{1,j} - (m_{1,j'} m_{0,j'}) \cdot \delta_{ij}) \\ A_{1i1j} &= \mu_1 \cdot \omega_{1i}^{(1)} \cdot \delta_{ij} + \mu_{01}^{(c)} \cdot \omega_{01}^{(c)} \cdot ((m_{0,j'} m_{0,j'}) \cdot \delta_{ij} - m_{0,i} \cdot m_{0,j})) \end{aligned} \tag{7.16}$$

CHAPTER
EIGHT

Mapping

Mapping classes map a level set function m as described in Chapter 7 onto a physical parameter such as density and susceptibility.

8.1 Density Map

For density we use the form

$$\rho = \rho_0 + \Delta\rho \cdot \left(\frac{z_0 - x_2}{l_z} \right)^{\frac{\beta}{2}} \cdot m \tag{8.1}$$

where ρ_0 is the reference density, $\Delta\rho$ is the density scaling, z_0 an offset, l_z vertical expansion of the domain and β is a suitable exponent.

class DensityMapping(domain [, z0=None] [, rho0=0] [, drho=$2750 \cdot kg \cdot m^{-3}$ **] [, beta=2.])**
a linear density mapping including depth weighting. `domain` is the domain of the inversion, `z0` reference depth in the depth weighting factor, `drho` is the density scaling factor (by default the density of granite is used) and `beta` is the exponent in the depth weighting factor. If no reference depth `z0` is given no depth weighting is applied. `rho0` is the reference density which may be a function of its location in the domain.

getValue(m)
returns the density for level set function m

getDerivative(m)
return the derivative of density with respect to the level set function.

getInverse(p)
returns the value level set function m for given density value p.

8.2 Susceptibility Map

For the magnetic susceptibility k the following mapping is used:

$$k = k_0 + \Delta k \cdot \left(\frac{z_0 - x_2}{l_z} \right)^{\frac{\beta}{2}} \cdot m \tag{8.2}$$

where k_0 is the reference density and Δk is the density scaling.

class SusceptibilityMapping(domain [, z0=None] [, k0=0] [, dk=1] [, beta=2.])
a linear susceptibility mapping including depth weighting. `domain` is the domain of the inversion, `z0` reference depth in the depth weighting factor, `dk` is the susceptibility scaling factor (by default one is

used) and `beta` is the exponent in the depth weighting factor. If no reference depth `z0` is given no depth weighting is applied. `k0` is the reference susceptibility which may be a function of its location in the domain.

getValue(m)
returns the susceptibility for level set function m

getDerivative(m)
return the derivative of susceptibility with respect to the level set function.

getInverse(p)
returns the value level set function m for given susceptibility value p.

8.3 General Mapping Class

Users can define their own mapping $p = \Psi(m)$. The following interface needs to be served

class Mapping()
mapping of a level set function onto a physical parameter to be used by a forward model.

getValue(m)
returns the result $\Psi(m)$ of the mapping for level set function m

getDerivative(m)
return the derivative $\frac{\partial \Psi}{\partial m}$ of the mapping with respect to the level set function for the level set function m.

getInverse(p)
returns the value level set function m for given value p of the physical parameter, ie $p = \Psi(m)$.

CHAPTER
NINE

Forward Models

9.1 Gravity Inversion

For the gravity inversion we use the anomaly of the gravity acceleration of the Earth. The controlling material parameter is the density ρ of the rock. If the density field ρ is known the gravitational potential ψ is given as the solution of the PDE

$$-\psi_{,ii} = -4\pi G \cdot \rho \tag{9.1}$$

where $G = 6.6730 \cdot 10^{-11} \frac{m^3}{kg \cdot s^2}$ is the gravitational constant. The gravitational potential is set to zero at the top of the domain Γ_0. On all other faces the normal component of the gravity acceleration anomaly g_i is set to zero, i.e. $n_i \psi_{,i} = 0$ with outer normal field n_i. The gravity force g_i is given as the negative of the gradient of the gravity potential ψ:

$$g_i = -\psi_{,i} \tag{9.2}$$

From the gravitational potential we can calculate the gravity acceleration anomaly via Equation (9.2) to obtain the defect to the given data. If $g_i^{(s)}$ is a measurement of the gravity acceleration anomaly for survey s and $\omega_i^{(s)}$ is a weighting factor the data defect $J^{grav}(k)$ in the notation of Chapter 5 is given as

$$J^{grav}(k) = \frac{1}{2} \sum_s \int_\Omega (\omega_i^{(s)} \cdot (g_i - g_i^{(s)}))^2 dx \tag{9.3}$$

Summation over i is performed. The cost function kernel is given as

$$K^{grav}(\psi_{,i}, k) = \frac{1}{2} \sum_s (\omega_i^{(s)} \cdot (\psi_{,i} + g_i^{(s)}))^2 \tag{9.4}$$

In practice the gravity acceleration $g^{(s)}$ is measured in vertical direction z with a standard error deviation $\sigma^{(s)}$ at certain locations in the domain. In this case one sets the weighting factors $\omega^{(s)}$ as

$$\omega_i^{(s)} = \begin{cases} f \cdot \frac{\delta_{iz}}{\sigma^{(s)}} & \text{data are available} \\ \text{where} & \\ 0 & \text{otherwise} \end{cases} \tag{9.5}$$

With the objective to control the gradient of the cost function the scaling factor f is chosen in the way that

$$\sum_s \int_\Omega (\omega_i^{(s)} g_i^{(s)}) \cdot (\omega_j^{(s)} \frac{1}{L_j}) \cdot 4\pi G L^2 \cdot \rho' \, dx = \alpha \tag{9.6}$$

where α defines a scaling factor which is typically set to one and L is defined by equation (7.5). ρ' is considering the derivative of the density with respect to the level set function.

9.1.1 Usage

class GravityModel(domain, w, g, [, fixPotentialAtBottom=False], [, tol=1e-8])

opens a gravity forward model over the `Domain` `domain` with weighting factors `w` ($= \omega^{(s)}$) and measured gravity acceleration anomalies `g` ($= g^{(s)}$). The weighting factors and the measured gravity acceleration anomalies must be vectors where components refer to the components (x_0, x_1, x_2) for the Cartesian coordinate system. `tol` set the tolerance for the solution of the PDE (9.1). If `fixPotentialAtBottom` is set to *True*, the gravitational potential at the bottom is set to zero in addition to the potential on the top.

rescaleWeights([scale=1.] [rho_scale=1.])

rescale the weighting factors such condition (9.6) holds where `scale` sets the scale α and `rho_scale` sets ρ'. This method should be called before any inversion is started in order to make sure that all components of the cost function are appropriately scaled.

9.1.2 Gradient Calculation

This section briefly explains how the gradient $\frac{\partial J^{grav}}{\partial \rho}$ of the cost function J^{grav} with respect to the density ρ is calculated. We follow the concept as outlined in section 5.2. The gravity potential ψ from PDE (9.1) is solved in weak form:

$$\int_\Omega q_{,i}\psi_{,i}\ dx = -\int_\Omega 4\pi G \cdot q\rho\ dx \tag{9.7}$$

for all q with $q = 0$ on Γ_0. In the following we set $\Psi[\cdot] = \psi$ for a given density $\cdot$ as solution of the variational problem (9.7). If Γ_ρ denotes the region of the domain where the density is known and for a given direction p with $p = 0$ on Γ_ρ one has

$$\int_\Omega \frac{\partial J^{grav}}{\partial \rho} \cdot p\ dx = \int_\Omega \sum_s (\omega_j^{(s)} \cdot (g_j^{(s)} - g_j)) \cdot (\omega_i^{(s)} \Psi[p]_{,i})\ dx \tag{9.8}$$

with

$$Y_i[\psi] = \sum_s (\omega_j^{(s)} \cdot (g_j^{(s)} - g_j)) \cdot \omega_i^{(s)} \tag{9.9}$$

This is written as

$$\int_\Omega \frac{\partial J^{grav}}{\partial \rho} \cdot p\ dx = \int_\Omega Y_i[\psi]\Psi[p]_{,i}\ dx \tag{9.10}$$

We then set $Y^*[\psi]$ as the solution of the equation

$$\int_\Omega r_{,i}Y^*[\psi]_{,i}\ dx = \int_\Omega r_{,i}, Y_i[\psi]\ dx \text{ for all } p \text{ with } r = 0 \text{ on } \Gamma_{top} \tag{9.11}$$

with $Y^*[\psi] = 0$ on Γ_0. With $r = \Psi[p]$ we get

$$\int_\Omega \Psi[p]_{,i}Y^*[\psi]_{,i}\ dx = \int_\Omega \Psi[p]_{,i}, Y_i[\psi]\ dx \tag{9.12}$$

and from Equation (9.7) with $q = Y^*[\psi]$ we get

$$\int_\Omega Y^*[\psi]_{,i}\Psi[p]_{,i}\ dx = -\int_\Omega 4\pi G \cdot Y^*[\psi] \cdot p\ dx \tag{9.13}$$

which leads to

$$\int_\Omega \Psi[p]_{,i}, Y_i[\psi]\ dx = -\int_\Omega 4\pi G \cdot Y^*[\psi] \cdot p\ dx \tag{9.14}$$

and finally

$$\int_\Omega \frac{\partial J^{grav}}{\partial \rho} \cdot p\ dx = -\int_\Omega 4\pi G \cdot Y^*[\psi] \cdot p\ dx \tag{9.15}$$

or

$$\frac{\partial J^{grav}}{\partial \rho} = -4\pi G \cdot Y^*[\psi] \tag{9.16}$$

9.2 Linear Magnetic Inversion

For the magnetic inversion we use the anomaly of the magnetic flux density of the Earth. The controlling material parameter is the susceptibility k of the rock. With magnetization M and inducing magnetic field anomaly H^s, the magnetic flux density anomaly B^s is given as

$$B_i = \mu_0 \cdot (H_i^s + M_i) \tag{9.17}$$

where $\mu_0 = 4\pi \cdot 10^{-7} \frac{Vs}{Am}$. In this forward model we make the simplifying assumption that the magnetization is proportional to the known geomagnetic flux density B^b:

$$\mu_0 \cdot M_i = k \cdot B_i^b \ . \tag{9.18}$$

Values for the magnetic flux density can be obtained by the International Geomagnetic Reference Field (IGRF) [3] (or the Australian Geomagnetic Reference Field (AGRF) [9]). In most cases it is reasonable to assume that that the background field is constant across the domain.

The magnetic field anomaly H^s can be represented by the gradient of a magnetic scalar potential ψ. We use the form

$$\mu_0 \cdot H_i^s = -\psi_{,i} \tag{9.19}$$

With this notation one gets from Equations (9.17) and (9.18):

$$B_i = -\psi_{,i} + k \cdot B_i^b \tag{9.20}$$

As the B^s magnetic flux density anomaly we obtain the PDE

$$-\psi_{,ii} = -(kB_i^r)_{,i} \tag{9.21}$$

with $B_i^r = B_i^b$ which needs to be solved for a given susceptibility k. The magnetic scalar potential is set to zero at the top of the domain Γ_0. On all other faces the normal component of the magnetic flux density anomaly B_i is set to zero, i.e. $n_i\psi_{,i} = k \cdot n_i B_i^b$ with outer normal field n_i.

From the magnetic scalar potential we can calculate the magnetic flux density anomaly via Equation (9.21) to calculate the defect to the given data. If $B_i^{(s)}$ is a measurement of the magnetic flux density anomaly for survey s and $\omega_i^{(s)}$ is a weighting factor the data defect $J^{mag}(k)$ in the notation of Chapter 5 is given as

$$J^{mag}(k) = \frac{1}{2} \sum_s \int_\Omega (\omega_i^{(s)} \cdot (B_i - B_i^{(s)}))^2 dx \tag{9.22}$$

Summation over i is performed. The cost function kernel is given as

$$K^{mag}(\psi_{,i}, k) = \frac{1}{2} \sum_s (\omega_i^{(s)} \cdot (k \cdot B_i^b - \psi_{,i} - B_i^{(s)}))^2 \tag{9.23}$$

Notice that if magnetic flux density is measured in air one can ignore the $k \cdot B_i^b$ as the susceptibility is zero.

In practice the magnetic flux density $b^{(s)}$ is measured along a certain direction $d_i^{(s)}$ with a standard error deviation $\sigma^{(s)}$ at certain locations in the domain. In this case one sets $B_i^{(s)} = b^{(s)} \cdot d_i^{(s)}$ and the weighting factors $\omega^{(s)}$ as

$$\omega_i^{(s)} = \begin{cases} f \cdot \frac{d_i^{(s)}}{\sigma^{(s)}} & \text{data are available} \\ \text{where} & \\ 0 & \text{otherwise} \end{cases} \tag{9.24}$$

where it is assumed that $d_i^{(s)} \cdot d_i^{(s)} = 1$. With the objective to control the gradient of the cost function the scaling factor f is chosen in the way that

$$\sum_s \int_\Omega (\omega_i^{(s)} B_i^{(s)}) \cdot (\omega_j^{(s)} \frac{1}{L_j}) \cdot L^2 \cdot (B_n^b \frac{1}{L_n}) \cdot k' \ dx = \alpha \tag{9.25}$$

where α defines a scaling factor which is typically set to one and L is defined by equation (7.5). k' is considering the derivative of the density with respect to the level set function.

9.2.1 Usage

class MagneticModel(domain, w, B, background_field, [, fixPotentialAtBottom=False], [, tol=1e-8],)
opens a magnetic forward model over the `Domain` `domain` with weighting factors `w` ($= \omega^{(s)}$) and measured magnetic flux density anomalies `B` ($= B^{(s)}$). The weighting factors and the measured magnetic flux density anomalies must be vectors. `background_field` defines the background magnetic flux density B^b as a vector with north, east and vertical components. `tol` sets the tolerance for the solution of the PDE (9.21). If `fixPotentialAtBottom` is set to *True*, the gravitational potential at the bottom is set to zero in addition to the potential on the top.

rescaleWeights([scale=1.] [k_scale=1.])
rescale the weighting factors such condition (9.25) holds where `scale` sets the scale α and `k_scale` sets k'. This method should be called before any inversion is started in order to make sure that all components of the cost function are appropriately scaled.

9.2.2 Gradient Calculation

This section briefly explains how the gradient $\frac{\partial J^{mag}}{\partial k}$ of the cost function J^{mag} with respect to the susceptibility k is calculated. We follow the concept as outlined in section 5.2.

The magnetic potential ψ from PDE (9.21) is solved in weak form:

$$\int_\Omega q_{,i}\psi_{,i}\ dx = \int_\Omega k \cdot q_{,i}B_i^r\ dx \tag{9.26}$$

for all q with $q = 0$ on Γ_0. In the following we set $\Psi[k] = \psi$ for a given susceptibility k as solution of the variational problem (9.26). If Γ_k denotes the region of the domain where the susceptibility is known and for a given direction p with $p = 0$ on Γ_k one has

$$\int_\Omega \frac{\partial J^{mag}}{\partial k} \cdot p\ dx = \int_\Omega \sum_s (\omega_j^{(s)}(B_j^{(s)} - B_j)) \cdot (\omega_i^{(s)}(\Psi[p]_{,i} - p \cdot B_i^b))\ dx \tag{9.27}$$

with

$$Y_i[\psi] = \sum_s (\omega_j^{(s)}(B_j^{(s)} - B_j)) \cdot \omega_i^{(s)} \tag{9.28}$$

This is written as

$$\int_\Omega \frac{\partial J^{mag}}{\partial k} \cdot p\ dx = \int_\Omega Y_i[\psi]\Psi[p]_{,i} - p \cdot Y_i[\psi]B_i^b\ dx \tag{9.29}$$

We then set adjoint function $Y^*[\psi]$ as the solution of the equation

$$\int_\Omega r_{,i}Y^*[\psi]_{,i}\ dx = \int_\Omega r_{,i}, Y_i[\psi]\ dx \text{ for all } p \text{ with } r = 0 \text{ on } \Gamma_0 \tag{9.30}$$

with $Y^*[\psi] = 0$ on Γ_0. With $r = \Psi[p]$ we get

$$\int_\Omega \Psi[p]_{,i}Y^*[\psi]_{,i}\ dx = \int_\Omega \Psi[p]_{,i}Y_i[\psi]\ dx \tag{9.31}$$

and from Equation (9.26) with $q = Y^*[\psi]$ we get

$$\int_\Omega Y^*[\psi]_{,i}\Psi[p]_{,i}\ dx = \int_\Omega p \cdot Y^*[\psi]_{,i}B_i^r\ dx \tag{9.32}$$

which leads to

$$\int_\Omega \Psi[p]_{,i}, Y_i[\psi]\ dx = \int_\Omega p \cdot Y^*[\psi]_{,i}B_i^r\ dx \tag{9.33}$$

and finally

$$\int_\Omega \frac{\partial J^{mag}}{\partial k} \cdot p\ dx = \int_\Omega p \cdot (Y^*[\psi]_{,i}B_i^r - Y_i[\psi]B_i^b)\ dx \tag{9.34}$$

or

$$\frac{\partial J^{mag}}{\partial k} = Y^*[\psi]_{,i}B_i^r - Y_i[\psi]B_i^b \tag{9.35}$$

Index

Bibliography

[1] L. Gross (eds.). *Documentation for esys.escript*. The Univeristy of Queensland, Australia, 2013.

[2] erdas. *ERMapper Customization Guide*. ERDAS, Inc., 5051 Peachtree Corners Circle, Suite 100, Norcross, GA 30092-2500 USA, 2008.

[3] C. C. Finlay, S. Maus, C. D. Beggan, T. N. Bondar, A. Chambodut, T. A. Chernova, A. Chulliat, V. P. Golovkov, B. Hamilton, M. Hamoudi, R. Holme, G. Hulot, W. Kuang, B. Langlais, V. Lesur, F. J. Lowes, H. Lhr, S. Macmillan, M. Mandea, S. McLean, C. Manoj, M. Menvielle, I. Michaelis, N. Olsen, J. Rauberg, M. Rother, T. J. Sabaka, A. Tangborn, L. Tffner-Clausen, E. Thbault, A. W. P. Thomson, I. Wardinski, Z. Wei, and T. I. Zvereva. International geomagnetic reference field: the eleventh generation. *Geophysical Journal International*, 183(3):1216–1230, 2010.

[4] Luis A Gallardo, Max A Meju, and Marco A Prez-Flores. A quadratic programming approach for joint image reconstruction: mathematical and geophysical examples. *Inverse Problems*, 21(2):435, 2005.

[5] Paradigm GOCAD homepage. http://www.pdgm.com/Products/GOCAD.

[6] Lutz Gross, Cihan Altinay, Artak Amirbekyan, Joel Fenwick, Louise M. Olsen-Kettle, Ken Steube, Leon Graham, and Hans B Muhlhaus. *esys-Escript Users Guide: Solving Partial Differential Equations with Escript and Finley. Release - 3.2*. The University of Queensland, Australia, 2010.

[7] Antony Hallam, Lutz Gross, Cihan Altinay, Artak Amirbekyan, Artak, Joel Fenwick, and Lin Gao. *The escript cookbook: Release - 3.2 (r3422)*. The Univeristy of Queensland, Australia, 2010.

[8] J. D. Hunter. Matplotlib: A 2d graphics environment. *Computing In Science & Engineering*, 9(3):90–95, 2007.

[9] W. Jones, P. Crosthwaite, A. Hitchman, A. Lewis, and L. Wang. Geoscience australia's geomagnetism program – assisting geophysical exploration. *ASEG Extended Abstracts*, 2012(1):1–3, 2012.

[10] *Mayavi2: The next generation scientific data visualization*, 2009.

[11] Jorge J. More and David J. Thuente. Line search algorithms with guaranteed sufficient decrease. *ACM Trans. Math. Software*, 20:286–307, 1992.

[12] netCDF homepage. http://www.unidata.ucar.edu/software/netcdf.

[13] Jorge Nocedal. Updating quasi-Newton matrices with limited storage. *Math. Comp.*, 35(151):773–782, 1980.

[14] Jorge Nocedal and Stephen J. Wright. *Numerical Optimization*. Springer, 2nd edition, 2006.

[15] Silo homepage. https://wci.llnl.gov/codes/silo.

[16] W.M. Telford, L.P. Geldart, R.E. Sheriff, and D.A Keys. *Applied Geophysics*. Cambridge University Press, 2nd edition, 1990.

[17] VisIt homepage. https://wci.llnl.gov/codes/visit/home.html.