1 Introduction

Measurement of the gravity field through satellites has progressed greatly in recent years. The latest of such satellite missions is the European Space Agency's Gravity Field and Steady-State Ocean Circulation Explorer (GOCE). Data from these satellites provides us with an unprecedentedly uniform global coverage. These new data products enable large scale interpretations, as exemplified by the recent article in the journal Nature by Panet et al. (2014).

Regional or global scale interpretations require forward modeling and inversion methods that can take the curvature of the Earth into account. As of this writing, I am aware of only one gravity inversion method developed for spherical coordinates (Chaves and Ussami, 2013). This means that there is much room to develop new modeling methods in spherical coordinates, and adapt and improve upon existing methods.

For the purposes of this text, I'll focus on two main inversion schemes: linear inversion for a 3D density distribution, and non-linear inversion for the relief of an interface. The non-linear inversion is useful for mapping the relief of a geologic interface, like the crystalline basement of sedimentary basins and the Mohorovičić discontinuity (also know as the Moho). The linear 3D inversion is useful for investigating the three-dimensional shape geologic features, like intrusive bodies, possible mantle plumes, magmatic underplating, suture zones, volcanic magma chambers, etc.

Many inversions are performed in the frequency domain (Farquharson and Oldenburg, 1998; Wieczorek and Phillips, 1998)). The advantage of frequency domain inversion is the increased speed of computations. The disadvantage is that it is that regularization is implicit (usually done by band-pass filtering). This makes it more difficult to control the constraints that are inserted, like similarity with previous results, compactness, sharp variations in the model (e.g., faults and contacts), etc. Space-domain inversions are usually formulated using prisms and allow better use of regularization techniques (Barbosa et al., 1999; Martins et al., 2011; Portniaguine and Zhdanov, 1999; Silva Dias et al., 2009; Li and Oldenburg, 1998). However, space-domain inversions pose challenges in the computational front. The main bottlenecks are computing the sensitivity matrix and solving the linear systems. Different algorithms have been developed to mitigate these problems: wavelet compression (Li and Oldenburg, 2010), octree meshes (Davis and Li, 2013), moving footprint (Cox et al., 2010), the planting method of inversion (Uieda and Barbosa, 2012), etc.

A lot of computational infrastructure is needed to implement these inversion methods:

- Reliable forward modeling.
- A software framework for solving regularized inverse problems.
- A way to deal with large computational loads.
- Plotting of the results.

In this project, we propose to build this computational infrastructure. We will use this infrastructure to develop 3D inversion methods in spherical coordinates using tesseroids. We will also investigate novel algorithms for regularization, parametrization, and ways to decrease the computational load.

2 Forward modeling

Forward modeling will be done using tesseroids (Figure 1). However, there are no analytic formulas for the computation of the gravitational potential and its derivatives of a tesseroid. These computations can performed numerically by two main methods: Taylor series expansion (Heck and Seitz, 2007) and Gauss-Legendre Quadrature (Asgharzadeh et al., 2007). We chose the Gauss-Legendre Quadrature (GLQ) for its better numerical accuracy (Wild-Pfeiffer, 2008) and used the simplified formulation by Grombein et al. (2013).

As with most numerical methods, the degree of accuracy of the GLQ integration varies with the discretization. In general, the accuracy increases with the number of discretization points. Setting the discretization for a desired accuracy can be done automatically by using recursively dividing the tesseroids into smaller ones (Li et al., 2011). Making the tesseroids smaller effectively decreases the space between discretization points and thus raises accuracy. However, the error committed during GLQ integration using the division scheme has not been quantified. In this project, we quantified the error involved in GLQ integration by comparing the tesseroid computations with the analytic formulas

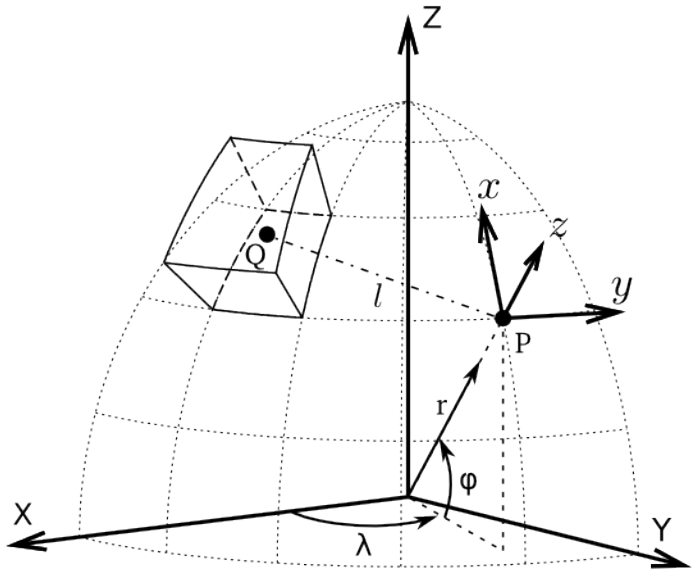


Figure 1: A tesseroid in a geocentric coordinate system.

for the half-sphere. We found an optimum trade-off between computation time and accuracy to achieve a modeling error bellow GOCE accuracy. We have implemented the forward modeling using an improved version of the recursive division scheme in two open-source software packages: Tesseroids and Fatiando a Terra.

Tesseroids (http://www.leouieda.com/tesseroids) is written in the ANSI C language. It employs a Unix style command-line interface with many component programs that do different tasks. These components can be combined in a shell script to produce a final result. Tesseroids implements the recursive division scheme using recursive function calls. This has the advantage of a straight forward implementation. However, it makes error handling more difficult and is subject to difficult-to-find bugs, like stack overflows.

Fatiando a Terra (http://www.fatiando.org) is a geophysical modeling and inversion library written in the Python language. It serves a wide range of applications, mainly in potential field methods and seismology. The recursive division scheme implemented in Fatiando a Terra uses a stack, or LIFO (last-in-first-out), algorithm to avoid recursive function calls. This is more complex to implement but has the advantage of being faster and more robust. Our results are consistent across implementations.

3 A framework for inverse problems

3.1 Mathematical background

We will follow the formulation of geophysical inverse problems adopted by traditional text books, like Aster et al. (2013) and Scales et al. (2001). We begin by defining a vector of observed data $\mathbf{d}^o$ and a function $f_i(\mathbf{p})$ that predicts the value of the ith observation given a vector of parameters $\mathbf{p}$. These predictions are grouped in a predicted data vector $\mathbf{d}$.

The inverse problem of estimating $\mathbf{p}$ from a set of observations $\mathbf{d}^o$ can be solved through least-squares estimation. This means finding $\mathbf{p}$ that minimizes

$$\Theta(\mathbf{p}) = ||\mathbf{r}||_2^2 = \mathbf{r}^T\mathbf{r}$$

in which $\mathbf{r} = \mathbf{d}^o - \mathbf{d}$ is the residual vector.

In geophysics, this parameter estimation problem is usually ill-posed and required the addition of prior information. The traditional way of doing this is through regularization. Instead of minimizing $\Theta(\mathbf{p})$, we minimize the objective function

$$\Gamma(\mathbf{p}) = \Theta(\mathbf{p}) + \mu\Phi(\mathbf{p})$$

in which $\Phi(\mathbf{p})$ is the regularizing function and μ is the regularization parameter that controls the trade-off between fitting the data and conforming to prior information. Many forms of regularization exist in the literature. The most widely used is Tikhonov regularization (damping and smoothness constraints). Different forms of regularization have been proposed that impose sharp transitions on the estimated parameters (Martins et al., 2011; Portniaguine and Zhdanov, 1999; Fomel, 2007).

Once the objective function is formulated, the inverse problem becomes an optimization problem. If function $f_i(\mathbf{p})$ and the regularizing function are both linear, the minimum can estimated directly. For non-linear cases, the minimum can be found through gradient descent methods or heurisict methods. Heuristic methods only require evaluating the objective function for different realizations of the parameter vector. However, the number of function evaluations required is generally very large and computation time for heuristic methods tends to be long. Gradient descent methods are generally faster to compute but require knowledge of the gradient vector (and sometimes Hessian matrix) of the objective function.

3.2 Software implementation

Making a software to solve an inverse problem involves the implementation of:

- forward modeling
- regularizing function
- building the required matrices
- an optimization method

Many of these can be automated and reused between different inversion methods. Forward modeling can have several applications, for example forward modeling with tesseroids can be used to terrain correction, non-linear inversion, and linear inversion. Conventional regularizing functions (damping, smoothness, total variation) are usually problem independent. Gradient descent optimization methods are also problem independent and require only knowledge of the Hessian matrix, gradient vector, and value of the goal function for different parameter vectors. Heuristic optimization methods require only knowledge of the value of the goal function.

This large re-usability potential is prime for building an application programing interface (API). An API implements the components that are common among different inversions. When implementing a new inversion method, the user of the API uses these components to build the new functionality. This minimizes the amount of code one needs to write for a new problem. Optimally, the API allows the user to try different combinations of methods of optimization and regularization with little or no effort. In short, the goals we set for an inverse problems API are:

1. Provide a consistent interface for executing the inversions.
2. Enable writing code that corresponds to the mathematics.
3. Automate the process of implementing a new inverse problem.
4. Allow code reuse and recombination requiring as little new code as possible.

We have built such an API in the open-source software package Fatiando a Terra. Fatiando already contains a range of forward modeling routines. It also implements the forward modeling with tesseroids discussed above). Fatiando a Terra is implemented in the Python language and takes advantage of the large number of scientific libraries available, notably:

- NumPy: an array processing library used for dense matrices and linear algebra operations.

- SciPy: collections of functions built on top of NumPy. Offers a range of utilities, from sparse matrices to signal processing.
- matplotlib: the standard for 2D plotting library in Python.
- Mayavi: a Python interface to the VTK 3D visualization library.

The new inverse problems API in Fatiando was designed with an object oriented (OO) approach. It is largely inspired on the design of the machine learning library scikit-learn. The main classes in the API are `Objective` and `Misfit`. `Objective` represents an objective function, i.e., a scalar function of the parameter vector p. All other classes in the API are derived from `Objective`. `Objective` has three main methods: `Objective.value`, `Objective.gradient`, and `Objective.hessian`. These methods take a parameter vector as an argument and return the value, gradient vector, and Hessian matrix, respectively, of the objective function. However, how to calculate these three quantities is problem specific and has to implemented by the class that derives from `Objective`.

`Objective` also has methods that estimate the parameter vector that minimizes it:

- `Objective.newton`: Newton's method.
- `Objective.steepest`: Steepest descent.
- `Objective.levmarq`: Newton's method with the Levemberg-Marquardt algorithm.
- `Objective.acor`: Ant Colony Optimization for Continuous Domains.

`Misfit` is derived from `Objective` and represents an ℓ_2-norm data misfit function. In addition to the methods of `Objective`, `Misfit` adds the following methods:

- `Misfit.predicted`: returns the predicted data vector.
- `Misfit.residuals`: returns the residual vector.
- `Misfit.jacobian`: returns the Jacobian (sensitivity) matrix.
- `Misfit.fit`: estimates the minimum of the data-misfit function using the configured optimization method.
- `Misfit.config`: configures the optimization method used by `fit`.

The `predicted` and `jacobian` methods are the only ones that are problem specific. Thus, implementing a new inversion is only a matter of creating a derivative class of `Misfit` and implementing these two methods (see section "Sample implementation of non-linear 2D gravity inversion").

`Misfit.fit` and `Misfit.config` are the main methods used across all inverse problems in Fatiando a Terra. This provides a consistent interface for all inversion methods: first create a data misfit object, then call `fit` to run the inversion. The solution (parameter vector that minimizes the objective function) is stored in the `Misfit.p_` attribute. The attribute `Misfit.estimate_` stores an optionally formatted version of `Misfit.p_` and it's use is preferred. The formatted solution can be used to convert the parameter vector into a more useful object, like a geometric shape representation, velocity instead of slowness for tomography problems, etc.

`Misfit` and `Objective` support the binary addition (+) and multiplication (*) operators. Addition of two or more `Misfit` or `Objective` instances creates a new object whose `value`, `gradient`, and `hessian` methods return the sum of the corresponding methods for each component of the addition. Multiplication is only supported for a scalar (floating point or integer) and behaves in the same way as addition. These operations can be combined to form a new objective function that is the weighted sum of other objective functions. For example, the objective function

$$\Gamma(\mathbf{p}) = \theta(\mathbf{p}) + 0.1\phi(\mathbf{p})$$

could be created by the following code:

```
gamma = Theta(...) + 0.1*Phi(...)
```

in which `Theta` and `Phi` are classes derived from `Objective` or `Misfit`. The inversion can then be run in the same way as for a single objective function: by calling `gamma.fit()` or `gamma.config(...).fit()`.

The addition operator can also be used with more than two objects, allowing the use of multiple regularizing functions and joint inversion. For the moment, joint inversion is limited only to problems that share the same parameter vector (like gravity and gravity gradients both use density).

Fatiando a Terra provides a few ready-to-use regularizing functions:

- `Damping`: damping or Tikhonov order zero regularization.
- `Smoothness1D`, `Smoothness2D`, and `Smoothness3D`: smoothness or Tikhonov order one regularization for 1D, 2D, and 3D problems.
- `TotalVariation1D`, `TotalVariation2D`, and `TotalVariation3D`: sharpness or total variation regularization for 1D, 2D, and 3D problems.

These classes are all derivatives of `Objective` and can be used without any modification. For example, damping regularization can be used in any inverse problem as:

```
gamma = Theta(...) + 0.001*Damping(number_of_parameters)
gamma.fit()
```

The patter of creating an instance of a derivative of `Misfit` or `Objective` and then calling `fit` is maintained throughout Fatiando a Terra. An example of this is the `LCurve` class, that runs the inversion and finds the optimal regularization parameter μ that falls in the "elbow" of the data-misfit x regularizing function curve. This class is used as follows:

```
misfit = Theta(...)
regul = Damping(misfit.nparams)
solver = LCurve(misfit, regul, [0.000001, 0.001, 0.1, 10, 1000])
solver.fit()
```

Calling `solver.fit()` runs the inversion using the values of μ specified by the third argument to `LCurve`, creates an L-curve, and automatically finds it's "elbow" point. The optimal value for μ is stored in `solver.regul_param_`. The solution obtained using the optimal μ is stored in `solver.p_` and `solver.estimate_`. Notice that `LCurve` runs several inversions in the background but maintains the same interface as a standard inversion with a fixed μ.

3.3 Sample implementation of 2D non-linear gravity inversion

To exemplify the use of the inverse problems API in Fatiando a Terra, we'll implement here a 2D gravity inversion for the relief of a sedimentary basin. Examples of simpler problems can be found on the online documentation of Fatiando a Terra (http://fatiando.readthedocs.org/en/latest/api/inversion.html).

Problem setup

Sedimentary basins generally have a smaller density than their crystalline basement and surrounding rocks. For this reason, they cause a gravity anomaly. We will assume that the basin is much larger in the y direction, making it approximately 2D. The vertical cross-section of such a basin can be modeled using a polygon. Talwani et al. (1959) provide the formulas necessary to calculate the gravitational effect 2D bodies with polygonal vertical cross-sections.

In our inversion, we'll create a polygon with N vertices. The first will be fixed at the surface at the left edge of the interpretation space and the second will be fixed at the right edge. All other vertices will be placed at regular intervals between the first and second with fixed x-coordinates. The parameter for the inversion will be the z-coordinates of these N - 2 vertices.

Implementation

The only component that must be implemented is the data-misfit function. We'll use the `Misfit` class as a base and implement only the required problem-specific methods. In this case, we must implement only two methods:

- `_get_predicted`, which returns the predicted data given a parameter vector.
- `_get_jacobian`, which returns the Jacobian (sensitivity) matrix given a parameter vector.

To compute the predicted data, we must convert the parameter vector (z-coordinates of the vertices) into a `fatiando.mesher.Polygon` object and pass it to the function `fatiando.gravmag.talwani.gz`.

The Jacobian matrix can also be computed using `fatiando.gravmag.talwani.gz` by a finite-difference approximation. For each unknown z-coordinate z_i with value z_i^k in the current iteration, we compute the difference between gravitational effect of a polygon with $z_i = z_i^k + 0.5\Delta z$ and $z_i = z_i^k - 0.5\Delta z$, then divide the difference by Δz. An efficient way of computing the difference is to compute the effect of the polygon with vertices (x_{i-1}, z_{i-1}), $(x_i, z_i - 0.5\Delta z)$, (x_{i+1}, z_{i+1}), and $(x_i, z_i + 0.5\Delta z)$.

We start by importing the required libraries and modules of Fatiando a Terra.

In [1]:
```
%matplotlib inline
```

In [2]:
```
import numpy as np
from fatiando import utils
from fatiando.vis import mpl
from fatiando.mesher import Polygon
from fatiando.gravmag import talwani
from fatiando.inversion.base import Misfit
from fatiando.inversion.regularization import (
    Damping, Smoothness1D, TotalVariation1D, LCurve)
```

Now we can create the `PolyBasin` class that will perform our inversion. This class will implement the `_get_predicted` and `_get_jacobian` methods and also the `p2verts` method to convert a parameter vector into a `Polygon` object. You'll notice that this class also implements the `fit` method. This is done so that we can convert the estimated parameter vector into a `Polygon` object and store it in the `estimate_` attribute. The actual computations are performed by `Misfit.fit`.

In [3]:
```
class PolyBasin(Misfit):
    def __init__(self, x, z, gz, npoints, density):
        super(PolyBasin, self).__init__(
            data=gz,
            positional=dict(x=x, z=z),
            model=dict(density=density),
            nparams=npoints, islinear=False)
        # Top left and right points of the polygon
        self._left, self._right = [[x.min(), 0]], [[x.max(), 0]]
        # x coordinates of the other polygon points
        self._xs = np.linspace(x.min(), x.max(), npoints + 2)[::-1][1:-1]

    def _get_jacobian(self, p):
        verts = self.p2verts(p)
        x, z = self.positional['x'], self.positional['z']
        xs = self._xs
        dz = 20.
        jac = np.zeros((self.ndata, self.nparams))
        for i in xrange(self.nparams):
            difference = Polygon(
                [verts[i + 2],
                 [xs[i], p[i] - 0.5*dz],
                 verts[i],
                 [xs[i], p[i] + 0.5*dz]], self.model)
            jac[:,i] = np.nan_to_num(talwani.gz(x, z, [difference]))/dz
```

```
        return jac

    def _get_predicted(self, p):
        x, z = self.positional['x'], self.positional['z']
        poly = Polygon(self.p2verts(p), self.model)
        return talwani.gz(x, z, [poly])

    def p2verts(self, p):
        return self._right + np.transpose([self._xs, p]).tolist() + self._

    def fit(self):
        super(PolyBasin, self).fit()
        self._estimate = Polygon(self.p2verts(self.p_), self.model)
        return self
```

Now that we have our `PolyBasin` class, we can use all of the regularizing functions implemented in `fatiando.inversion.regularization` to run out inversion.

Testing on synthetic data

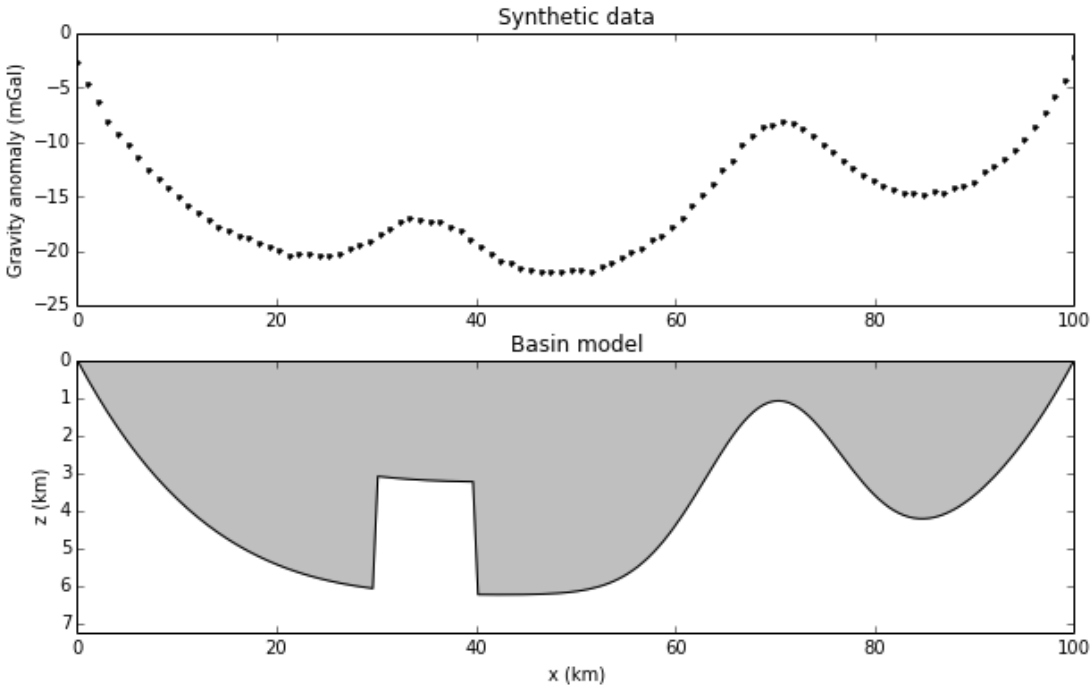
We will generate some synthetic data from a model sedimentary basin. The data will be contaminated with pseudo-random Gaussian noise with zero mean and 0.1 mGal standard deviation.

In [4]:
```
# The x-coordinates of the polygon vertices
xs = np.linspace(0, 100000, 200)[::-1]
# The z-coordinates of the polygon vertices
depths = (-10**-15*(xs - 50000)**4 + 6000
          - 5000*np.exp(-(xs - 70000)**2/(10000**2)))
depths -= depths.min()
# Create a discontinuity
depths[(xs > 30000) & (xs < 40000)] -= 3000
density = -100
model = Polygon(np.transpose([xs, depths]), {'density':density})

x = np.linspace(0, 100000, 100)
z = -100*np.ones_like(x)
gz = utils.contaminate(talwani.gz(x, z, [model]), 0.1, seed=20)

mpl.figure(figsize=(10, 6))
mpl.subplot(2, 1, 2)
mpl.title('Basin model')
mpl.polygon(model, fill='gray', alpha=0.5)
mpl.ylim(depths.max() + 1000, depths.min())
mpl.xlim(x.min(), x.max())
mpl.m2km()
mpl.xlabel('x (km)')
mpl.ylabel('z (km)')
mpl.subplot(2, 1, 1)
mpl.title('Synthetic data')
mpl.plot(x*0.001, gz, '.k')
mpl.ylabel('Gravity anomaly (mGal)')
```

Out [4]: `<matplotlib.text.Text at 0x3f0a7d0>`



For convenience, I'll create a function that plots the model, synthetic data, an estimated basin relief, and the corresponding predicted data.

In [5]:
```
def plot(solver, title):
    mpl.figure(figsize=(10, 6))
    mpl.subplot(2, 1, 1)
    mpl.title(title)
    mpl.plot(x*0.001, gz, '.k', label='Observed')
    mpl.plot(x*0.001, solver.predicted(), '-r', label='Predicted')
    mpl.legend(loc='upper center')
    mpl.ylabel('Gravity anomaly (mGal)')
    mpl.subplot(2, 1, 2)
    mpl.polygon(model, fill='gray', alpha=0.5)
    mpl.polygon(solver.estimate_, '.-b', linewidth=2)
    mpl.ylim(depths.max() + 1000, depths.min())
    mpl.xlim(x.min(), x.max())
    mpl.m2km()
    mpl.xlabel('x (km)')
    mpl.ylabel('z (km)')
```

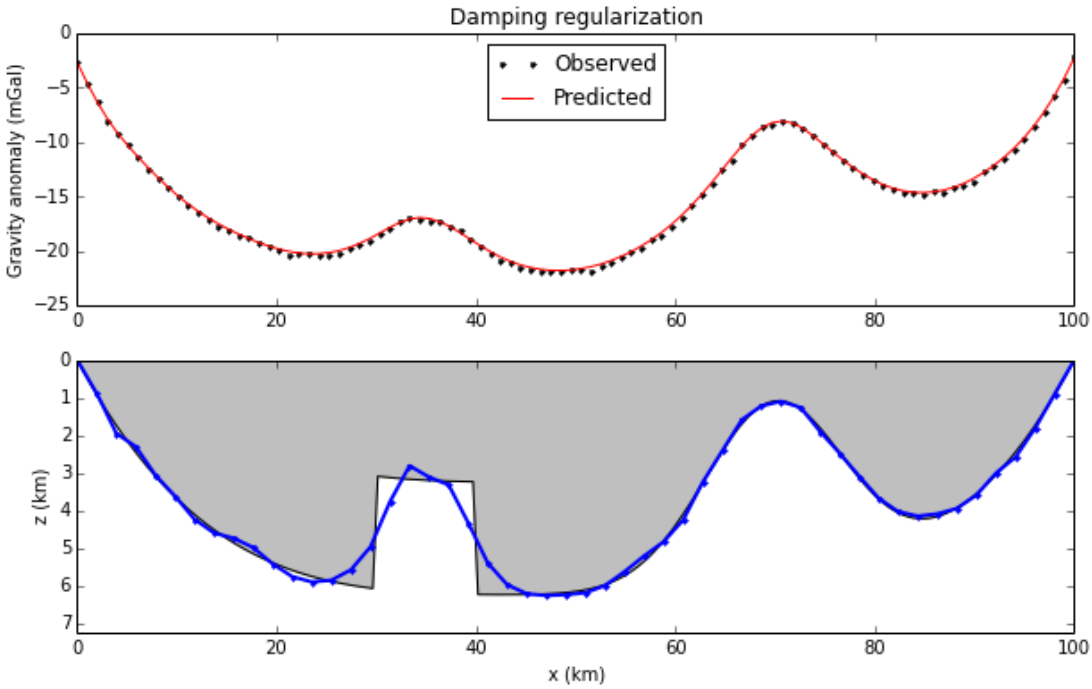
Now we can create an instance of our `PolyBasin` class. Since this is a non-linear problem, we must also specify an initial estimate. We'll use a constant depth of 5 km for this.

In [6]:
```
misfit = PolyBasin(x, z, gz, 50, density)
initial = 5000*np.ones(misfit.nparams)
```

We can create an inversion solver that uses damping (Tikhonov order zero) regularizing by simply adding a `Damping` instance multiplied by the regularization parameter (in this case, 10^{-9}). We must call `config` to specify the optimization method and initial estimate. Calling `fit` runs the inversion using the specified setup.

In [7]:
```
damped = misfit + 10**-9*Damping(misfit.nparams)
# We'll use the Levemberg-Marquardt algorithm for this
damped.config('levmarq', initial=initial).fit()
```

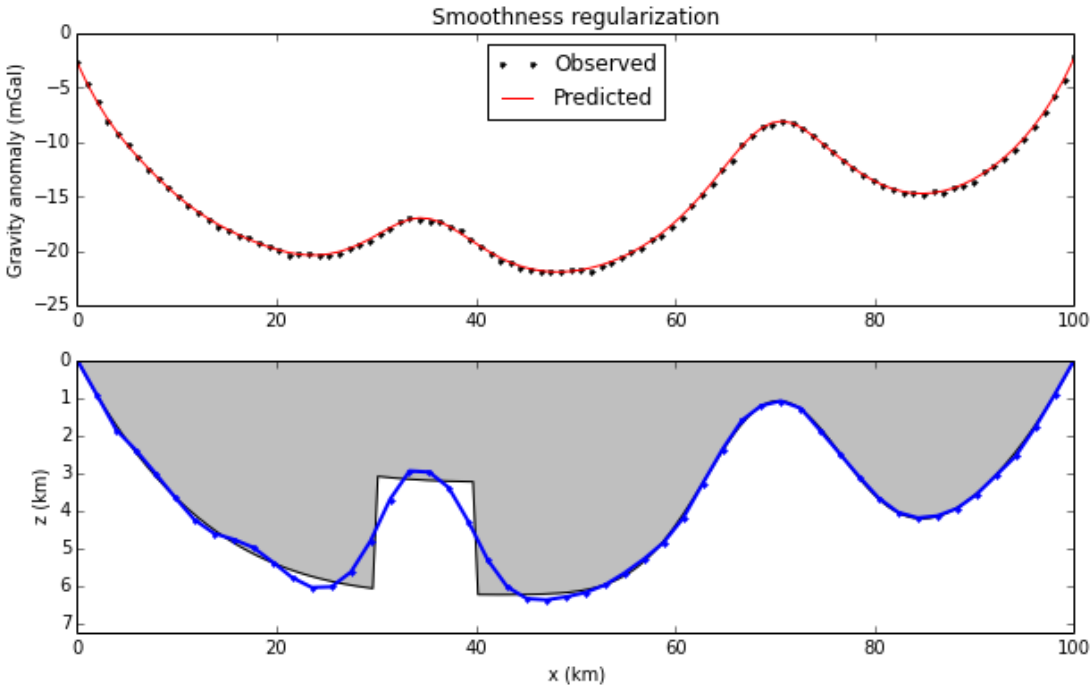
```
plot(damped, 'Damping regularization')
```



To use a different type of regularization, simply exchange `Damping` for the appropriate class. The code bellow uses smoothness (Tikhonov order one) regularization.

In [8]:
```
smooth = misfit + 10**-9*Smoothness1D(misfit.nparams)
smooth.config('levmarq', initial=initial).fit()

plot(smooth, 'Smoothness regularization')
```

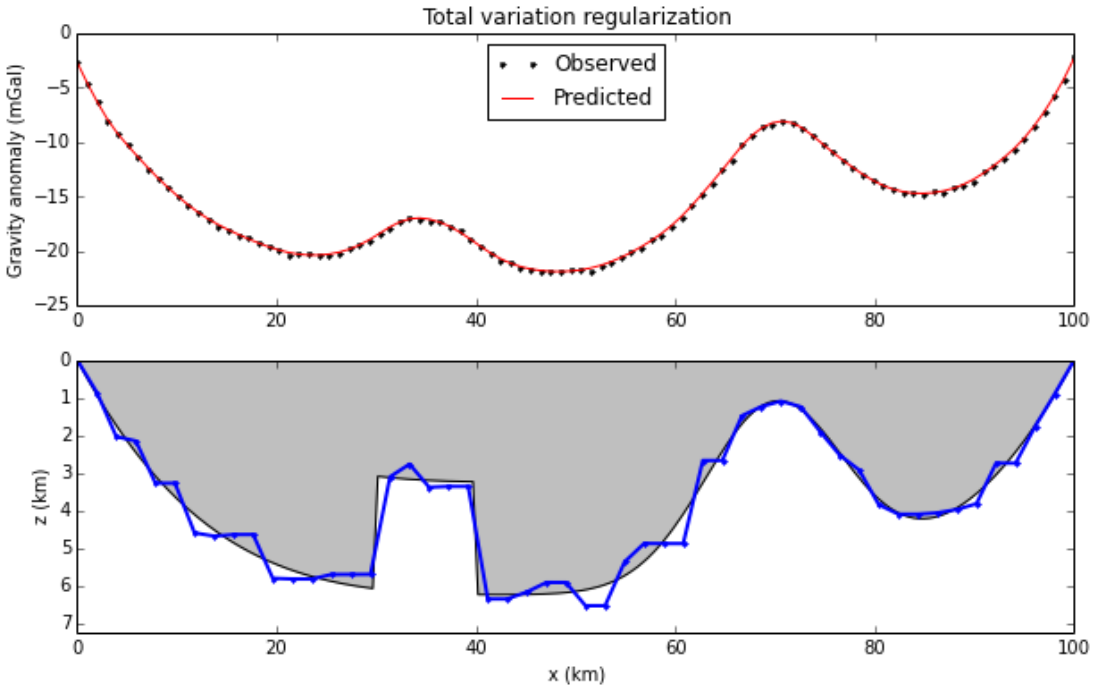


The same instance of `PolyBasin` can be used with any regularization class that is compatible.

Notice how imposing smoothness on the solution results in a smooth basin, even in places where the relief is not smooth. We can use total variation regularization to impose sharp variations in the relief. However, this will also create sharp variations where none exist.

In [9]:
```
sharp = misfit + 10**-7*TotalVariation1D(10**-3, misfit.nparams)
sharp.config('levmarq', initial=initial).fit()

plot(sharp, 'Total variation regularization')
```

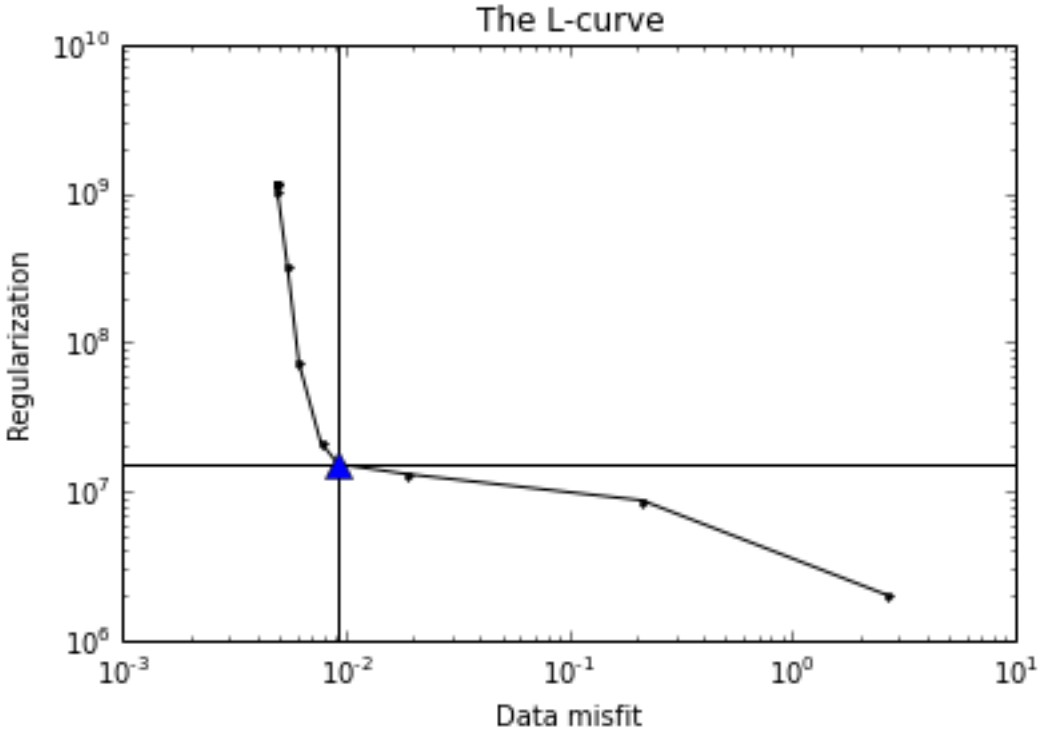


In the above examples, the regularization parameter μ was set using trial-and-error. An automatic way to estimate the optimum regularization parameter is using an L-curve analysis. This is implemented in class `LCurve` of Fatiando a Terra. It requires a data-misfit instance, a regularizing function instance, and a list of regularization parameters to consider. When the `fit` method of `LCurve` is called, it will search the provided list for the value of μ that falls in the "elbow" of the data-misfit x regularizing function plot (marked by the blue triangle in the figure bellow).

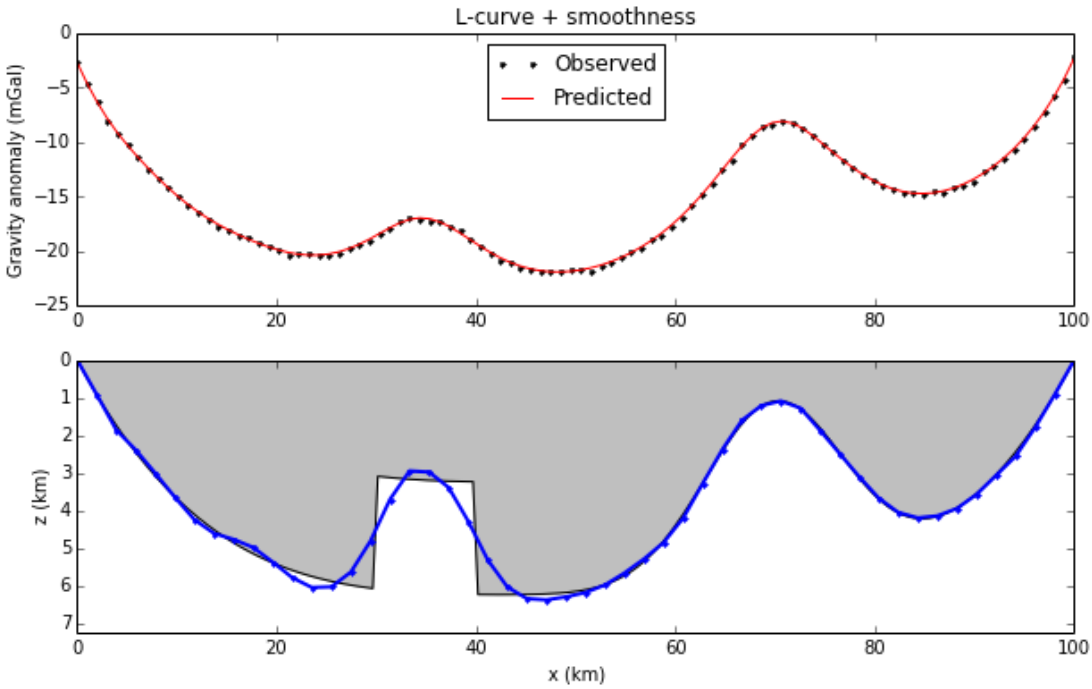
In [10]:
```
lcurve = LCurve(misfit, Smoothness1D(misfit.nparams),
                [10**i for i in range(-20, -5, 1)])
# LCurve behaves exactly like all the other solvers
lcurve.config('levmarq', initial=initial).fit()

lcurve.plot_lcurve()
mpl.title('The L-curve')
```

Out [10]: <matplotlib.text.Text at 0x46b0990>



In [11]:
```
plot(lcurve, 'L-curve + smoothness')
```



`LCurve` stores the estimated regularization parameter in the `regul_param_` attribute.

```
In [12]: print lcurve.regul_param_
```

```
1e-09
```

4 Conclusions and future work

In the period of March 2013 until March 2014, I have focused on creating the computational infrastructure needed to implemented a range a gravity inversion methods in spherical coordinates using tesseroids. The forward modeling code has been written and tested using two different implementations. We obtained novel results regarding the accuracy of the modeling algorithm and are working of producing an article describing these finds.

A framework for solving inverse problems has been added to Fatiando a Terra to facilitate the development of new inversion methods. This framework automates common tasks and provides a range of components that can be readily used in new inverse problems. Some improvements still need to made, especially:

- Optimize the computation time and memory usage to deal with large-scale problems.
- Implement some form of uncertainty estimation.

5 References

Asgharzadeh, M. F., von Frese, R. R. B., Kim, H. R., Leftwich, T. E., and Kim, J. W. (2007). Spherical prism gravity effects by Gauss-Legendre quadrature integration. Geophysical Journal International, 169(1), 1-11.

Aster, R. C., Borchers, B., and Thurber, C. H. (2013). Parameter estimation and inverse problems. Academic Press.

Barbosa, V. C. F., Silva, J. B. C., and Medeiros, W. E. (1999). Gravity inversion of a discontinuous relief stabilized by weighted smoothness constraints on depth. GEOPHYSICS, 64(5), 1429-1437.

Chaves, C. A. M., and Ussami, N. (2013). Modeling 3-D density distribution in the mantle from inversion of geoid anomalies: Application to the Yellowstone Province. Journal of Geophysical Research: Solid Earth, 118(12), 6328-6351.

Cox, L. H., Wilson, G. A., and Zhdanov, M. S. (2010). 3D inversion of airborne electromagnetic data using a moving footprint. Exploration Geophysics, 41(4), 250.

Davis, K., and Li, Y. (2013). Efficient 3D inversion of magnetic data via octree-mesh discretization, space-filling curves, and wavelets. GEOPHYSICS, 78(5), J61-J73.

Farquharson, C. G., and Oldenburg, D. W. (1998). Non-linear inversion using general measures of data misfit and model structure. Geophysical Journal International, 134(1), 213-227.

Fomel, S. (2007). Shaping regularization in geophysical-estimation problems. Geophysics, 72(2), R29-R36.

Grombein, T., Seitz, K., and Heck, B. (2013). Optimized formulas for the gravitational field of a tesseroid. Journal of Geodesy, 87(7), 645-660.

Heck, B., and Seitz, K. (2007). A comparison of the tesseroid, prism and point-mass approaches for mass reductions in gravity field modelling. Journal of Geodesy, 81(2), 121-136.

Li, Y., and Oldenburg, D. W. (1998). 3-D inversion of gravity data. GEOPHYSICS, 63(1), 109-119.

Li, Y., and Oldenburg, D. W. (2010). Rapid construction of equivalent sources using wavelets. Geophysics, 75(3), L51-L59.

Li, Z., Hao, T., Xu, Y., and Xu, Y. (2011). An efficient and adaptive approach for modeling gravity effects in spherical coordinates. Journal of Applied Geophysics, 73(3), 221-231.

Martins, C. M., Lima, W. A., Barbosa, V. C., and Silva, J. B. (2011). Total variation regularization for depth-to-basement estimate: Part 1 - Mathematical details and applications. Geophysics, 76(1), I1-I12.

Panet, I., Pajot-Métivier, G., Greff-Lefftz, M., Métivier, L., Diament, M., and Mandea, M. (2014). Mapping the mass distribution of Earth/'s mantle using satellite-derived gravity gradients. Nature Geoscience, 7(2), 131-135.

Portniaguine, O., and Zhdanov, M. S. (1999). Focusing geophysical inversion images. Geophysics, 64(3), 874-887.

Scales, J. A., Smith, M. L., and Treitel, S. (2001). Introductory Geophysical Inverse Theory. Samizdat Press.

Silva Dias, F. J., Barbosa, V. C., and Silva, J. B. (2009). 3D gravity inversion through an adaptive-learning procedure. GEOPHYSICS, 74(3), I9-I21.

Talwani, M., Worzel, J. L., and Landisman, M. (1959). Rapid gravity computations for two-dimensional bodies with application to the Mendocino submarine fracture zone. Journal of Geophysical Research, 64(1), 49-59.

Uieda, L., and Barbosa, V. C. F. (2012). Robust 3D gravity gradient inversion by planting anomalous densities. Geophysics, 77(4), G55-G66.

Wieczorek, M. A., and Phillips, R. J. (1998). Potential anomalies on a sphere: Applications to the thickness of the lunar crust. Journal of Geophysical Research: Planets (1991-2012), 103(E1), 1715-1724.

Wild-Pfeiffer, F. (2008). A comparison of different mass elements for use in gravity gradiometry. Journal of Geodesy, 82(10), 637-653.