Assignment 2 - Inverse Problems 2023 - Emilie Jessen

In [ ]:

```python
import numpy as np
import matplotlib as mpl
import matplotlib.pyplot as plt
from numpy import linalg
```

In [ ]:

```python
# Set som plotting standards:
font = {'family' : 'serif',
        'weight' : 'normal',
        'size'   : 18}
mpl.rc('font', **font)

axes = {'facecolor': 'ghostwhite',
        'grid': 'True'}
mpl.rc('axes', **axes)
```

In [ ]:

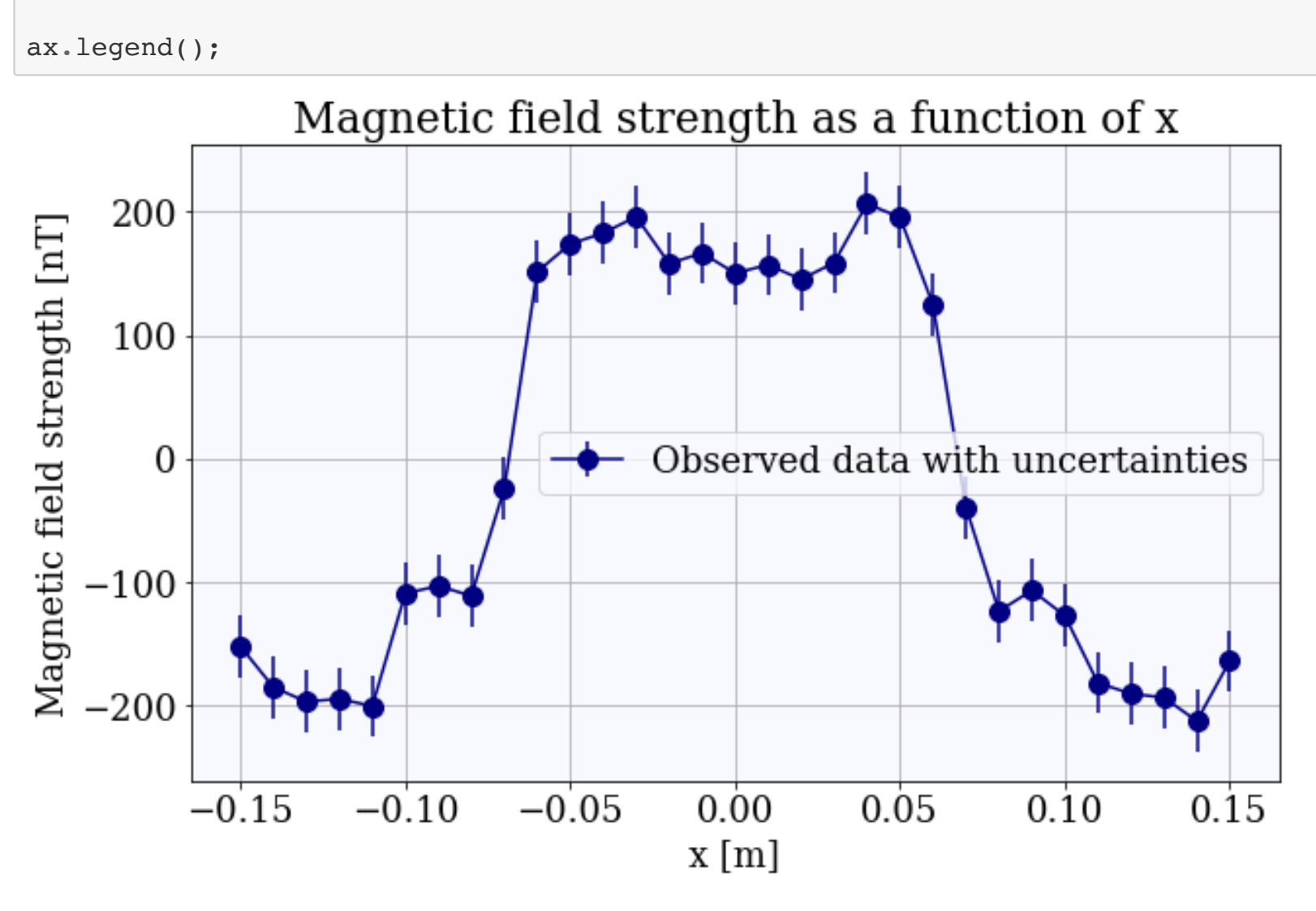
```python
# Loading the data
data = np.loadtxt("dataM.txt")
np.random.seed(20)
```

In [ ]:

```python
# Importing Tikhonov functions from assignment 1:

def Tikhonov(epsilon, G, t_obs):
    """Function to estimate the model m_tilde, i.e. the position of the grey using Tikhonov regularization.
    For each epsilon the function calulates m_tilde using the normal eqs.
    For each epsilon the function also calculates the error norm |t_obs - Gm|^2."""

    # Save model parameters i.e. the estimate of the position of the box:
    # Save the error norm |t_obs - Gm|^2
    model = []
    error_norm = []

    # Loop through the different epsilon values:
    for i in epsilon:
        m_tilde = np.linalg.inv(G.T @ G + i**2 * np.eye(G.shape[1])) @ G.T @ t_obs     # Normal egs.
        model.append(m_tilde)
        error_norm.append(np.linalg.norm(t_obs - G @ m_tilde)**2)

    return model, error_norm


def optimum(epsilon, error_norm, criterium, model):
    """Function to find the best epsilon value that satisfies the criterion.
    The function returns the index and value of the epsilon that is closest to satisfy the criterion
     and the corresponding model parameters. """

    # Finding the index of the epsilon that is closest to satisfy the criterium:
    eps_best_idx = np.argmin(abs(error_norm - criterium))

    # Getting the best epsilon and corresponding model parameters:
    epsilon_best = epsilon[eps_best_idx]
    m_best = model[eps_best_idx]

    return eps_best_idx, epsilon_best, m_best
```

In [ ]:

```python
# Visualizing the data
x_data = data[:,0] / 100                    # m
m_data = data[:,1]                          # nT -> N / (A * m)
m_data_sigma = np.ones(len(m_data)) * 25 # nT

fig, ax = plt.subplots(figsize=(10, 6))
ax.errorbar(x_data, m_data, yerr=m_data_sigma, fmt='-o', ms=10, color='navy', label="Observed data with uncertainties"
)
ax.set(xlabel = "x [m]",
       ylabel = "Magnetic field strength [nT]",
       title = "Magnetic field strength as a function of x")

ax.legend();
```



1. The likelihood function:

$$L(\mathbf{m}) = c \cdot \exp \frac{1}{2}(\mathbf{d_{obs}} - \mathbf{Gm})^T \mathbf{C_d}^{-1}(\mathbf{d_{obs}} - \mathbf{Gm})$$

The likelihood function is given by the expression above, as the uncertainties on the data and surface dipole magnetization follows a Gaussian distribution, and the expression for the likelihood is of this form when the problem is Gaussian. The difference $(\mathbf{d_{obs}} - \mathbf{Gm})$ describes how far the computed data is from the observed data. The covariance matrix $\mathbf{C_d}$ for the data (and for the noise) is in this case diagonal, as the uncertainties on the measured data are statiscally independent.

1. The function to sample the prior distribution is in the code below.

In [ ]:

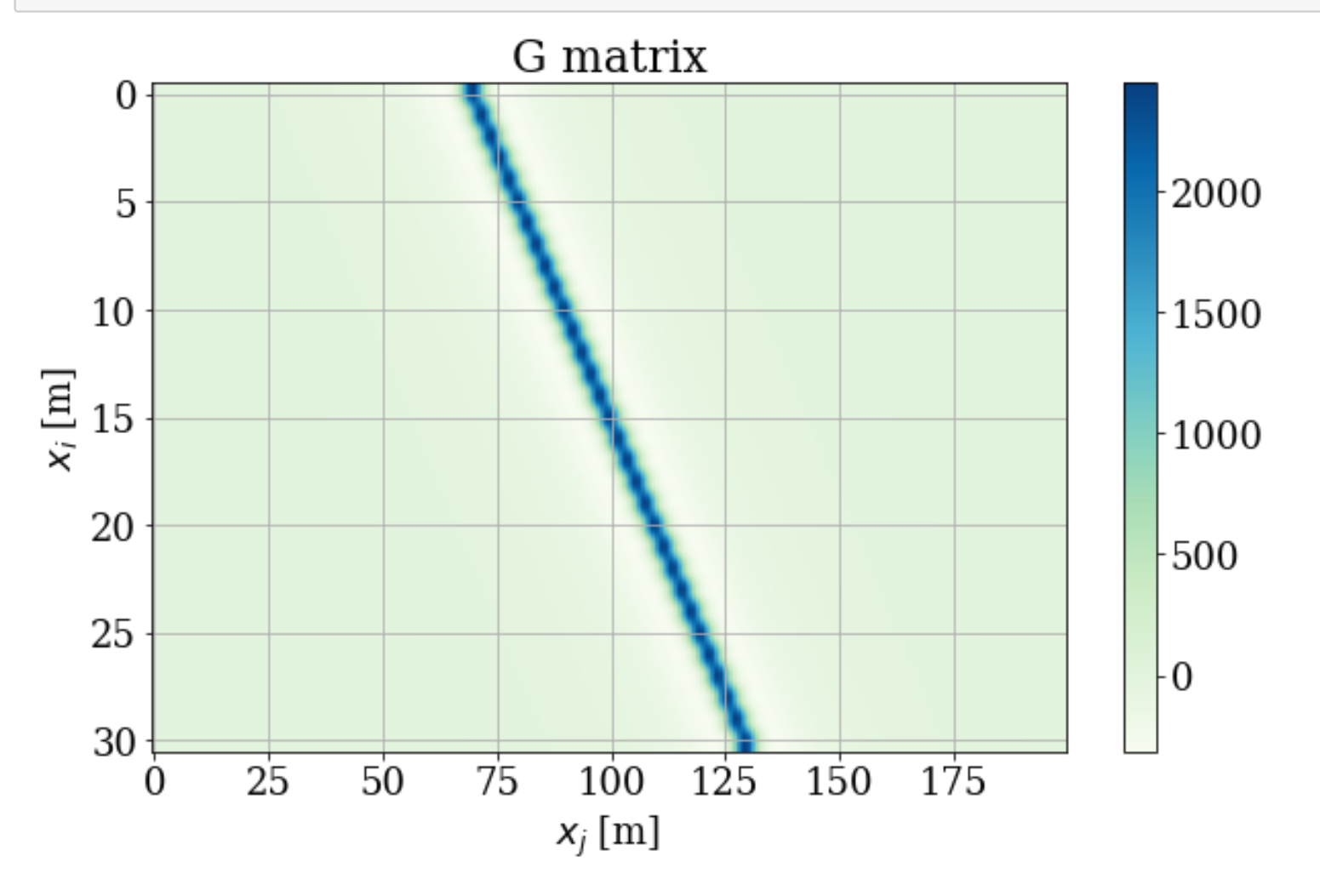
```python
N_bands = 200
mag_sigma = 0.025  # A / m

def pertubation(plate):

    new_plate = plate.copy()

    # Choose a random band
    idx = np.random.randint(0, N_bands)
    stripe_idx = np.where(new_plate == new_plate[idx])[0]

    if idx == 199:
        return new_plate

    # Choose the type of pertubation
    # If the probability is less than 0.5, we perform a pure stripe magnetization pertubation
    if np.random.uniform() < 0.5:
        new_plate[stripe_idx] = np.random.normal(0, mag_sigma)

    # Else we perform a boundary pertubation
    else:

        # First we check if the band is a boundary band
        boundary = False
        if idx == stripe_idx[-1]:
            boundary = True

        # Adding a stripe boundary with probability 0.125

        if np.random.rand() <= 0.125:
            if boundary == False:

                m1 = np.random.normal(0, mag_sigma)
                m2 = np.random.normal(0, mag_sigma)

                new_plate[stripe_idx[0]:idx+1], new_plate[idx+1:stripe_idx[-1]] = m1, m2

        # Removing a stripe boundary
        else:
            if boundary == True:
                old_stripe_idx = np.where(new_plate == new_plate[idx+1])[0]

                m = np.random.normal(0, mag_sigma)
                new_plate[stripe_idx], new_plate[old_stripe_idx] = m, m

    return new_plate
```

1. The null information distribution The null information distribution is a constant. The magnetization values can take all values on the real axis. The magnetization values are Gaussian distributed, and in the case where we don't have any information the standard deviation will approach infinity, whereby the distribution will become a constant. In the Metropolis-Hastings algorithm, we take advantage of the fact, that the acceptance probability is a ratio, whereby the constant will be divided out, and we don't need to determine its value.

In the code below, the covariance matrix is defined, as a diagonal matrix, as the uncertainties on the observed data are independent. The G matrix mapping the model plate to the data space is computed from eq. 3.

In [ ]:

```python
# Create the covariance matrix as a diagonal matrix as the observed data uncertainties are independent
C = np.diag(m_data_sigma**2)

# Create the G matrix from eq. 3
h = 2 / 100    # m
mu_0 = 4 * np.pi * 1e-7  #  H/m = N * A**-2
x_i = x_data + 0.5        # m
x_j = np.linspace(0, 1, 200)  # m

G = np.zeros((len(x_i), len(x_j)))
G = - mu_0 / (2 * np.pi) * ((x_i[:,None] - x_j[None,:])**2 - h**2) / \
    ((x_i[:,None] - x_j[None,:])**2 + h**2)**2

# 1e9 to get right order of magnitude in computed data (nT)
# Resolution of 0.5 cm (0.005m).
# Unit of G: kg / (s**2 * A**2 * m)
G *= 1e9  * 0.005

# Show the G matrix
fig, ax = plt.subplots(figsize=(10, 6))
im = ax.imshow(G, cmap='GnBu', aspect='auto')
fig.colorbar(im)
ax.set(xlabel = r'$x_j$ [m]',
       ylabel = r'$x_i$ [m]',
       title = r'G matrix');
```

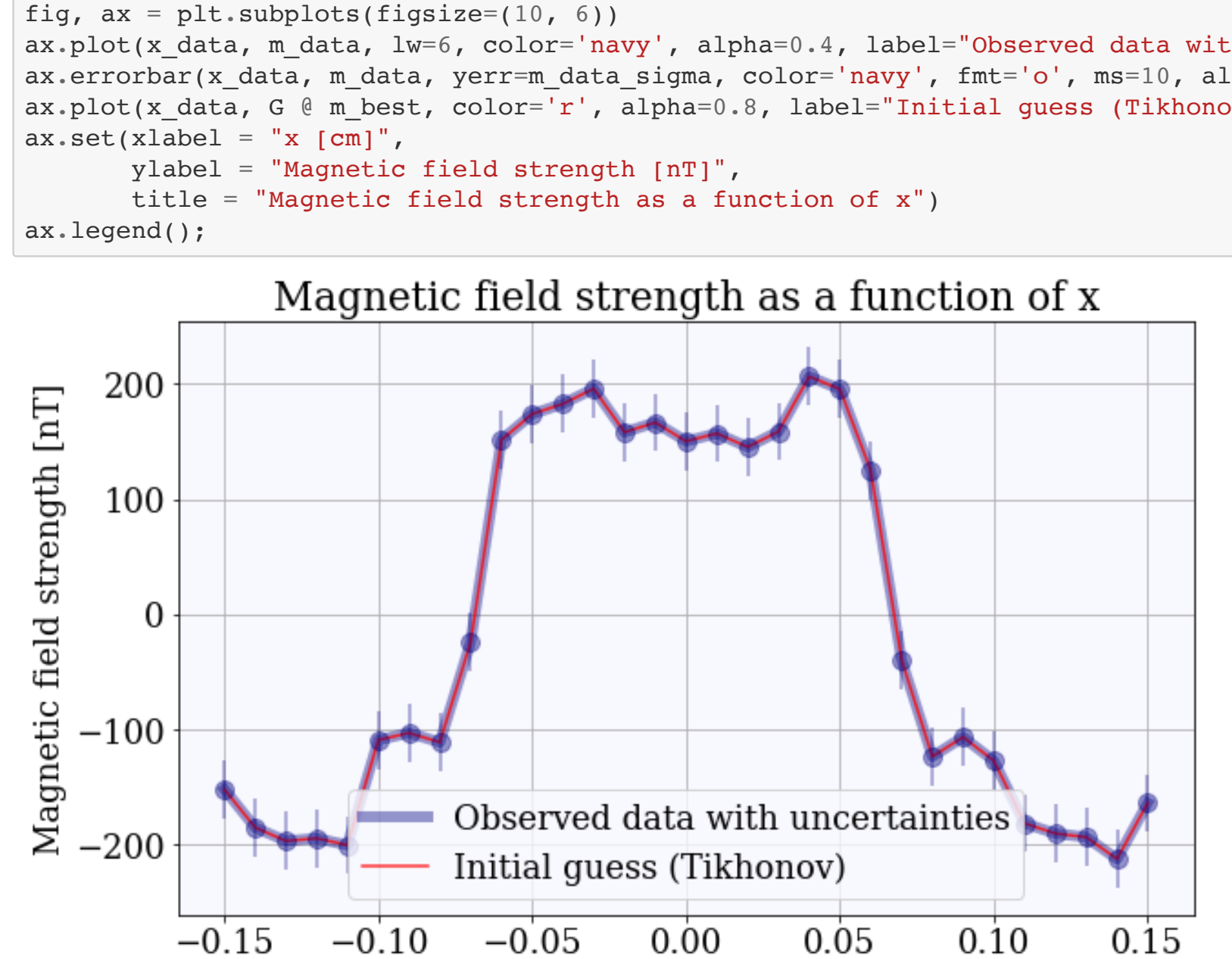


The interpretation of the G-matrix: The bands x_j contributing the most to the data points x_i are the ones in closest proximity (bands with the shortest distance to the data point). The bands furthest out to the sides contributes next to nothing.

In the code below, Tikhonov regularization is used for computed an initial guess on the plate. As the Tikhonov regularization has no knowlegde about our prior distribution, the initial guess will not follow that distribution and the Metropolis-Hastings algorithm is used to compute a model for the plate, that will follow the prior distribution.

In [ ]:

```python
# The criterion for choosing regularization parameter epsilon: N * sigma^2
criterium = len(m_data) * m_data_sigma**2

# Different values of epsilon to try in Tikhonov regularization:
epsilon = np.linspace(0.1, 10, 31)

# Compute the model for each epsilon and the error norm:
model, error_norm = Tikhonov(epsilon, G, m_data)

# Finding the index of the epsilon that is closest to satisfy the criterium:
eps_best_idx, epsilon_best, m_best = optimum(epsilon, error_norm, criterium, model)

# Plotting the model parameters for the best epsilon:
fig, ax = plt.subplots(figsize=(10, 6))
ax.plot(x_data, m_data, lw=6, color='navy', alpha=0.4, label="Observed data with uncertainties")
ax.errorbar(x_data, m_data, yerr=m_data_sigma, color='navy', fmt='o', ms=10, alpha=0.4)
ax.plot(x_data, G @ m_best, color='r', alpha=0.8, label="Initial guess (Tikhonov)")
ax.set(xlabel = "x [cm]",
       ylabel = "Magnetic field strength [nT]",
       title = "Magnetic field strength as a function of x")
ax.legend();
```



1. The Metropolis-Hastings algorithm In the code below, the Metropolis-Hastings algorithm is defined. It takes the plate generated by Tikhonov regularization as an initial guess on the plate and generates models distributed according to the a posterior distribution given by:

$$\sigma(\mathbf{m}) = \frac{\rho(\mathbf{m})L(\mathbf{m})}{\mu(\mathbf{m})}$$

where the product in the numerator is computed by calculating the likelihood function $L(\mathbf{m})$ for a model $\mathbf{m}$, that is following the prior distribution $\rho(\mathbf{m})$. The null information distribution $\mu(\mathbf{m})$, which is constant, is divided out, when calculating the acceptance ratio.

In [ ]:

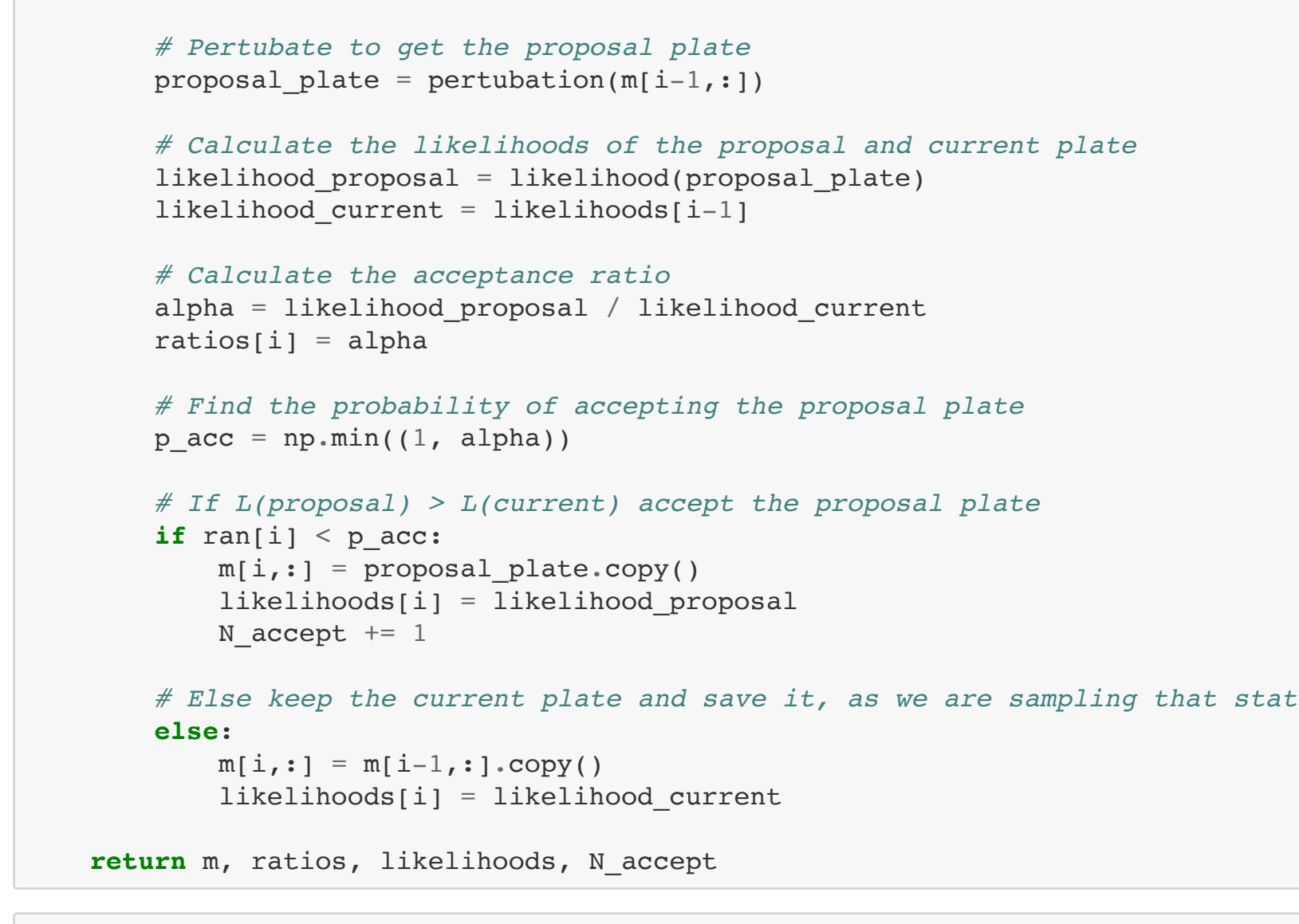
```python
# Defining the likelihood function to be used in the Metropolis-Hastings algorithm
def likelihood(m):
    """Function to calculate the likelihood of the model parameters m given the observed data m_data. """
    diff = m_data - G @ m
    L = np.exp(-0.5 * diff.T @ np.linalg.inv(C) @ diff)
    return L


def metroplis_hastings(plate, N):
    """Function to perform the Metropolis-Hastings algorithm."""

    # Creating matrix to save the model parameters for each iteration
    m = np.zeros((N, len(plate)))

    # Save the initial guess
    m[0,:] = plate.copy()

    # Compute array of random numbers to be used in the algorithm. Faster than calculating one at a time
    ran = np.random.rand(N)

    # Save the number of accepted steps, the acceptance ratios and the likelihoods
    N_accept = 0
    ratios = np.zeros(N)
    likelihoods = np.zeros(N)
    likelihoods[0] = likelihood(plate)

    for i in range(1, N):

        # Pertubate to get the proposal plate
        proposal_plate = pertubation(m[i-1,:])

        # Calculate the likelihoods of the proposal and current plate
        likelihood_proposal = likelihood(proposal_plate)
        likelihood_current = likelihoods[i-1]

        # Calculate the acceptance ratio
        alpha = likelihood_proposal / likelihood_current
        ratios[i] = alpha

        # Find the probability of accepting the proposal plate
        p_acc = np.min((1, alpha))

        # If L(proposal) > L(current) accept the proposal plate
        if ran[i] < p_acc:
            m[i,:] = proposal_plate.copy()
            likelihoods[i] = likelihood_proposal
            N_accept += 1

        # Else keep the current plate and save it, as we are sampling that state one more time
        else:
            m[i,:] = m[i-1,:].copy()
            likelihoods[i] = likelihood_current

    return m, ratios, likelihoods, N_accept
```

In [ ]:

```python
# Running the Metropolis-Hastings algorithm generating models according to the posterior distribution
N_runs = 100_000
m, ratios, likelihoods, N_accept = metroplis_hastings(m_best, N_runs)
```

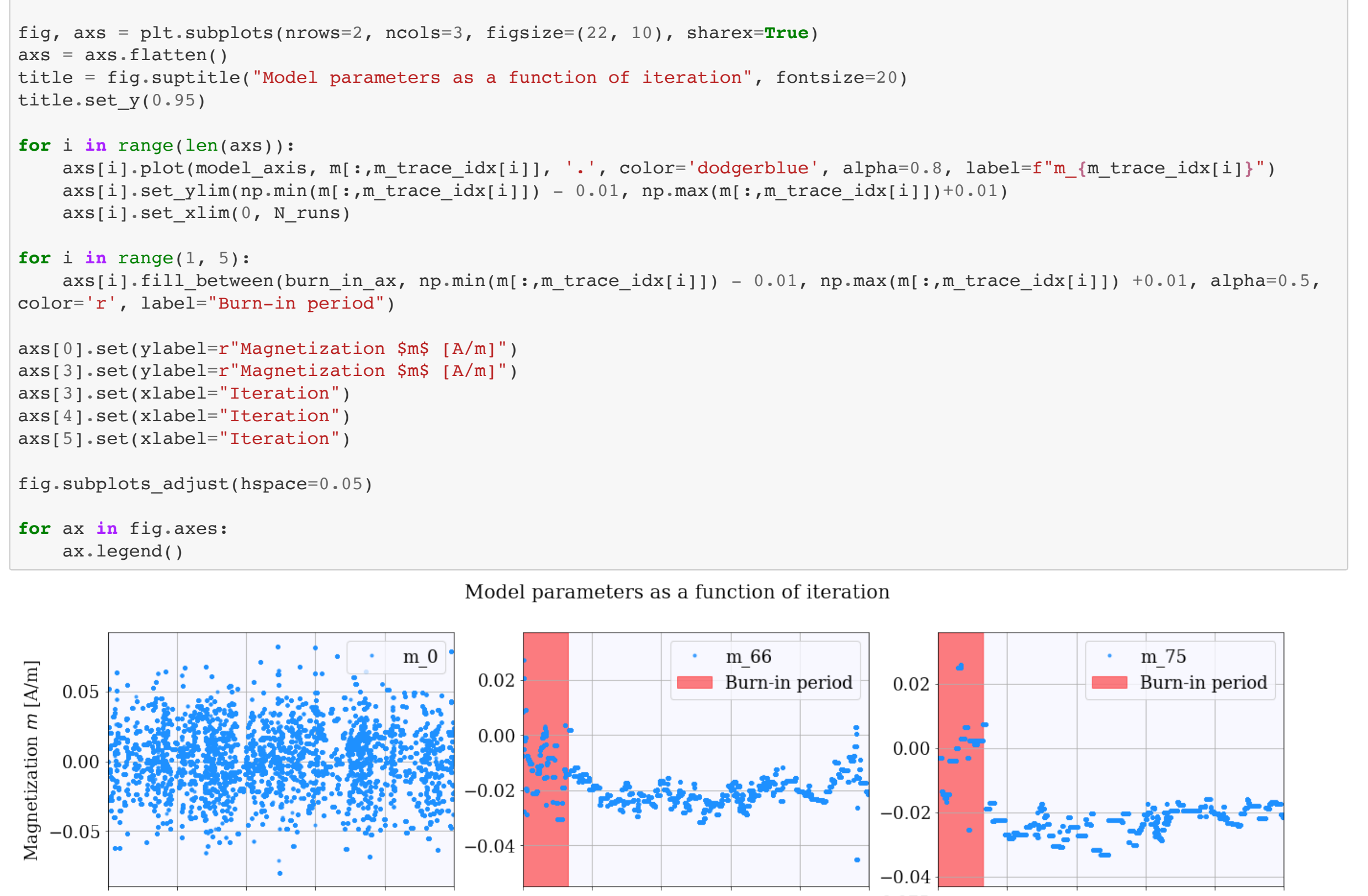
In [ ]:

```python
# Plotting the last model for visulaization
fig, ax = plt.subplots(figsize=(10, 6))
ax.plot(x_data, G @ m[-2,:], color='dodgerblue', lw=6, alpha=0.6, label="Metropolis-Hastings")
ax.plot(x_data, m_data, color='navy', label="Observed data")
ax.errorbar(x_data, m_data, yerr=m_data_sigma, fmt='o', ms=10, color='navy', label="Observed data")
ax.set(xlabel = "x [m]",
       ylabel = "Magnetic field strength [nT]",
       title = "Magnetic field strength as a function of x")
plt.legend();
```



In [ ]:

```python
model_axis = np.linspace(0, N_runs, N_runs)
m_trace_idx = [0, 66, 75, 86, 126, 199]
burn_in = 13000
burn_in_ax = np.linspace(0, burn_in, burn_in)

# Plotting six model parameters for visualization and to determine the burn-in period

fig, axs = plt.subplots(nrows=2, ncols=3, figsize=(22, 10), sharex=True)
axs = axs.flatten()
title = fig.suptitle("Model parameters as a function of iteration", fontsize=20)
title.set_y(0.95)

for i in range(len(axs)):
    axs[i].plot(model_axis, m[:,m_trace_idx[i]], '.', color='dodgerblue', alpha=0.8, label=f"m_{m_trace_idx[i]}")
    axs[i].set_ylim(np.min(m[:,m_trace_idx[i]]) - 0.01, np.max(m[:,m_trace_idx[i]])+0.01)
    axs[i].set_xlim(0, N_runs)

for i in range(1, 5):
    axs[i].fill_between(burn_in_ax, np.min(m[:,m_trace_idx[i]]) - 0.01, np.max(m[:,m_trace_idx[i]]) +0.01, alpha=0.5,
color='r', label="Burn-in period")

axs[0].set(ylabel=r"Magnetization $m$ [A/m]")
axs[3].set(ylabel=r"Magnetization $m$ [A/m]")
axs[3].set(xlabel="Iteration")
axs[4].set(xlabel="Iteration")
axs[5].set(xlabel="Iteration")

fig.subplots_adjust(hspace=0.05)

for ax in fig.axes:
    ax.legend()
```

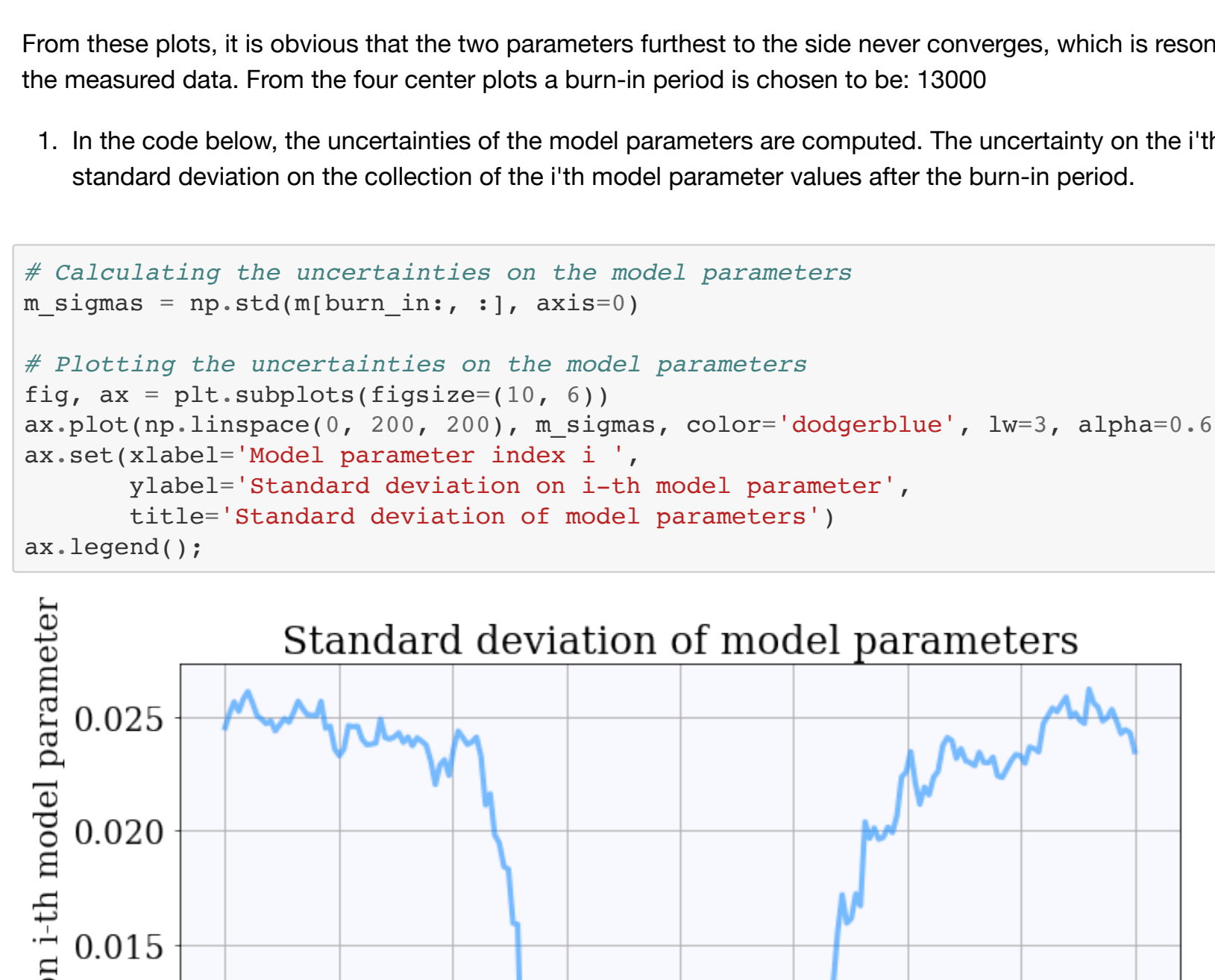


From these plots, it is obvious that the two parameters furthest to the side never converges, which is resonable as they are the ones contributing the least to the measured data. From the four center plots a burn-in period is chosen to be: 13000

1. In the code below, the uncertainties of the model parameters are computed. The uncertainty on the i'th model parameter is computed by taking the standard deviation on the collection of the i'th model parameter values after the burn-in period.

In [ ]:

```python
# Calculating the uncertainties on the model parameters
m_sigmas = np.std(m[burn_in:, :], axis=0)

# Plotting the uncertainties on the model parameters
fig, ax = plt.subplots(figsize=(10, 6))
ax.plot(np.linspace(0, 200, 200), m_sigmas, color='dodgerblue', lw=3, alpha=0.6, label="Std")
ax.set(xlabel='Model parameter index i ',
       ylabel='Standard deviation on i-th model parameter',
       title='Standard deviation of model parameters')
ax.legend();
```



The uncertainties on the model parameter values are largest to the side, where the magnetiszation value of the bands contributes the least to the data. In the middle where the bands are contributing the most, the uncertainties decreases, as the model "are more sure" of those values.

In the code below, we are checking if the length of the model stripes are distributed according to the exponential probability density:

$$f(\omega) = \frac{1}{\omega_0} \exp - \frac{\omega}{\omega_0}$$

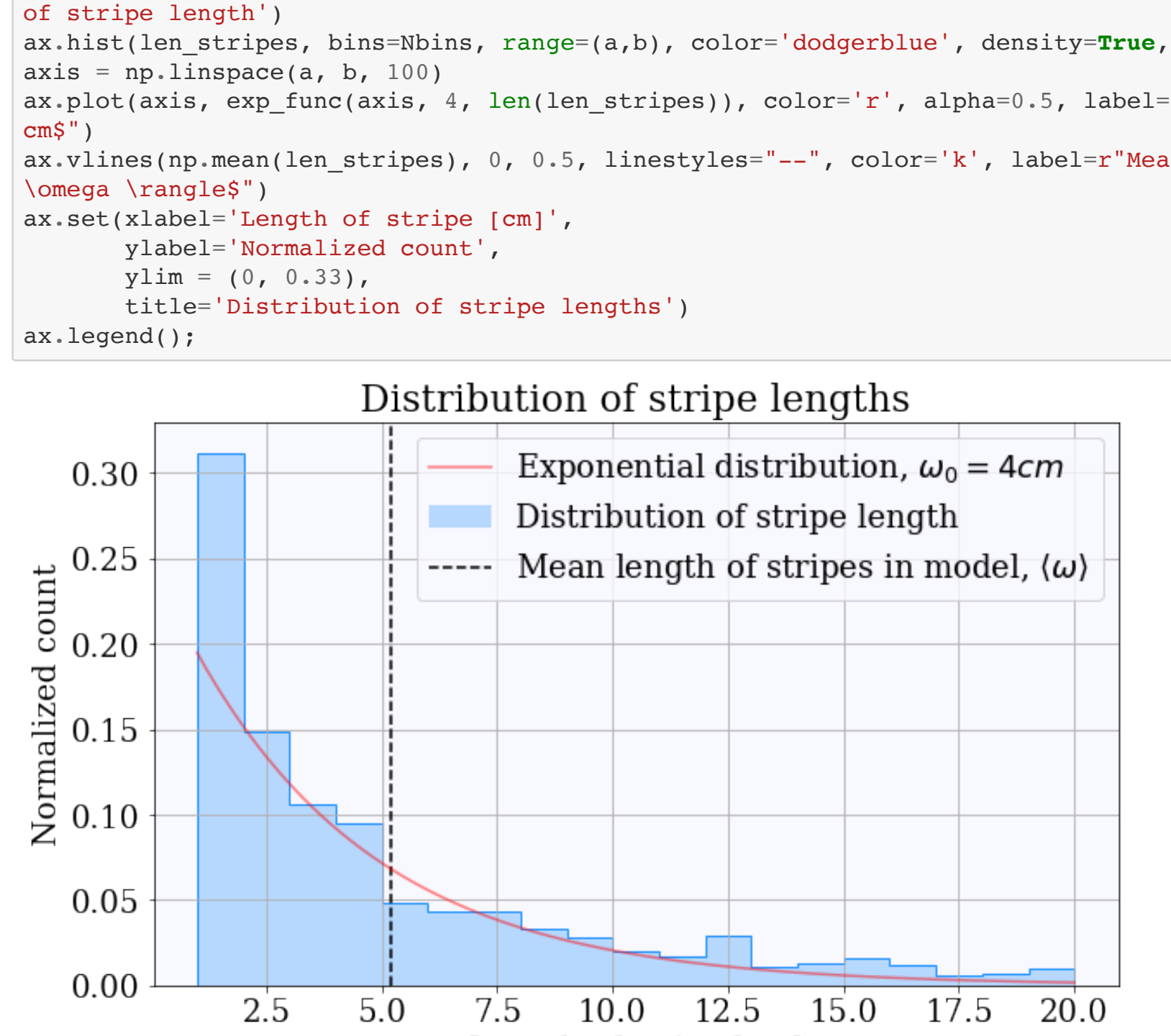
where ω_0 , the mean stripe width has the value: 4cm

In [ ]:

```python
# Calculating the number of stripes and the length of each stripe
N_stripes = []
len_stripes = []

for i in range(burn_in, N_runs):
    vals = np.unique(m[i,:], return_counts=True)[1]
    N_stripes.append(len(vals))
    len_stripes.extend(vals)
```

In [ ]:

```python
def exp_func(x, a, N):
    return 1/a * np.exp(-x/a)

Nbins = 19
a, b = 1, 20

# Visualizing the number of stripes
fig, ax = plt.subplots(figsize=(10, 6))
hist = ax.hist(len_stripes, bins=Nbins, range=(a,b), color='dodgerblue', density=True, alpha=0.3, label='Distribution
of stripe length')
ax.hist(len_stripes, bins=Nbins, range=(a,b), color='dodgerblue', density=True, histtype="step")
axis = np.linspace(a, b, 100)
ax.plot(axis, exp_func(axis, 4, len(len_stripes)), color='r', alpha=0.5, label=r"Exponential distribution, $\omega_0=4
cm$")
ax.vlines(np.mean(len_stripes), 0, 0.5, linestyles="--", color='k', label=r"Mean length of stripes in model, $\langle
\omega \rangle$")
ax.set(xlabel='Length of stripe [cm]',
       ylabel='Normalized count',
       ylim = (0, 0.33),
       title='Distribution of stripe lengths')
ax.legend();
```



The distribution of stripe lengths follows to a large degree an exponential function, but the mean length of the stripes is a bit larger than the theoretical value.