synthetic-simple

March 23, 2016

1 Test on simple synthetic data

This notebook generates some synthetic data based on a simple model and tests the inversion method on it. The test will be used to show that the method works and is efficient. This test uses cross-validation to determine the optimal regularization parameter. **We will assume that the reference level and density contrast are known**. The data, model, and results generated here are saved to files in the respective folders for later use.

1.1 Package imports

```
In [1]: # Insert the plots into the notebook
        %matplotlib inline
```

Load the standard scientific Python stack to numerical analysis and plotting.

```
In [2]: from __future__ import division, print_function
        import pstats
        import datetime
        import zipfile
        import multiprocessing
        import cPickle as pickle
        import numpy as np
        import scipy as sp
        import matplotlib.pyplot as plt
        from mpl_toolkits.basemap import Basemap
        import seaborn  # Makes the default style of the plots nicer
```

The computations generate a lot of run-time warnings. They aren't anything to be concerned about so disable them to avoid clutter.

```
In [3]: import warnings
        warnings.simplefilter('ignore')
```

Load the required modules from Fatiando a Terra.

```
In [4]: from fatiando.inversion import Smoothness2D
        from fatiando.vis import mpl
        from fatiando.gravmag import tesseroid
        from fatiando import utils, gridder
        import fatiando

In [5]: print("Version of Fatiando a Terra used: {}".format(fatiando.__version__))

Version of Fatiando a Terra used: 237ba1dd35d47ea0a5e286781faddfe60ab4d12d
```

Load our custom classes and functions.

```
In [6]: from mohoinv import TesseroidRelief, MohoGravityInvSpherical, make_mesh
        from mohoinv import split_data, score_test_set, score_all, fit_all
```

Get the number of cores in this computer to run the some things in parallel.

```
In [7]: ncpu = multiprocessing.cpu_count()
        print("Number of cores: {}".format(ncpu))
```

```
Number of cores: 8
```

1.2 Create a model and generate synthetic data

Below we define a function that generates our tesseroid model of the Moho relief. The model assumes a homogeneous density contrast along the entire Moho. The relief of the Moho simulates the transition between continental and oceanic crust.

```
In [8]: def generate_model():
            # Make a regular grid inside an area.
            # Grid points will be the center of the top of each tesseroid in the model
            shape = (40, 50)
            area = (10, 70, -50, 50)
            lat, lon = gridder.regular(area, shape)
            # Define our model Moho relief
            # The general flow from continent to ocean
            # is generated by the erf function
            relief = (
                -30e3 + 15e3*sp.special.erf((lon - 10)/20)
                - 10e3*utils.gaussian2d(lat, lon, 10, 15, x0=25, y0=-30)
                + 10e3*utils.gaussian2d(lat, lon, 15, 20, x0=53, y0=-25)
                - 10e3*utils.gaussian2d(lat, lon, 3, 3, x0=50, y0=30)
                - 10e3*utils.gaussian2d(lat, lon, 10, 10, x0=30, y0=25)**2
                + 5e3*utils.gaussian2d(lat, lon, 40, 3, x0=40, y0=40, angle=15))
            density_contrast = 400
            density = density_contrast*np.ones_like(relief)
            # Define the reference level (height in meters).
            # This is the Moho depth of the Normal Earth
            reference = -30e3
            # The density contrast is negative if the relief is below the reference
            density[relief < reference] *= -1
            # Make a TesseroidRelief using the make_mesh utility function and
            # assign the density contrast values to it.
            model = make_mesh(area, shape, relief, reference)
            model.addprop('density', density)
            # Print some information
            print(u"Density contrast: {} kg/m$^3$".format(density_contrast))
            print(u"Reference level: {} m".format(reference))
            print(u"Number of tesseroids: {} x {} = {}".format(shape[0], shape[1],
                                                               model.size))
            return model
```

```
In [9]: model = generate_model()
```

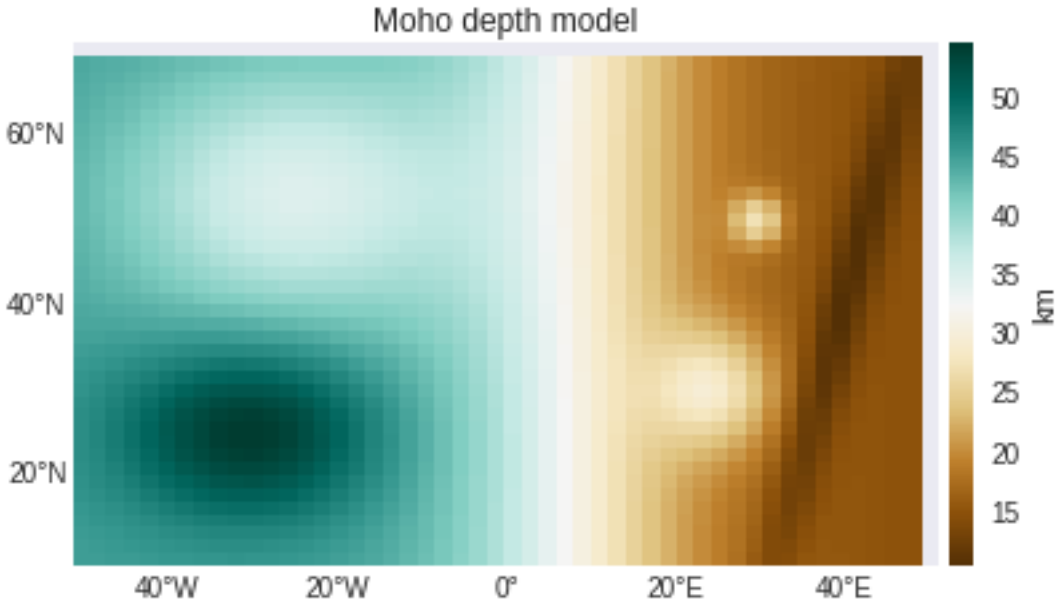
```
Density contrast: 400 kg/m$^3$
Reference level: -30000.0 m
Number of tesseroids: 40 x 50 = 2000
```

Lets make a plot of our Moho model.

```
In [10]: bm = Basemap(projection='cyl',
                      llcrnrlon=model.area[2], urcrnrlon=model.area[3],
                      llcrnrlat=model.area[0], urcrnrlat=model.area[1],
                      resolution='c')
```

```
In [11]: x, y = bm(model.lon.reshape(model.shape), model.lat.reshape(model.shape))

         plt.figure(figsize=(7, 3.5))
         bm.pcolormesh(x, y, -0.001*model.relief.reshape(model.shape), cmap='BrBG')
         plt.colorbar(pad=0.01).set_label('km')
         bm.drawmeridians(np.arange(-40, 45, 20), labels=[0, 0, 0, 1], linewidth=0)
         bm.drawparallels(np.arange(20, 65, 20), labels=[1, 0, 0, 0], linewidth=0)
         plt.title("Moho depth model")
```

```
Out[11]: <matplotlib.text.Text at 0x7f49c39db490>
```



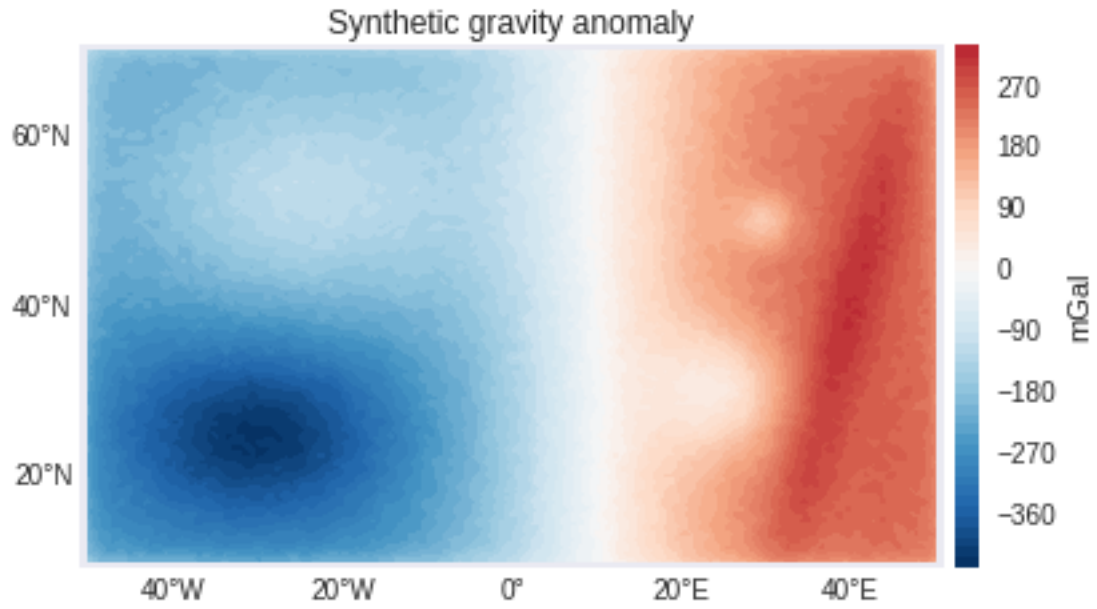
Generate the computation grid for our synthetic dataset. The grid will have half the spacing of the model. This way, we'll have more points than we'll need to run the inversion. The extra points will be separated into a test dataset for cross-validation (see section Cross-validation below).

```
In [12]: # clon and clat are the coordinates of the center of each model cell
         area = [model.clat.min(), model.clat.max(), model.clon.min(), model.clon.max()]
         # Increase the shape to have half the grid spacing
         full_shape = [s*2 - 1 for s in model.shape]
         grid_height = 50e3
         full_lat, full_lon, full_height = gridder.regular(area, full_shape, z=grid_height)
         print('Number of grid points: {} x {} = {}'.format(full_shape[0], full_shape[1],
                                                            full_shape[0]*full_shape[1]))
         print('Grid height: {} m'.format(grid_height))
```

```
Number of grid points: 79 x 99 = 7821
Grid height: 50000.0 m
```

Forward model the synthetic data on a regular grid at a constant height and contaminate it with pseudo-random Gaussian noise.

```
In [13]: %%time
         full_data_noisefree = tesseroid.gz(full_lon, full_lat, full_height, model, njobs=ncpu)

CPU times: user 16 ms, sys: 44 ms, total: 60 ms
Wall time: 6.53 s

In [14]: full_data = utils.contaminate(full_data_noisefree, 5, seed=0)

In [15]: x, y = bm(full_lon, full_lat)

         plt.figure(figsize=(7, 3.5))
         ranges = np.abs([full_data.max(), full_data.min()]).max()
         bm.contourf(x, y, full_data, 50, cmap='RdBu_r', tri=True, vmin=-ranges, vmax=ranges)
         plt.colorbar(pad=0.01).set_label('mGal')
         bm.drawmeridians(np.arange(-40, 45, 20), labels=[0, 0, 0, 1], linewidth=0)
         bm.drawparallels(np.arange(20, 65, 20), labels=[1, 0, 0, 0], linewidth=0)
         plt.title('Synthetic gravity anomaly')

Out[15]: <matplotlib.text.Text at 0x7f49e7ff0490>
```



1.3 Save the model and synthetic data

We'll save the data to a text file ../data/synthetic-data-simple.txt for later use and plotting.

```
In [16]: now = datetime.datetime.utcnow().strftime('%d %B %Y %H:%M:%S UTC')
         header = """# Generated by sinthetic-simple.ipynb on {date}
         # shape (nlat, nlon):
         # {nlat} {nlon}
         # lat lon height gravity_anomaly_noisefree gravity_anomaly_noisy
```

```
        """.format(date=now, nlat=full_shape[0], nlon=full_shape[1])
        with open('../data/synthetic-data-simple.txt', 'w') as f:
            f.write(header)
            np.savetxt(f, np.c_[full_lat, full_lon, full_height, full_data_noisefree, full_data],
                       fmt='%.5f')
```

The model we'll save to a Python pickle file. The `pickle` module allows us to serialize an object and load it back later. We'll use it to serialize the `model` object and save the file to the `model` folder.

```
In [17]: now = datetime.datetime.utcnow().strftime('%d %B %Y %H:%M:%S UTC')
         model.metadata = "Generated by sinthetic-simple.ipynb on {date}".format(date=now)
         with open('../model/synthetic-simple.pickle', 'w') as f:
             pickle.dump(model, f)
```

1.4 Cross-validation

First, we must separate the dataset into two parts: one for the inversion, another for cross-validation. The inversion dataset will have double the grid spacing and (for this test) must fall on top of each grid cell. The remaining data will be used for cross-validation.

```
In [18]: inversion_set, test_set, shape = split_data([full_lat, full_lon, full_height, full_data],
                                                    full_shape, every_other=2)

         print("Number of inversion grid points: {} x {} = {}".format(shape[0], shape[1],
                                                                      shape[0]*shape[1]))
         print("Number of test set points: {}".format(test_set[0].size))
```

```
Number of inversion grid points: 40 x 50 = 2000
Number of test set points: 5821
```

Test if the inversion set falls on top of the model cells.

```
In [19]: lat, lon, height, data = inversion_set
         assert np.allclose(model.clon.ravel(), lon, rtol=1e-10, atol=0)
         assert np.allclose(model.clat.ravel(), lat, rtol=1e-10, atol=0)
```

No errors means that both checks (`assert`) passed.
The score (MSE) should be zero (perfect fit) if we pass in the error-free model data.

```
In [20]: score_test_set(model, full_lat, full_lon, full_height, full_data_noisefree, njobs=ncpu)
```

```
Out[20]: 0.0
```

1.5 Inversion setup

We need to make a mesh for the inversion. We'll use a mesh that is equal to the original model. This will make it easier to compare the inversion results with the true model.

```
In [21]: mesh = model.copy(deep=True)
```

We also need to define an initial estimate because this is a non-linear inversion. Next, we create the data-misfit and regularization objects that we'll use in the inversions below.

```
In [22]: misfit = MohoGravityInvSpherical(lat, lon, height, data, mesh)
         regul = Smoothness2D(mesh.shape)
         initial = np.ones(mesh.size)*(-60e3)
```

1.6 Define plotting functions

These functions plot the results of the inversion for us. We'll define them all here and use them below for each inversion that we run.

In [23]:
```
def plot_fit(result, bm):
    solver = result['solution']
    predicted = solver[0].predicted()

    ranges = np.abs([data.max(), data.min()]).max()

    plt.figure(figsize=(7, 3.5))
    plt.title('Observed and predicted data ({})'.format(solver.stats_['method']))
    levels = mpl.contourf(lon, lat, data, shape, 25, cmap='RdBu_r', basemap=bm,
                          vmin=-ranges, vmax=ranges)
    plt.colorbar(pad=0.01).set_label('mGal')
    mpl.contour(lon, lat, predicted, shape, levels, basemap=bm, color='#333333')
    bm.drawmeridians(np.arange(-40, 45, 20), labels=[0, 0, 0, 1], linewidth=0)
    bm.drawparallels(np.arange(20, 65, 20), labels=[1, 0, 0, 0], linewidth=0)

def plot_residuals(result):
    solver = result['solution']
    residuals = solver[0].residuals()

    plt.figure(figsize=(5, 4))
    plt.text(0.58, 0.8,
             "mean = {:.2f}\n    std = {:.2f}".format(residuals.mean(), residuals.std()),
             transform=plt.gca().transAxes)
    plt.hist(residuals, bins=15, normed=True, histtype='stepfilled')
    plt.xlabel('Residuals (mGal)')
    plt.ylabel('Normalized frequency')
    plt.tight_layout(pad=0)

def plot_estimate(result, bm):
    solver = result['solution']
    moho = solver.estimate_
    x, y = bm(moho.lons, moho.lats)

    plt.figure(figsize=(7, 3.5))
    plt.title("Estimated Moho depth ({})".format(solver.stats_['method']))
    bm.pcolormesh(x, y, -0.001*moho.relief.reshape(moho.shape), cmap='BrBG')
    plt.colorbar(pad=0.01).set_label('km')
    bm.drawmeridians(np.arange(-40, 45, 20), labels=[0, 0, 0, 1], linewidth=0)
    bm.drawparallels(np.arange(20, 65, 20), labels=[1, 0, 0, 0], linewidth=0)

def plot_diff(result, model, bm):
    solver = result['solution']
    moho = solver.estimate_
    x, y = bm(moho.lons, moho.lats)

    diff = -0.001*(model.relief - moho.relief).reshape(moho.shape)
    ranges = np.abs([diff.max(), diff.min()]).max()

    plt.figure(figsize=(7, 3.5))
    plt.title('Difference between true and estimated ({})'.format(solver.stats_['method']))
```

```
        bm.pcolormesh(x, y, diff, cmap='RdYlBu_r', vmin=-ranges, vmax=ranges)
        plt.colorbar(pad=0.01).set_label('km')
        bm.drawmeridians(np.arange(-40, 45, 20), labels=[0, 0, 0, 1], linewidth=0)
        bm.drawparallels(np.arange(20, 65, 20), labels=[1, 0, 0, 0], linewidth=0)

    def plot_cv(result, log=True):
        regul_params = result['regul_params']
        scores = result['scores']
        best = result['best_index']
        solver = result['solution']
        plt.figure(figsize=(5, 3.5))
        plt.title('Cross-validation for {}'.format(solver.stats_['method']))
        plt.plot(regul_params, scores, marker='o')
        plt.plot(regul_params[best], scores[best], 's', markersize=10,
                 color=seaborn.color_palette()[2], label='Minimum')
        plt.legend(loc='upper left')
        plt.xscale('log')
        if log:
            plt.yscale('log')
        plt.xlabel('Regularization parameter')
        plt.ylabel(u'Mean Square Error')
        plt.tight_layout()

    def plot_convergence(result):
        solver = result['solution']
        plt.figure(figsize=(5, 3.5))
        plt.title('Convergence of {}'.format(solver.stats_['method']))
        plt.plot(range(solver.stats_['iterations'] + 1), solver.stats_['objective'])
        plt.xlabel('Iteration')
        plt.yscale('log')
        plt.ylabel('Goal function')
        plt.tight_layout()
```

1.7 Run the inversion and cross-validation

We'll keep the results in a Python dictionary (`dict`) along with all configuration and other metadata. We can then save this dict to a Pickle file and have inversion information saved with the results.

```
In [24]: results = dict()  # For now, the results dict is empty
```

We'll make a Python dictionary (`dict`) to store the solver configuration for this method. `tol` is the tolerance level that controls the stopping criterion. `maxit` is the maximum allowed number of iterations.

```
In [25]: results['config'] = dict(method='newton', initial=initial, tol=0.1, maxit=15)
```

Next, we define a list of the possible regularization parameters that will be considered during cross-validation. `logspace` generates a list of values evenly spaced in a logarithmic scale.

```
In [26]: results['regul_params'] = np.logspace(-6, -1, 16)
         results['regul_params']

Out[26]: array([  1.00000000e-06,   2.15443469e-06,   4.64158883e-06,
                  1.00000000e-05,   2.15443469e-05,   4.64158883e-05,
                  1.00000000e-04,   2.15443469e-04,   4.64158883e-04,
                  1.00000000e-03,   2.15443469e-03,   4.64158883e-03,
                  1.00000000e-02,   2.15443469e-02,   4.64158883e-02,
                  1.00000000e-01])
```

Run the inversion for each value in `regul_params` (in parallel using all available cores). This takes some time to run.

```
In [27]: solvers = [(misfit + mu*regul).config(**results['config'])
                    for mu in results['regul_params']]

         %time solutions = fit_all(solvers, njobs=ncpu)
```

```
CPU times: user 356 ms, sys: 100 ms, total: 456 ms
Wall time: 5min 3s
```

Store only the estimated Moho models. We can compute the predicted data from them later if we want.

```
In [28]: results['solutions'] = [s.estimate_ for s in solutions]
```

Score the results against the test dataset.

```
In [29]: %%time
         results['scores'] = score_all(results['solutions'], test_set,
                                       points=False, njobs=ncpu)
```

```
CPU times: user 84 ms, sys: 72 ms, total: 156 ms
Wall time: 1min 9s
```

The best solution is the one with the smallest cross-validation score.

```
In [30]: best = np.argmin(results['scores'])
         results['best_index'] = best
         results['solution'] = solutions[best]
```

1.7.1 Save the results to a pickle file

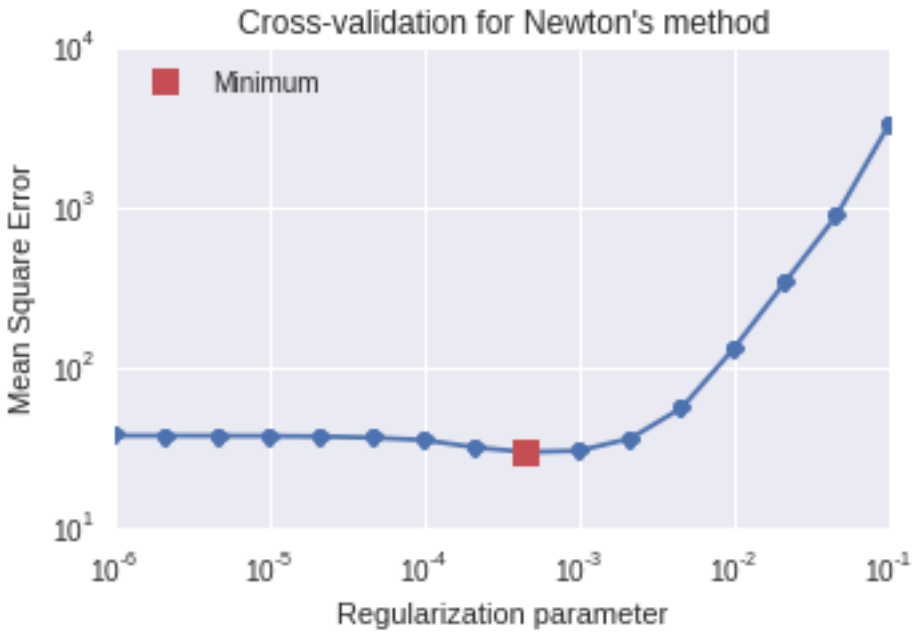
Also save some metadata about it. Since the resulting pickle file will be large, we'll store it in a `zip` archive.

```
In [31]: def pickle_results(results, fname):
             # Dump the results dict to a pickle file
             pickle_file = '{}.pickle'.format(fname)
             now = datetime.datetime.utcnow().strftime('%d %B %Y %H:%M:%S UTC')
             results['metadata'] = "Generated by sinthetic-simple.ipynb on {date}".format(date=now)
             with open('results/{}'.format(pickle_file), 'w') as f:
                 pickle.dump(results, f)
             # Zip the pickle file
             zipargs = dict(mode='w', compression=zipfile.ZIP_DEFLATED)
             with zipfile.ZipFile('results/{}.zip'.format(fname), **zipargs) as f:
                 f.write('results/{}'.format(pickle_file), arcname=pickle_file)

In [32]: pickle_results(results, 'synthetic-simple')
```

1.7.2 Plot the results

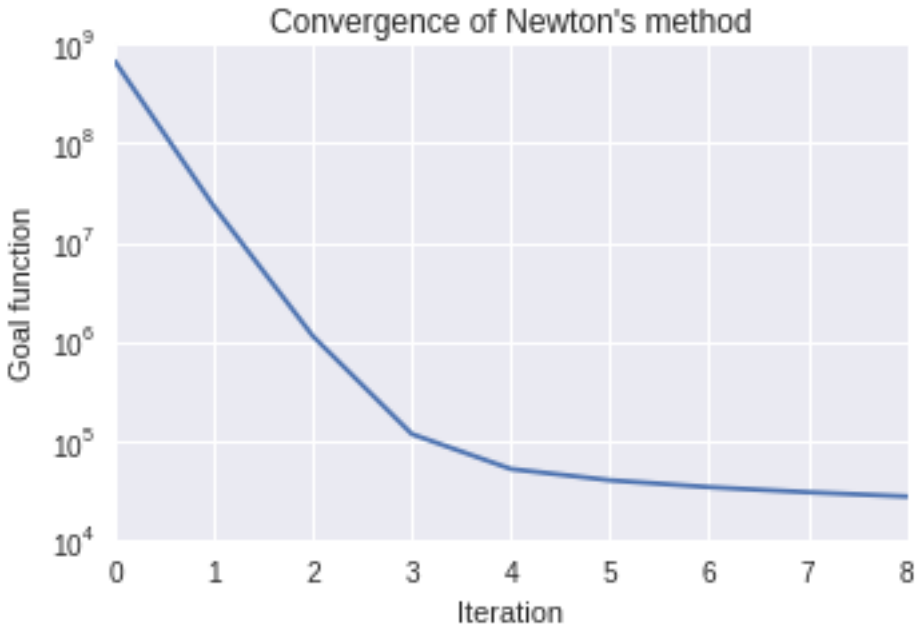
Plot the cross-validation data. The graph will show the MSE per regularization parameter. The chosen solution is the one that minimizes the MSE.

```
In [56]: plot_cv(results)
```



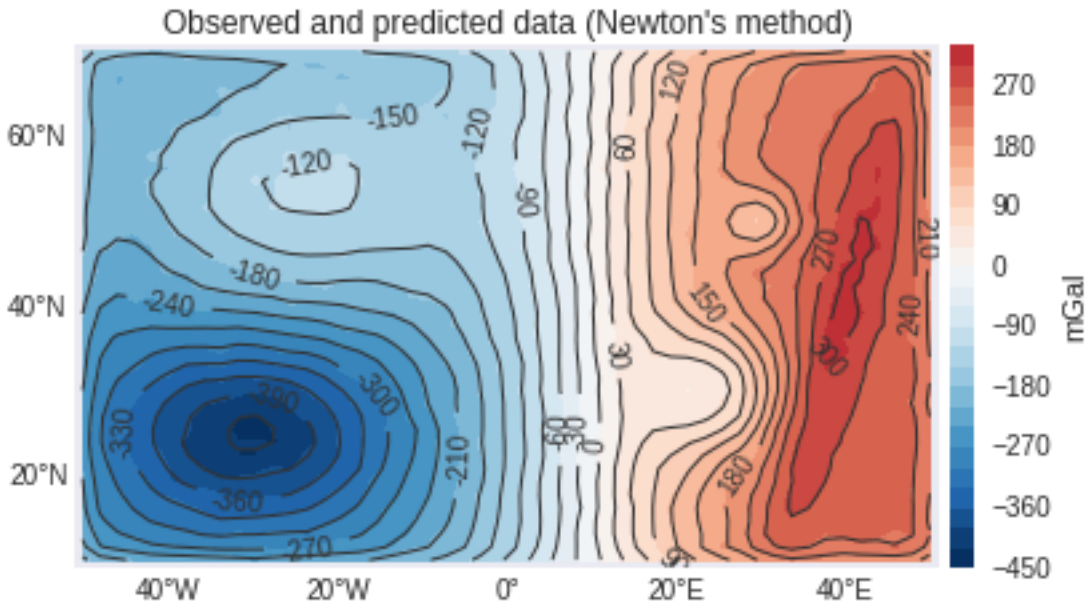
Plot the convergence of the best solution (goal function value per iteration) to see if the solution converges. Note that the y-axis is in a logarithmic scale.

```
In [34]: plot_convergence(results)
```

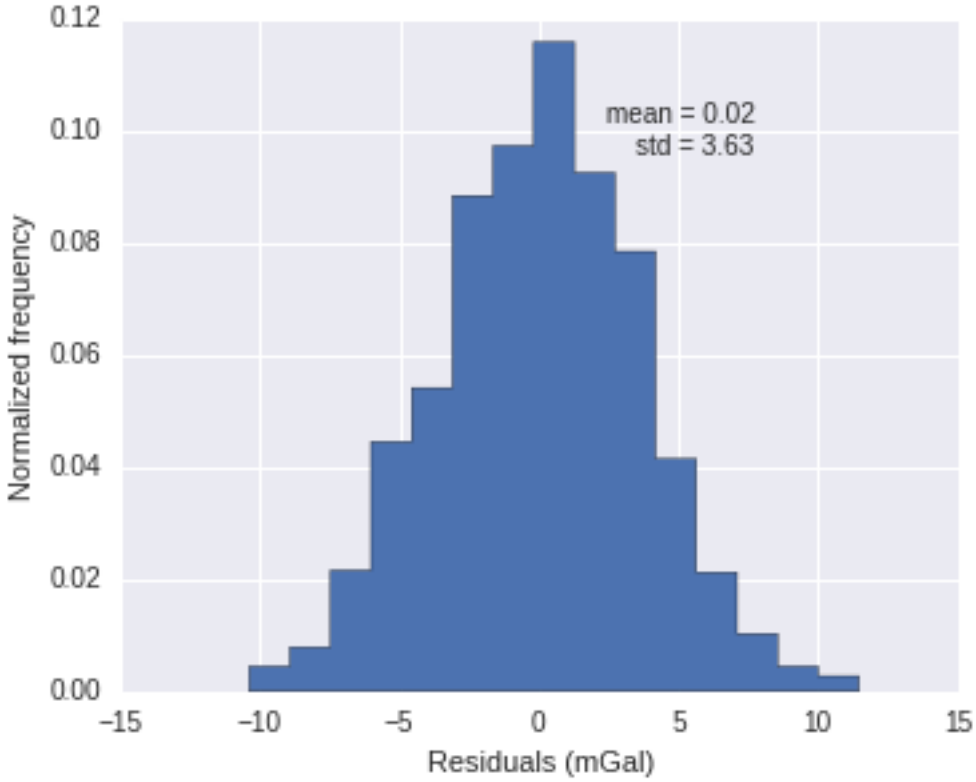


Now we can plot the results for the best solution found. The maps below show the data fit (observed vs predicted), a histogram of the inversion residuals, the estimated Moho depth, and the difference between the estimated and true Moho depths.

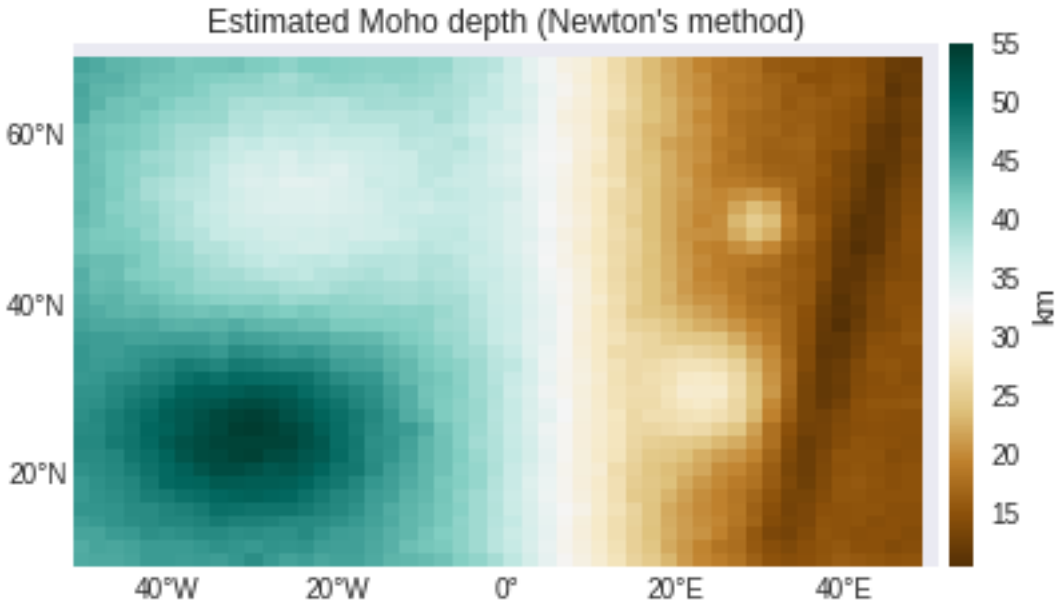
In [35]: `plot_fit(results, bm)`



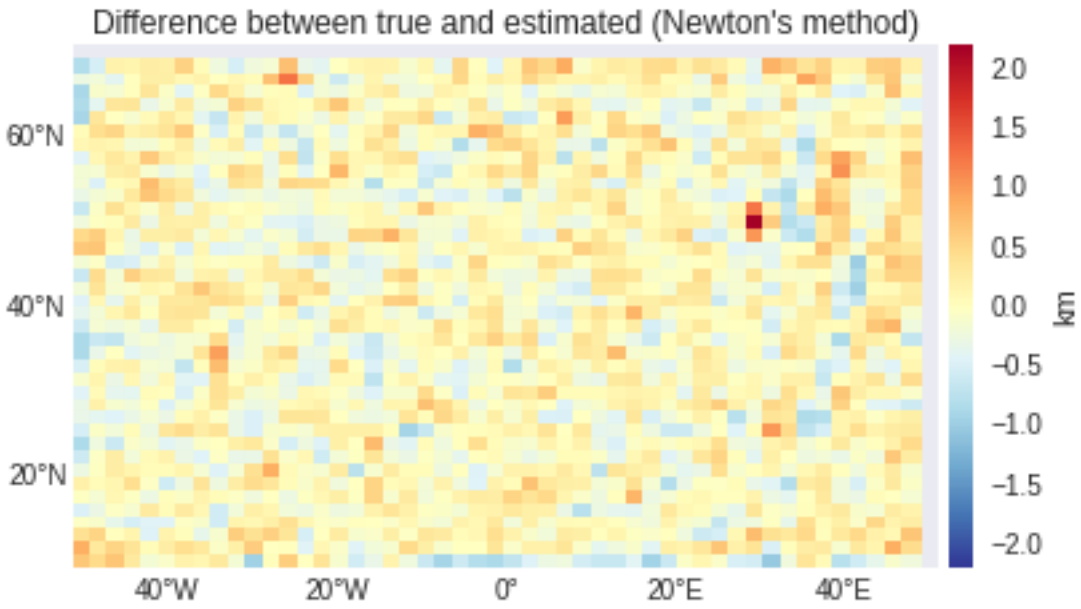
In [36]: `plot_residuals(results)`



In [37]: `plot_estimate(results, bm)`



In [38]: `plot_diff(results, model, bm)`



1.8 Profiling

We'll run the Python profiler (using the `%prun` IPython magic command) to measure how much time is speant on each function call during the inversion. This will allow us to see where time is being spent. I'll test the inversion by Gauss-Newton to see if solving the linear system has any impact on performance.

For consistency, we'll run the inversion using the optimal regularization parameter determined above by cross-validation.

```
In [39]: mu = results['regul_params'][results['best_index']]
```

The following cell runs the profiling on all the code inside it and saves the output to a pretty-printed file and to a raw data format.

```
In [40]: %%prun -q -T results/profiling.txt -D results/profiling-raw.dat
         misfit = MohoGravityInvSpherical(lat, lon, height, data, mesh)
         regul = Smoothness2D(mesh.shape)
         (misfit + mu*regul).config(**results['config']).fit()

*** Profile stats marshalled to file u'results/profiling-raw.dat'.

*** Profile printout saved to text file u'results/profiling.txt'.
```

There are the first N lines of the pretty-printed profiling output. The list of functions is sorted by the time speant inside each specific function (`tottime`).

```
In [41]: !head -n 20 results/profiling.txt

2310514 function calls (2180215 primitive calls) in 42.133 seconds

   Ordered by: internal time

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
        9   39.229    4.359   42.059    4.673 tesseroid.py:324(_forward_model)
    18198    0.138    0.000    0.357    0.000 context.py:139(resolve_data_type)
    18000    0.132    0.000    0.238    0.000 mohoinv.py:476(__getitem__)
    18120    0.121    0.000    0.121    0.000 __init__.py:501(cast)
        8    0.113    0.014    0.113    0.014 executionengine.py:93(finalize_object)
   371163    0.110    0.000    0.111    0.000 {isinstance}
     9716    0.102    0.000    0.209    0.000 instructions.py:12(__init__)
69565/49934    0.088    0.000    0.154    0.000 {method 'format' of 'str' objects}
        8    0.087    0.011    0.087    0.011 passmanagers.py:26(run)
    61709    0.082    0.000    0.119    0.000 StringIO.py:208(write)
    18000    0.076    0.000    0.098    0.000 mesher.py:527(__init__)
30960/21244    0.073    0.000    0.551    0.000 {print}
     9950    0.061    0.000    0.442    0.000 values.py:141(__str__)
    18104    0.055    0.000    0.176    0.000 ctypes_utils.py:40(is_ctypes_funcptr)
    18009    0.049    0.000    0.316    0.000 mohoinv.py:469(next)
```

We'll use the `pstats` module to investigate the raw profiling data for what we want. The three major time sinks in the inversion process that we'll investigate are: **solving sparse linear systems, matrix dot products, forward modeling**. We'll be looking at the cummulative time (`cumtime`) which is the total amount of time spent inside a given function.

After filtering the results to these three specific functions, we'll print the them to separate files for later use.

```
In [42]: def filter_profiling(keys, fname):
             output = 'results/{}'.format(fname)
             with open(output, 'w') as f:
                 # Load the profiling data and set the output file
                 st = pstats.Stats('results/profiling-raw.dat', stream=f)
                 # Filter the data to isolate the time in the desired
```

```
        # function and print to the file
        st.sort_stats('cumtime').print_stats(*keys)
    # Print the file contents
    !cat $output
```

First, investigate how time is spent solving sparse linear systems using conjugate gradient (the `cgs` function in `scipy.sparse.linalg.isolve.iterative`).

```
In [43]: filter_profiling(['scipy', 'cgs'], 'profiling-linsys.txt')

Sat Feb 20 20:06:53 2016    results/profiling-raw.dat

         2310514 function calls (2180215 primitive calls) in 42.133 seconds

   Ordered by: cumulative time
   List reduced from 1597 to 82 due to restriction <'scipy'>
   List reduced from 82 to 1 due to restriction <'cgs'>

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
        8    0.007    0.001    0.021    0.003 /home/leo/bin/anaconda/envs/moho/lib/python2.7/site-packa
```

Now for the dot products of sparse matrices in `scipy.sparse`.

```
In [44]: filter_profiling(['scipy', 'sparse', 'dot'], 'profiling-dot.txt')

Sat Feb 20 20:06:53 2016    results/profiling-raw.dat

         2310514 function calls (2180215 primitive calls) in 42.133 seconds

   Ordered by: cumulative time
   List reduced from 1597 to 82 due to restriction <'scipy'>
   List reduced from 82 to 81 due to restriction <'sparse'>
   List reduced from 81 to 1 due to restriction <'dot'>

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
      200    0.000    0.000    0.007    0.000 /home/leo/bin/anaconda/envs/moho/lib/python2.7/site-packa
```

Finally, investigate the forward modeling using function `gz` of `fatiando.gravmag.tesseroid`.

```
In [45]: filter_profiling(['fatiando', 'tesseroid', 'gz'], 'profiling-forward.txt')

Sat Feb 20 20:06:53 2016    results/profiling-raw.dat

         2310514 function calls (2180215 primitive calls) in 42.133 seconds

   Ordered by: cumulative time
   List reduced from 1597 to 55 due to restriction <'fatiando'>
   List reduced from 55 to 7 due to restriction <'tesseroid'>
   List reduced from 7 to 1 due to restriction <'gz'>

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
        9    0.000    0.000   42.059    4.673 /home/leo/bin/anaconda/envs/moho/lib/python2.7/site-packa
```

Extract the times from each file and store them in a dictionary.

```
In [46]: keys = 'forward dot linsys'.split()
         times = {}
         total_times = []
         for k in keys:
             with open('results/profiling-{}.txt'.format(k)) as f:
                 lines = f.readlines()
             times[k] = float(lines[-3].strip().split()[3])
             total_times.append(float(lines[2].strip().split()[-2]))
         # Check if all total execution times read are equal
         assert np.all(np.equal(total_times[0], total_times))
         total_time = total_times[0]

In [47]: total_time, times

Out[47]: (42.133, {'dot': 0.007, 'forward': 42.059, 'linsys': 0.021})
```

Calculate the percentage of the total time for each function.

```
In [48]: percentage = {k:100*times[k]/total_time for k in keys}

In [49]: for k in keys:
             print('{} = {:.5f} %'.format(k, percentage[k]))

forward = 99.82437 %
dot = 0.01661 %
linsys = 0.04984 %
```

Get the current processor name from the operating system.

```
In [50]: tmp = !cat /proc/cpuinfo | grep "model name"
         processor = tmp[0].split(':')[1].strip()
         print(processor)

Intel(R) Core(TM) i7-3612QM CPU @ 2.10GHz
```

Make the Latex table for the article.

```
In [51]: caption = """
                 Time spent on each function during a single inversion of
                 simple synthetic data.
                 The inversion was performed on a laptop computer with a
                 {processor} processor.
                 The total time for the inversion was {total_time:.3f} seconds.
         """.format(processor=processor, total_time=total_time)
         print(caption)

Total time spent on each function during a single inversion of
        simple synthetic data.
        The inversion was performed on a laptop computer with a
        Intel(R) Core(TM) i7-3612QM CPU @ 2.10GHz processor.
        The total time for the inversion was 42.133 seconds.

In [52]: line = "{func} & {time:.3f} & {percent:.3f}"
         cg = line.format(func="Sparse conjugate gradient",
                          time=times['linsys'], percent=percentage['linsys'])
         dot = line.format(func="Sparse dot product",
                           time=times['dot'], percent=percentage['dot'])
         fwd = line.format(func="Tesseroid forward modeling",
                           time=times['forward'], percent=percentage['forward'])
```

```
In [53]: table = r"""
         \begin{table}
             \centering
             \caption{%s    }
             \label{profiling}
             \begin{tabular}{lcc}
                 Function description & Time (s) & Percentage of total time (\%)\\
                 \hline
                 %s\\
                 %s\\
                 %s\\
                 \hline
             \end{tabular}
         \end{table}
         """ % (caption, cg, dot, fwd)
         print(table)
```

```
\begin{table}
    \centering
    \caption{
        Total time spent on each function during a single inversion of
        simple synthetic data.
        The inversion was performed on a laptop computer with a
        Intel(R) Core(TM) i7-3612QM CPU @ 2.10GHz processor.
        The total time for the inversion was 42.133 seconds.
    }
    \label{profiling}
    \begin{tabular}{lcc}
        Function description & Time & Percentage of total time\\
        \hline
        Sparse conjugate gradient & 0.021 s & 0.050\%\\
        Sparse dot product & 0.007 s & 0.017\%\\
        Tesseroid forward modeling & 42.059 s & 99.824\%\\
        \hline
    \end{tabular}
\end{table}
```

Print this table to a `.tex` file for inclusion in the manuscript.

```
In [54]: with open('../manuscript/profiling.tex', 'w') as f:
             f.write(table)
```