Inverse Problems
Assignment 2

Georgios Sevastakis

April 2024

Introduction to the problem

Our goal is to estimate the magnetization $m(x)$ of a magnetized plate from the observed data acquired at $h = 2cm$ above the plate. Even though our problem is linear, the prior information is not Gaussian and therefore, Tikhonov inversion is avoided and Bayesian inversion is chosen.

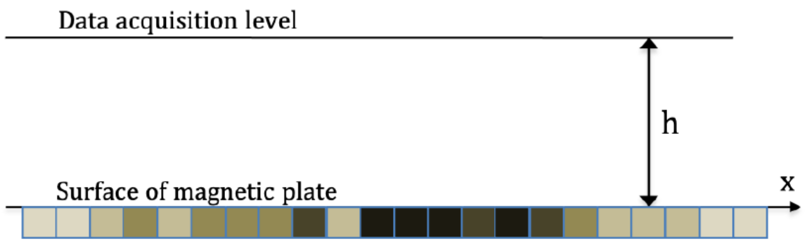

Figure 1: Discretized version of the geometry of the problem (201 discrete points, where the first and last constitute stable boundaries); The magnetic plate is subdivided in stripes with widths with size specified by Equation 8.

The direct problem is formulated as follows

$$d_j = \int_{-\infty}^{\infty} g_j(x) m(x)\, dx \tag{1}$$

where d_j is one datum, $m(x)$ the magnetization at point x and g is expressed as

$$g_j(x) = -\frac{\mu_0}{2\pi} \frac{(x_j - x)^2 - h^2}{\left((x_j - x)^2 + h^2\right)^2} \tag{2}$$

In the discretized setting we have

$$g_i(x_j) = -\frac{\mu_0}{2\pi} \frac{(x_j - x_i)^2 - h^2}{\left((x_j - x_i)^2 + h^2\right)^2} \tag{3}$$

To solve the direct problem, the magnetizations and g are known and we sought for the data d_j. For the inverse problem, the data are known and we sought for the magnetizations magnetizations $m(x)$. As mentioned before, we are going to employ Bayesian inversion. We start off by defining the conditional probability density induced by $\rho(\mathbf{m}, \mathbf{d})$ on the hypersurface $\mathbf{d} = \mathbf{f}(\mathbf{m})$ known as *a posteriori* probability density

$$\sigma(\mathbf{m}) \equiv k\rho(\mathbf{m}, \mathbf{f}(\mathbf{m})) \tag{4}$$

where k is a normalization constant, $\rho(\mathbf{m}, \mathbf{d})$ is the probability density of the model parameters and the data in the space defined by the model parameters and the data.

(5)

Since our problem is linear and the probability distributions of the data and the model parameters are independent and the input probability density of the data is Gaussian, then the equation above simplifies to

$$\sigma(\mathbf{m}) = k \cdot \rho(\mathbf{m}) \frac{L(\mathbf{m})}{\mu(\mathbf{m})} \tag{6}$$

where

$$L(\mathbf{m}) = constant \cdot e^{-\frac{1}{2}(\mathbf{d\_obs} - \mathbf{Gm})^T C_d^{-1} (\mathbf{d\_obs} - \mathbf{Gm})} \tag{7}$$

where d_obs are our initial data stored in 1D array, G is the matrix consisting of the $g_i(x_j)$ elements from equation 3 and m are the magnetizations on the 201 point grid stored in a 1D array as well.

Introduction to the code

Since the code is quite lengthy an introduction was deemed necessary. Having loaded the data, we first initialize the magnetizations randomly by making sure that the probability of finding a boundary at a given point is 12.5% since this constitutes prior knowledge. Then, we spawn initial matrices for the width of the stripes (*w_initial*), for the magnetizations (*m_initial*), for the boundaries (*boundaries*) (matrix constituting of booleans: True if on a boundary, else False, excluding the last point) and for the extended boundaries (*extended_boundaries*) (same as *boundaries*) but includes the last point). Then, we construct the G matrix based on equation 3. Note that we work with *cm* and *nT* units. After that, we are introduced to the *reduce_m*, *extend_m* and *new_model* function, which are to be discussed later. The important thing to note is that *new_model* samples the prior probability density of the model parameters. Ultimately, the *log_Likelihood* function computes the logarithm of the Likelihood (and not the Likelihood directly, so as to avoid overflows) and the *Simulation* function runs the Metropolis algorithm.

1

As discussed in the Introduction of the problem, the Likelihood is given by equation 7, because our data are statistically independent and Gaussian with mean $0\,nT$ and standard deviation $25nT$.

2

Below, we provide the code snippet, which samples the prior $\rho(\mathbf{m})$ to be called iteratively in the Metropolis algorithm.

The horizontal widths w of the physical stripes are distributed according to the exponential probability density

$$f(w) = \frac{1}{w_0} e^{-\frac{w}{w_0}} \tag{8}$$

where w_0, the mean stripe width, has the value $w_0 = 4cm$.

On the other hand, the magnetizations are Gaussian distributed. To sample the prior then, we proceed as explained in the assignment text by performing one of the following three perturbations in each iteration:

- changing the magnetization in one stripe by randomly choosing a point on the discretized model. The new magnetization is taken from a Gaussian distribution with mean 0 and standard deviation $2.5A/cm$.
- adding a new stripe boundary there, where one doesn't already exist and assigning (new) random magnetizations taken from a Gaussian distribution with mean 0 and standard deviation $2.5A/cm$ to the stripes on both sides of it, or
- removing one stripe boundary there, where one doesn't already exist and assigning a (new) magnetization taken from a Gaussian distribution with mean 0 and standard deviation $2.5A/cm$ to the new compound stripe.

Essentially, the code does exactly what was described above, but there are a few details that require a more detailed explanation

- Two functions are introduced, namely *reduce_ m* and *extend_ m*. Both return m, but they refer to different discretizations, namely the former one returns the individual magnetizations as found in **one stripe**, while the latter returns magnetizations as found on **each point of the discretized model**. The whole function then, runs with the m matrix being **reduced**.
- The w matrix carries the information of the widths of the individual stripes and is used mainly to reduce and extend the m matrix as described above.

Let's take a closer look at Listing 1. Initially, we reduce the extended m matrix and randomly choose a type of perturbation. If it is a stripe perturbation, we change the magnetization of a randomly chosen stripe directly.

In case of a stripe boundary perturbation step we choose a point at random. We exploit the fact that (approximately) exponentially distributed stripe thicknesses can be obtained by assuming that the probability that a stripe interface is present at a given position (sample point) is equal to $(0.5cm/w_0) = 0.125$ and independent of the presence of other stripe interfaces. This means that we are to add a boundary (12.5% of the time). If that's the case, we first check if the chosen point constitutes a boundary point. If yes, we make sure that the iteration is not lost by reducing the count by one. Other wise, we proceed by adding a boundary; We assign new magnetizations as described above and we update the *w, boundaries and extended_ boundaries* at that point. Equivalently, in the case of removing a boundary (87.5% of the time), we do so only if the chosen point is a boundary point. Then we assign a new magnetization as described above and update the *w, boundaries and extended_ boundaries* at that point. Ultimately, the *extend_ m* function is called to extend the matrix m to all points of the discretized model.

```python
def new_model(m_new, w_new, N_points, boundaries_new, extended_boundaries_new, i):

    mm = m_new.copy()
    ww = w_new.copy()
    m_new = reduce_m(mm, ww)

    perturbation_type = np.random.choice(['stripe_magnetization', '
    boundary_perturbation'])

    if perturbation_type == 'stripe_magnetization':
        stripe_ID = np.random.randint(1, len(w_new))
        m_new[stripe_ID - 1] = np.random.normal(0, 2.5)
    else:
        point_index = np.random.randint(1, N_points - 1)                    #Leaving out the boundaries of the plate
        add_boundary = np.random.choice([True, False], p=[0.125, 0.875])
        if add_boundary:
            # Add a boundary
            if boundaries_new[point_index] == True:
                if i == 0:
                    pass
                else:
                    i -= 1
            else:
                stripe_ID = boundaries_new[:point_index].count(True)
                index_i = [i for i, v in enumerate(boundaries_new) if v == True][stripe_ID - 1]
                index_f = [i for i, v in enumerate(extended_boundaries_new) if v == True][stripe_ID]
                w_i = (point_index - index_i) * 0.5
                w_f = (index_f - point_index) * 0.5
                w_new[stripe_ID - 1] = w_i
                w_new = np.insert(w_new, stripe_ID, w_f)
                m_new[stripe_ID - 1] = np.random.normal(0, 2.5)
                m_new = np.insert(m_new, stripe_ID, np.random.normal(0, 2.5))
                boundaries_new[point_index] = True                          #set a new boundary at that point
                extended_boundaries_new[point_index] = True                 #set a new boundary at that point

        else:
            if boundaries_new[point_index] == False:
                if i == 0:
                    pass
                else:
                    i -= 1
            else:
                stripe_ID = boundaries_new[:point_index].count(True)
                index_i = [i for i, v in enumerate(boundaries_new) if v == True][stripe_ID - 1]
                index_f = [i for i, v in enumerate(extended_boundaries_new) if v == True][stripe_ID + 1]
                w_f = (index_f - index_i) * 0.5
                w_new[stripe_ID - 1] = w_f
                w_new = np.delete(w_new, stripe_ID)
                m_new[stripe_ID - 1] = np.random.normal(0, 2.5)
                m_new = np.delete(m_new, stripe_ID)
                boundaries_new[point_index] = False                         #set a new boundary at that point
                extended_boundaries_new[point_index] = False                #set a new boundary at that point
    mmm = extend_m(m_new, w_new)
    return mmm, w_new, boundaries_new, extended_boundaries_new
```

Listing 1. Function that samples the prior $\rho(\mathbf{m})$.

3

The null–information probability density $\mu(\mathbf{m})$ for this problem is a constant since the coordinate system we are working on is the Cartesian.

4

Below, we provide the code snippet regarding the Metropolis algorithm simulation. In each iteration, we compare two sets of magnetizations: the current one and the proposed one. The latter constitutes the former, with the difference that a parturbation has taken place. Subsequently, their respective Likelihoods are computed (the logarithms to be specific) and a criterion is set into place. If the Likelihood of the proposed is greater than that of the current's then the proposed model will be accepted. Otherwise, the proposed model is accepted with a propability given by the ratio of the likelihoods. The closer the new likelihood value to the older one, the higher the chance it will be accepted.

```python
m_initial = extend_m(m_initial, w_initial)

def Simulation(m, w, N_points, boundaries, extended_boundaries, d_obs, G, covariance_inv):

    N_t = 100_000
    model_samples = []

    for i in tqdm(range(N_t)):
        current_m, current_w, current_boundaries, current_ext_boundaries = m, w, boundaries, extended_boundaries
        proposed_m, proposed_w, proposed_boundaries, proposed_ext_boundaries = new_model(current_m.copy(), current_w.copy(), N_points, current_boundaries.copy(), current_ext_boundaries.copy(), i)

        logL_old = log_Likelihood(current_m.copy(), d_obs, G, covariance_inv)
        logL_new = log_Likelihood(proposed_m.copy(), d_obs, G, covariance_inv)

        ratio = np.exp(logL_new - logL_old)
        p_accept = min(1, float(ratio))

        Criterion = np.random.choice([True, False], p=[p_accept, 1. - p_accept])
        if Criterion:
            m, w, boundaries, extended_boundaries = proposed_m, proposed_w, proposed_boundaries, proposed_ext_boundaries
            model_samples.extend(m)
        else:
            m, w, boundaries, extended_boundaries = current_m, current_w, current_boundaries, current_ext_boundaries

    proposed_m = np.array([proposed_m])
    model_samples = np.array([model_samples])
    return proposed_m.T, proposed_w, model_samples.reshape(len(m), -1)
```

Listing 2. Function that runs the Metropolis algorithm.

Running the code, we visualize the results

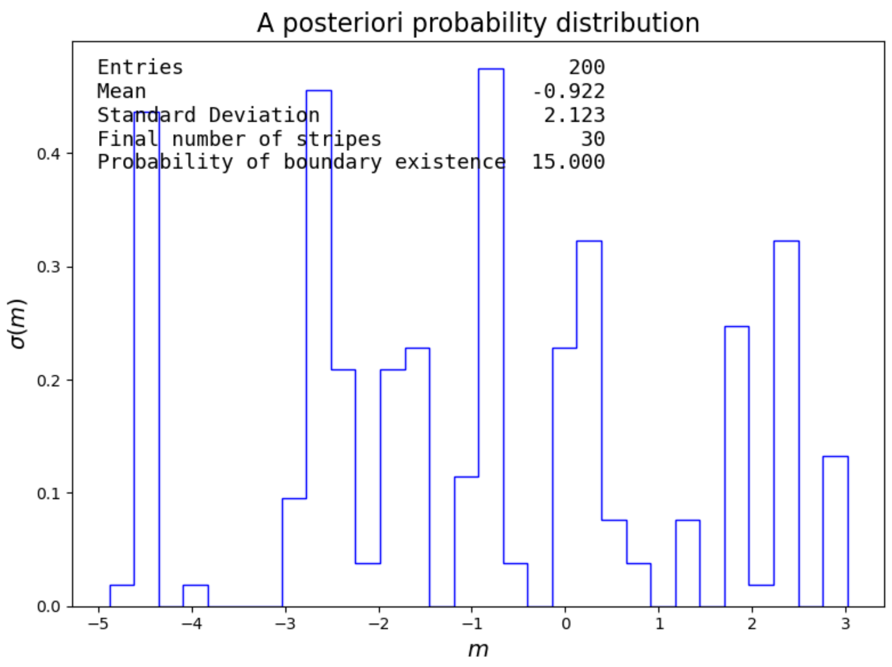

Figure 2: Normalized a posteriori probability density $\sigma(\mathbf{m})$

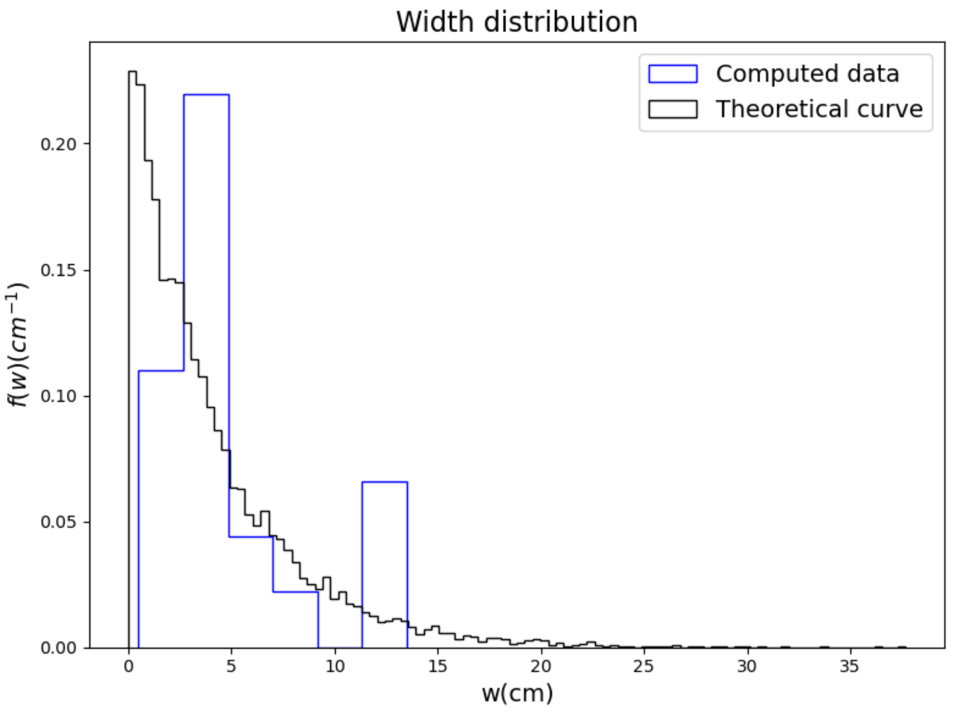

Figure 3: Histogram presenting the distribution of the widths as described by equation 8. From the plot we conclude that the sampling algorithm was a success.

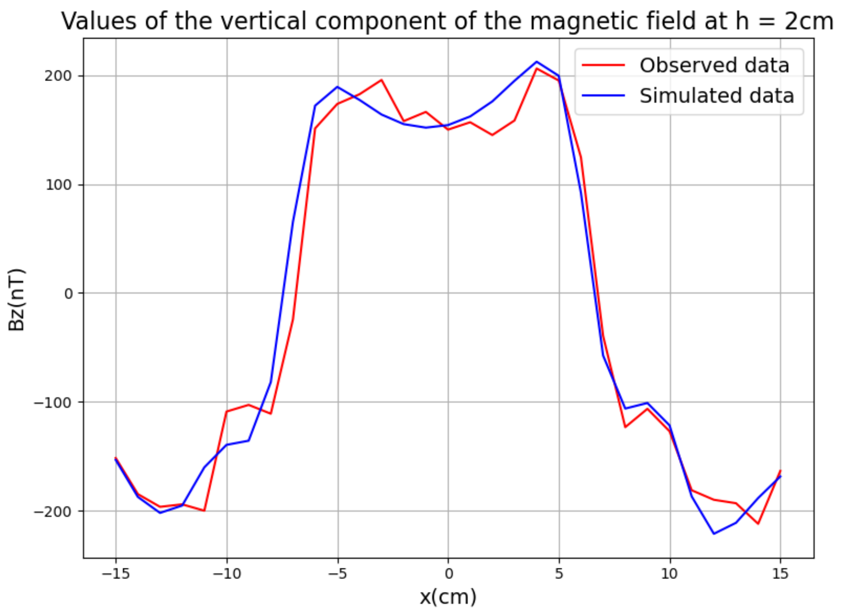

Figure 4: Graph presenting the data measured compared to the computed data by solving the direct problem with the magnetization values acquired by the inverse.

5

Below we present a graph with the uncertainties of the model parameters. The uncertainties we computed by storing the accepted model parameters in a matrix throughout the whole duration of the 'movie'. After the 'movie' was over, we computed the uncertainties of the model parameters by comparing the model parameters from all different accepted models.Specifically, we read off the uncertainties from the covariance matrix.

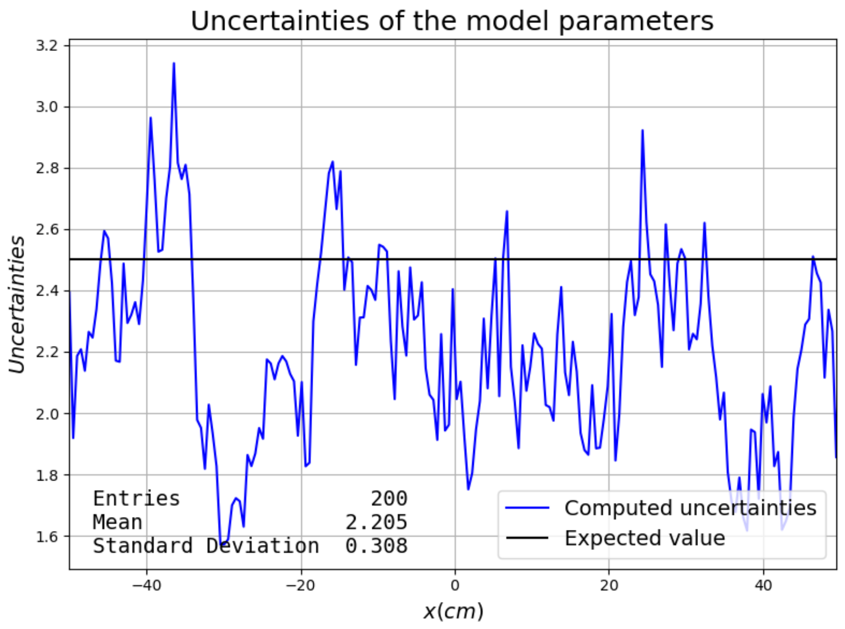

Figure 5: Uncertainties of the model parameters m.

Notice that the uncertainties are within 1σ from the value $\sigma = 2.5A/cm$ which was our prior knowledge.