Lab3_MCMC_TotallySolved-exp

May 6, 2020

```
In [1]: import numpy as np
        import scipy.stats
        import matplotlib.pyplot as plt
```

1 Markov Chain Monte Carlo

1.1 1. Implement Metropolis-Hasting

Implement the Metropolis-Hasting algorithm to estimate the distributions of parameters given a set of observations and a prior belief about the distribution that may have generated your data.

Consider a set of observations generated according to a mixture of 3 two-dimensional Gaussians:

$$p(x \mid \theta) = \sum_{k=1}^{3} w_i \cdot \mathcal{N}(\mathbf{x} \mid \mu_k, \Sigma_k),$$

where $\mu_k = [\mu_k^{(1)}, \mu_k^{(2)}]$ and Σ_k is the 2×2 precision matrix. Here $\theta = \{(w_i, \mu_i, \Sigma_i)\}_{i=1}^{3}$.

```
In [ ]:
```

1.1.1 1.1. Generate dataset

Choose a set of parameters and generate samples from the resulting distribtion. The goal is to learn a distribution over the parameters space that capture well the real parameters.

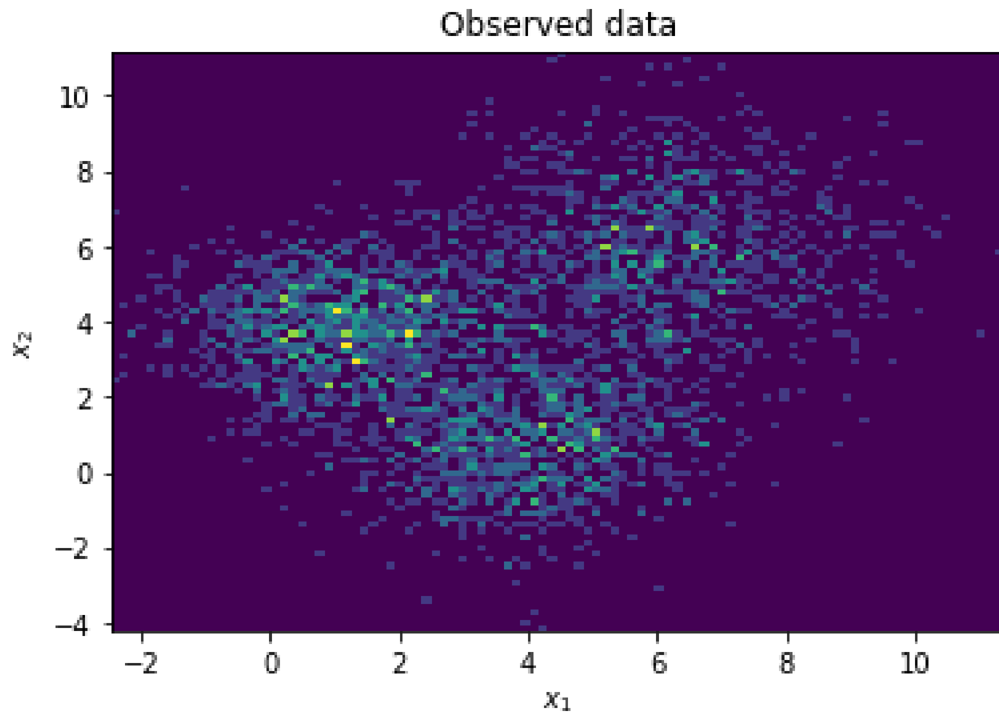
```
In [2]: n_param = 9

        # theta =
        #  0    1    2    3    4    5    6    7    8
        # m11, m12, m21, m22, m31, m32, s1, s2, s3

        real_theta = np.array([1, 4, 4, 1, 6, 6, 1.5, 2, 3])
        dim = 2
        n_1, n_2, n_3 = 1000, 1000, 1000
        n_obs = n_1+n_2+n_3
```

```
In [3]: def sample_from_model(theta, n_samples):
```

```
    ni = int(n_samples/3)

    mu_1 = theta[0:2]
    mu_2 = theta[2:4]
    mu_3 = theta[4:6]
    cov_matrix_1 = np.diag(theta[6]*np.ones(dim))
    cov_matrix_2 = np.diag(theta[7]*np.ones(dim))
    cov_matrix_3 = np.diag(theta[8]*np.ones(dim))

    samples_1 = scipy.stats.multivariate_normal.rvs(mu_1,cov_matrix_1, ni)
    samples_2 = scipy.stats.multivariate_normal.rvs(mu_2,cov_matrix_2, ni)
    samples_3 = scipy.stats.multivariate_normal.rvs(mu_3,cov_matrix_3, ni)
    samples = np.vstack((samples_1, samples_2, samples_3))

    return samples
```

```
In [4]: observations = sample_from_model(real_theta, n_samples = n_obs)
```

```
In [5]: plt.hist2d(observations[:,0], observations[:,1], bins=100)
        plt.xlabel("$x_1$")
        plt.ylabel("$x_2$")
        plt.title("Observed data")
        plt.show()
```



1.1.2 1.2. Proposal distribution

The tranistion model defines how to move from the current θ to a new θ. It is needed to randomly walk in the parameter space.

```
In [6]: def transition_model(curr_theta):
            c = 0.1
            new_theta = np.random.uniform(curr_theta-c, curr_theta+c)

            while np.any(new_theta < 0):
                new_theta = np.random.uniform(curr_theta-c, curr_theta+c)

            return new_theta
```

1.1.3 1.3. Prior distribution

Choose a non-informative prior distribution, such as the uniform distribution.

Remark: some parameters must be strictly positive, remember to ensure such property in your prior.

```
In [7]: def prior(theta):
            if(np.any(theta <=0)):
                return 0
            return 1
```

1.1.4 1.4. Log-likelihood function

By looking at observed data, one can guess that the underlying distribution is a mixture of Gaussians. Let assume we know the model that generated the data but not the parameters.

```
In [8]: def log_lkh(theta, data):
            mu_1 = theta[0:2]
            mu_2 = theta[2:4]
            mu_3 = theta[4:6]
            cov_matrix_1 = np.diag(theta[6]*np.ones(dim))
            cov_matrix_2 = np.diag(theta[7]*np.ones(dim))
            cov_matrix_3 = np.diag(theta[8]*np.ones(dim))
            comp_1 = scipy.stats.multivariate_normal(mu_1,cov_matrix_1).pdf(data)
            comp_2 = scipy.stats.multivariate_normal(mu_2,cov_matrix_2).pdf(data)
            comp_3 = scipy.stats.multivariate_normal(mu_3,cov_matrix_3).pdf(data)
            return np.sum(np.log((comp_1+comp_2+comp_3)/3))
```

1.1.5 1.5. Acceptance rule

Defines whether to accept or reject the new sample. Since we use a log likelihood, we must remember to exponentiate the likelihood before the comparison.

```
In [9]: def acceptance_rule(z, z_new):
            if z_new>z:
```

```
            return True
        else:
            u=np.random.uniform(0,1)
            return (u < (np.exp(z_new-z)))
```

1.1.6 1.6. Metropolis-Hasting algorithm

Inputs: - *likelihood*: returns the likelihood that these parameters generated the data - *prior*: prior knowledge about the parameters that generated the data - *proposal*: a function that draws a sample from a symmetric distribution and returns it - *theta_0*: a starting sample - *traj_len*: number of samples to be generated (i.e. the number of accepted samples - *data*: the data that we wish to model - *acceptance_rule*: decides whether to accept or reject the new sample

```
In [10]: def metropolis_hastings(lkh_fnc, prior, transition_model, theta_0, traj_len, data, acc

             curr_theta = theta_0
             samples = []
             n_acc = 0
             n_rej = 0

             for i in range(traj_len):
                 new_theta =  transition_model(curr_theta)

                 curr_lkh = lkh_fnc(curr_theta, data)
                 new_lkh = lkh_fnc(new_theta, data)
                 if (acceptance_rule(curr_lkh + np.log(prior(curr_theta)), new_lkh+np.log(prior
                     curr_theta = new_theta
                     samples.append(new_theta)
                     n_acc += 1
                 else:
                     samples.append(curr_theta)
                     n_rej += 1
                 if (n_acc-1)%50 == 0:
                     print("step ", n_acc, " / ", traj_len)

             print("Chain generated!")
             return np.array(samples), n_rej
```

1.2 2. Evaluate results

```
In [11]: n_param = 9
         theta_0 = np.random.uniform(0,1, (n_param,))

         traj_len = 5000

         samples, n_rej = metropolis_hastings(log_lkh, prior, transition_model, theta_0, traj_l
```

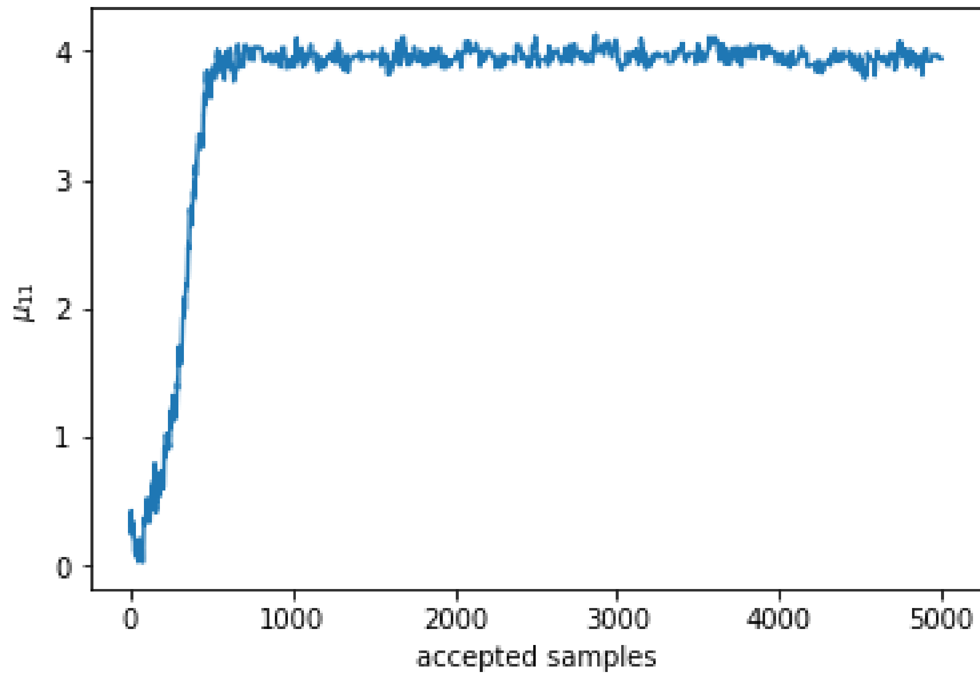
```
step  1  /  5000
step  1  /  5000
```

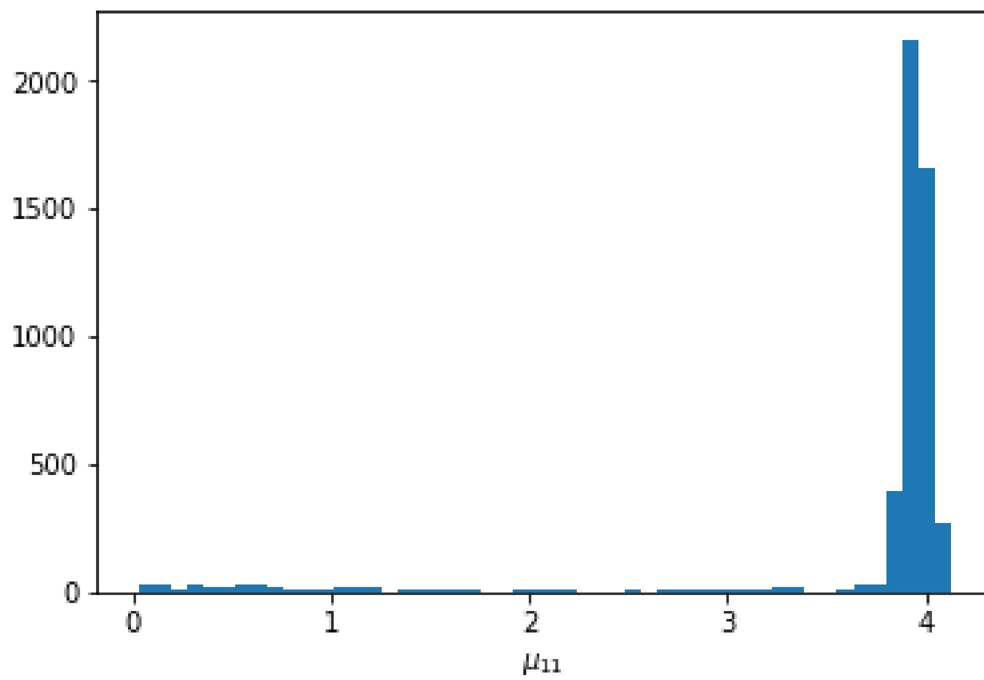
```
step 1 / 5000
step 1 / 5000
step 1 / 5000
step 51 / 5000
step 51 / 5000
step 51 / 5000
step 101 / 5000
step 101 / 5000
step 151 / 5000
step 201 / 5000
step 201 / 5000
step 251 / 5000
step 301 / 5000
step 301 / 5000
step 351 / 5000
step 351 / 5000
step 351 / 5000
step 351 / 5000
step 351 / 5000
step 401 / 5000
step 401 / 5000
step 401 / 5000
step 401 / 5000
step 401 / 5000
step 401 / 5000
step 401 / 5000
step 401 / 5000
step 401 / 5000
step 451 / 5000
step 451 / 5000
step 451 / 5000
step 451 / 5000
step 451 / 5000
step 501 / 5000
step 501 / 5000
step 551 / 5000
step 551 / 5000
step 551 / 5000
step 551 / 5000
step 551 / 5000
step 551 / 5000
step 601 / 5000
step 601 / 5000
step 651 / 5000
step 651 / 5000
step 651 / 5000
step 651 / 5000
step 651 / 5000
```

```
step 701 / 5000
step 701 / 5000
step 701 / 5000
step 701 / 5000
step 701 / 5000
step 701 / 5000
step 701 / 5000
step 701 / 5000
step 701 / 5000
step 701 / 5000
step 701 / 5000
step 701 / 5000
step 701 / 5000
step 751 / 5000
step 751 / 5000
step 801 / 5000
step 851 / 5000
step 851 / 5000
step 851 / 5000
step 851 / 5000
step 851 / 5000
step 851 / 5000
step 851 / 5000
step 851 / 5000
step 851 / 5000
step 851 / 5000
step 851 / 5000
step 851 / 5000
step 901 / 5000
step 901 / 5000
step 901 / 5000
step 901 / 5000
step 951 / 5000
Chain generated!
```

Plot the trace and the empirical distribution for each parameter (remember to throw away the burn-in samples)

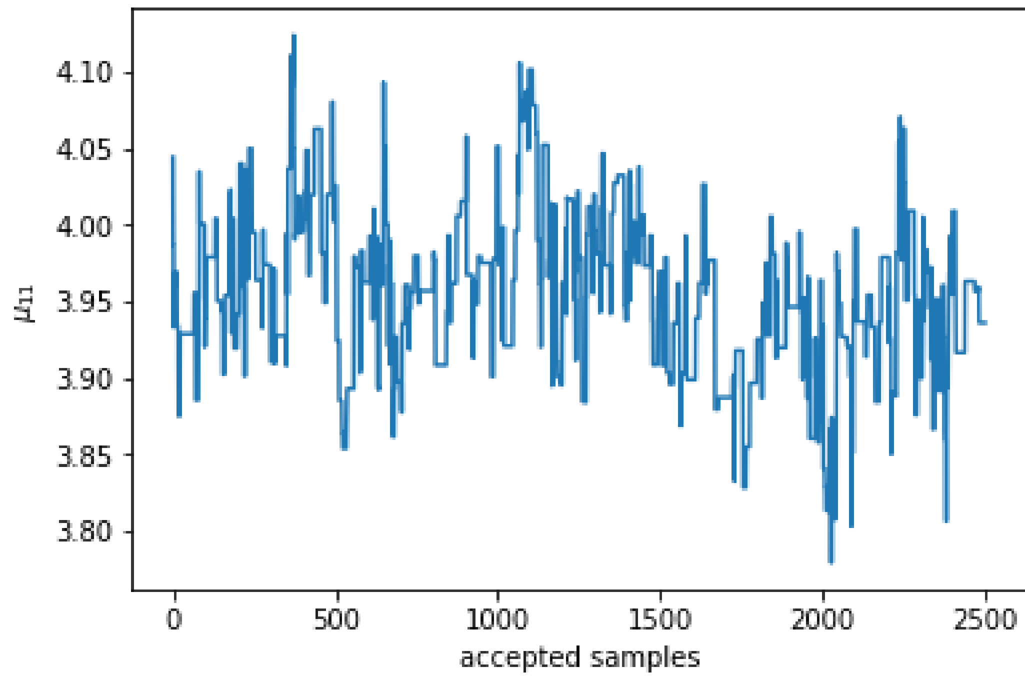
```
In [12]: plt.plot(np.arange(traj_len), samples[:,0])

         plt.xlabel("accepted samples")
         plt.ylabel("$\mu_{11}$")
         plt.show()

         plt.hist(samples[:, 0], bins=50)
         plt.xlabel("$\mu_{11}$")
         plt.show()
```

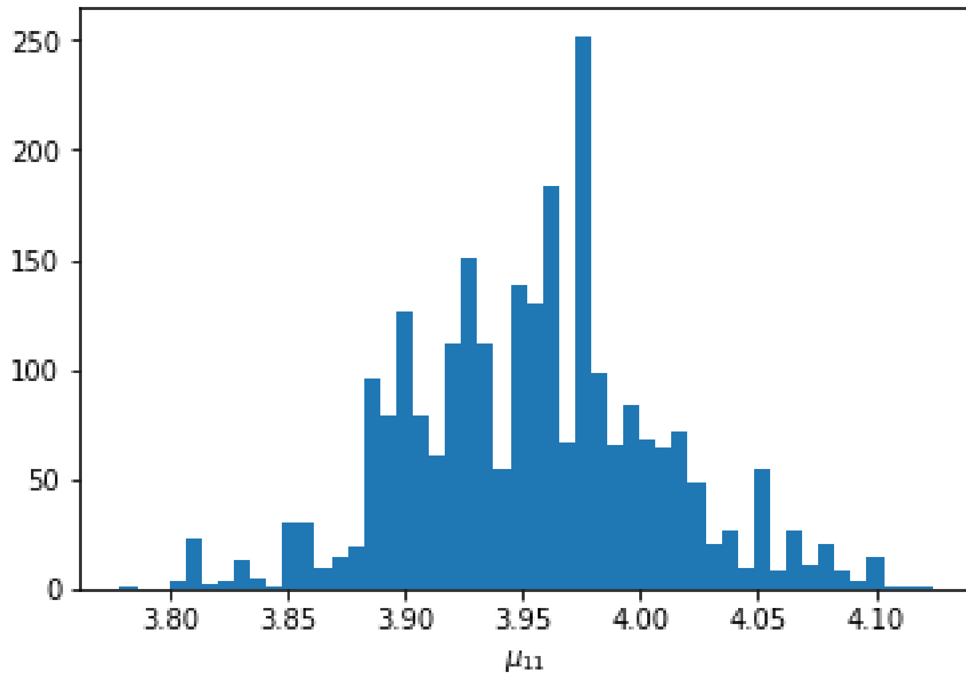





```
In [13]: burn_in = int(traj_len/2)
```

```
plt.plot(np.arange(burn_in), samples[burn_in:,0])

plt.xlabel("accepted samples")
plt.ylabel("$\mu_{11}$")
plt.show()

plt.hist(samples[burn_in:, 0], bins=50)
plt.xlabel("$\mu_{11}$")
plt.show()
```

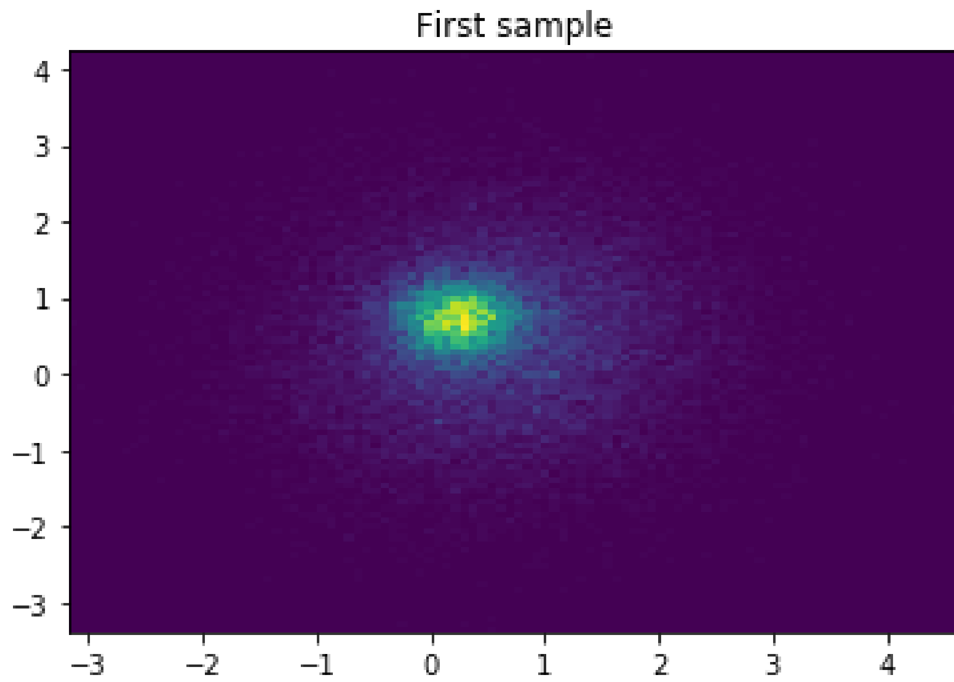




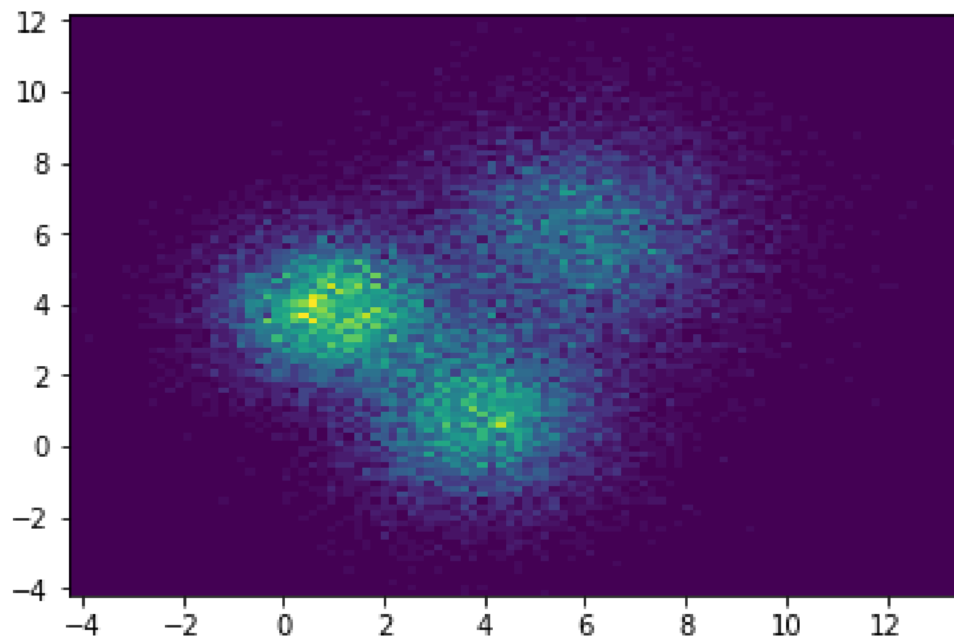
Importance of removing the burn-in samples. The first samples typically do not produce a nice reconstruction of the distribution that generates the data.

```
In [14]: first_sample = sample_from_model(samples[0], 30000)
         last_sample = sample_from_model(samples[-1], 30000)
```

```
In [15]: plt.hist2d(first_sample[:,0], first_sample[:,1], bins=100)
         plt.title("First sample")
         plt.show()
```




```
In [16]: plt.hist2d(last_sample[:,0], last_sample[:,1], bins=100)
         plt.show("Last sample")
```



1.3 3. Diagnostic of convergence

Monitoring of the convergence performances.

1.3.1 3.1. Generate m trajectories of length n:

- Choose m/2 (m even number) initial points, well-dispersed troughout the parameter space.
- From each point generate a trajectory of length 4n
- Burn-in: throw away the first half → m/2 chains of length 2n
- Divide each chain in half → m chains of length n

```
In [17]: m = 20
         m_half = int(m/2)
         n = 1000

         traj_len = 4*n

         full_chains = np.zeros((m_half, traj_len, n_param))
         for i in range(m_half):
             theta_0 = np.random.uniform(0,10, (n_param,))
             samples, _ = metropolis_hastings(log_lkh, prior, transition_model, theta_0, traj_l
             full_chains[i] = samples
```

```
step  1  /  4000
step  51  /  4000
step  51  /  4000
step  101  /  4000
step  151  /  4000
step  151  /  4000
step  201  /  4000
step  251  /  4000
step  301  /  4000
step  351  /  4000
step  401  /  4000
step  401  /  4000
step  451  /  4000
step  451  /  4000
step  501  /  4000
step  551  /  4000
step  551  /  4000
step  551  /  4000
step  551  /  4000
step  551  /  4000
step  551  /  4000
step  551  /  4000
step  551  /  4000
step  551  /  4000
step  551  /  4000
step  551  /  4000
```

```
step  551  /  4000
step  601  /  4000
step  651  /  4000
step  651  /  4000
step  651  /  4000
step  701  /  4000
step  701  /  4000
step  701  /  4000
step  701  /  4000
step  701  /  4000
step  701  /  4000
step  701  /  4000
step  701  /  4000
step  751  /  4000
step  751  /  4000
step  751  /  4000
step  751  /  4000
step  751  /  4000
step  751  /  4000
step  751  /  4000
step  751  /  4000
step  751  /  4000
step  751  /  4000
step  751  /  4000
step  801  /  4000
step  851  /  4000
step  851  /  4000
step  851  /  4000
step  851  /  4000
step  851  /  4000
step  851  /  4000
step  851  /  4000
step  851  /  4000
step  851  /  4000
step  851  /  4000
step  851  /  4000
step  851  /  4000
step  851  /  4000
step  901  /  4000
step  901  /  4000
step  901  /  4000
step  901  /  4000
step  901  /  4000
Chain generated!
step  1  /  4000
step  51  /  4000
step  101  /  4000
step  101  /  4000
```

```
step 151 / 4000
step 201 / 4000
step 251 / 4000
step 251 / 4000
step 251 / 4000
step 251 / 4000
step 251 / 4000
step 251 / 4000
step 251 / 4000
step 251 / 4000
step 251 / 4000
step 251 / 4000
step 301 / 4000
step 301 / 4000
step 351 / 4000
step 351 / 4000
step 401 / 4000
step 401 / 4000
step 401 / 4000
step 401 / 4000
step 401 / 4000
step 401 / 4000
step 401 / 4000
step 451 / 4000
step 451 / 4000
step 451 / 4000
step 451 / 4000
step 451 / 4000
step 451 / 4000
step 451 / 4000
step 451 / 4000
step 451 / 4000
step 451 / 4000
step 451 / 4000
step 451 / 4000
step 451 / 4000
step 451 / 4000
step 451 / 4000
step 451 / 4000
step 451 / 4000
step 451 / 4000
step 451 / 4000
step 451 / 4000
step 501 / 4000
step 501 / 4000
step 501 / 4000
step 501 / 4000
step 501 / 4000
```

```
step 501 / 4000
step 501 / 4000
step 501 / 4000
step 551 / 4000
step 551 / 4000
step 551 / 4000
step 551 / 4000
step 551 / 4000
step 551 / 4000
step 551 / 4000
step 551 / 4000
step 551 / 4000
step 551 / 4000
step 551 / 4000
step 551 / 4000
step 551 / 4000
step 551 / 4000
step 551 / 4000
step 551 / 4000
step 601 / 4000
step 601 / 4000
step 601 / 4000
step 601 / 4000
step 601 / 4000
step 601 / 4000
step 601 / 4000
step 601 / 4000
step 651 / 4000
Chain generated!
step 1 / 4000
step 51 / 4000
step 51 / 4000
step 51 / 4000
step 51 / 4000
step 101 / 4000
step 151 / 4000
step 151 / 4000
step 201 / 4000
step 251 / 4000
step 251 / 4000
step 251 / 4000
step 251 / 4000
step 301 / 4000
step 301 / 4000
step 301 / 4000
step 301 / 4000
step 301 / 4000
step 351 / 4000
```

```
step 351 / 4000
step 351 / 4000
step 351 / 4000
step 401 / 4000
step 451 / 4000
step 451 / 4000
step 451 / 4000
step 451 / 4000
step 451 / 4000
step 451 / 4000
step 451 / 4000
step 451 / 4000
step 451 / 4000
step 451 / 4000
step 451 / 4000
step 451 / 4000
step 451 / 4000
step 451 / 4000
step 501 / 4000
step 501 / 4000
step 501 / 4000
step 551 / 4000
step 551 / 4000
step 551 / 4000
step 551 / 4000
step 551 / 4000
step 551 / 4000
step 551 / 4000
step 551 / 4000
step 551 / 4000
step 601 / 4000
step 601 / 4000
step 651 / 4000
step 651 / 4000
step 651 / 4000
step 651 / 4000
step 651 / 4000
step 701 / 4000
step 751 / 4000
step 751 / 4000
step 751 / 4000
step 751 / 4000
step 751 / 4000
step 751 / 4000
step 751 / 4000
step 751 / 4000
step 751 / 4000
step 751 / 4000
```

```
step  751  /  4000
step  751  /  4000
step  751  /  4000
step  751  /  4000
step  751  /  4000
step  801  /  4000
step  801  /  4000
step  801  /  4000
step  801  /  4000
step  801  /  4000
Chain generated!
step  1  /  4000
step  51  /  4000
step  101  /  4000
step  151  /  4000
step  151  /  4000
step  151  /  4000
step  151  /  4000
step  201  /  4000
step  201  /  4000
step  201  /  4000
step  251  /  4000
step  301  /  4000
step  351  /  4000
step  351  /  4000
step  351  /  4000
step  351  /  4000
step  351  /  4000
step  351  /  4000
step  351  /  4000
step  351  /  4000
step  351  /  4000
step  401  /  4000
step  401  /  4000
step  401  /  4000
step  401  /  4000
step  401  /  4000
step  401  /  4000
step  401  /  4000
step  401  /  4000
step  401  /  4000
step  401  /  4000
step  401  /  4000
step  401  /  4000
step  401  /  4000
step  401  /  4000
step  401  /  4000
step  401  /  4000
```

```
step  401  /  4000
step  401  /  4000
step  401  /  4000
step  401  /  4000
step  401  /  4000
step  401  /  4000
step  401  /  4000
step  401  /  4000
step  401  /  4000
step  401  /  4000
step  401  /  4000
step  401  /  4000
step  401  /  4000
step  401  /  4000
step  401  /  4000
step  401  /  4000
step  401  /  4000
step  401  /  4000
step  401  /  4000
step  401  /  4000
step  401  /  4000
step  401  /  4000
step  451  /  4000
step  451  /  4000
step  501  /  4000
step  501  /  4000
step  501  /  4000
step  501  /  4000
step  501  /  4000
step  501  /  4000
step  551  /  4000
step  551  /  4000
step  551  /  4000
step  551  /  4000
step  551  /  4000
step  551  /  4000
step  601  /  4000
step  651  /  4000
step  701  /  4000
step  701  /  4000
step  701  /  4000
step  701  /  4000
step  701  /  4000
step  701  /  4000
step  701  /  4000
step  701  /  4000
step  701  /  4000
Chain generated!
```

```
step  1  /  4000
step  1  /  4000
step  51  /  4000
step  51  /  4000
step  101  /  4000
step  101  /  4000
step  101  /  4000
step  101  /  4000
step  101  /  4000
step  101  /  4000
step  101  /  4000
step  151  /  4000
step  151  /  4000
step  151  /  4000
step  151  /  4000
step  201  /  4000
step  201  /  4000
step  201  /  4000
step  201  /  4000
step  201  /  4000
step  251  /  4000
step  301  /  4000
step  301  /  4000
step  351  /  4000
step  401  /  4000
step  401  /  4000
step  401  /  4000
step  401  /  4000
step  401  /  4000
step  401  /  4000
step  401  /  4000
step  451  /  4000
step  451  /  4000
step  451  /  4000
step  451  /  4000
step  451  /  4000
step  451  /  4000
step  501  /  4000
step  501  /  4000
step  551  /  4000
step  551  /  4000
step  551  /  4000
step  551  /  4000
step  551  /  4000
step  601  /  4000
step  601  /  4000
step  651  /  4000
step  651  /  4000
```

```
Chain generated!
step  1  /  4000
step  1  /  4000
step  1  /  4000
step  1  /  4000
step  1  /  4000
step  1  /  4000
step  51  /  4000
step  101  /  4000
step  151  /  4000
step  201  /  4000
step  201  /  4000
step  251  /  4000
step  301  /  4000
step  351  /  4000
step  351  /  4000
step  351  /  4000
step  351  /  4000
step  351  /  4000
step  401  /  4000
step  401  /  4000
step  401  /  4000
step  451  /  4000
step  451  /  4000
step  451  /  4000
step  451  /  4000
step  451  /  4000
step  451  /  4000
step  451  /  4000
step  451  /  4000
step  451  /  4000
step  451  /  4000
step  451  /  4000
step  451  /  4000
step  451  /  4000
step  451  /  4000
step  451  /  4000
step  451  /  4000
step  451  /  4000
step  451  /  4000
step  451  /  4000
step  451  /  4000
step  451  /  4000
step  451  /  4000
step  451  /  4000
step  501  /  4000
step  501  /  4000
step  501  /  4000
```

```
step 501 / 4000
step 501 / 4000
step 501 / 4000
step 501 / 4000
step 501 / 4000
step 501 / 4000
step 501 / 4000
step 501 / 4000
step 501 / 4000
step 501 / 4000
step 501 / 4000
step 501 / 4000
step 501 / 4000
step 501 / 4000
step 501 / 4000
step 501 / 4000
step 501 / 4000
step 501 / 4000
step 501 / 4000
step 501 / 4000
step 501 / 4000
step 501 / 4000
step 501 / 4000
step 501 / 4000
step 501 / 4000
step 501 / 4000
step 501 / 4000
step 501 / 4000
step 501 / 4000
step 501 / 4000
step 501 / 4000
step 501 / 4000
step 501 / 4000
step 501 / 4000
step 501 / 4000
step 501 / 4000
step 551 / 4000
step 551 / 4000
step 551 / 4000
step 551 / 4000
step 551 / 4000
step 551 / 4000
step 551 / 4000
step 551 / 4000
step 551 / 4000
step 551 / 4000
step 601 / 4000
step 601 / 4000
```

```
step  601  /  4000
step  601  /  4000
step  601  /  4000
step  651  /  4000
step  651  /  4000
step  651  /  4000
step  651  /  4000
step  701  /  4000
step  751  /  4000
step  751  /  4000
step  801  /  4000
step  801  /  4000
step  801  /  4000
step  801  /  4000
Chain generated!
step  1  /  4000
step  1  /  4000
step  51  /  4000
step  101  /  4000
step  151  /  4000
step  151  /  4000
step  201  /  4000
step  201  /  4000
step  201  /  4000
step  251  /  4000
step  301  /  4000
step  301  /  4000
step  301  /  4000
step  301  /  4000
step  351  /  4000
step  351  /  4000
step  401  /  4000
step  451  /  4000
step  451  /  4000
step  451  /  4000
step  451  /  4000
step  451  /  4000
step  451  /  4000
step  451  /  4000
step  451  /  4000
step  451  /  4000
step  451  /  4000
step  501  /  4000
step  501  /  4000
step  501  /  4000
step  501  /  4000
step  551  /  4000
step  551  /  4000
```

```
step  551  /  4000
step  601  /  4000
step  601  /  4000
step  601  /  4000
step  601  /  4000
step  601  /  4000
step  601  /  4000
step  601  /  4000
step  601  /  4000
step  601  /  4000
step  601  /  4000
step  601  /  4000
step  601  /  4000
step  601  /  4000
step  601  /  4000
step  601  /  4000
step  601  /  4000
step  601  /  4000
step  601  /  4000
step  601  /  4000
step  601  /  4000
step  601  /  4000
step  651  /  4000
step  651  /  4000
step  651  /  4000
step  651  /  4000
Chain generated!
step  1  /  4000
step  1  /  4000
step  1  /  4000
step  51  /  4000
step  51  /  4000
step  101  /  4000
step  151  /  4000
step  201  /  4000
step  201  /  4000
step  251  /  4000
step  301  /  4000
step  351  /  4000
step  351  /  4000
step  351  /  4000
step  351  /  4000
step  401  /  4000
step  401  /  4000
step  401  /  4000
step  451  /  4000
step  451  /  4000
step  451  /  4000
```

```
step  451  /  4000
step  451  /  4000
step  451  /  4000
step  451  /  4000
step  501  /  4000
step  501  /  4000
step  501  /  4000
step  501  /  4000
step  501  /  4000
step  501  /  4000
step  501  /  4000
step  501  /  4000
step  501  /  4000
step  551  /  4000
step  601  /  4000
step  601  /  4000
step  601  /  4000
step  601  /  4000
step  601  /  4000
step  601  /  4000
step  601  /  4000
step  601  /  4000
step  651  /  4000
step  701  /  4000
step  701  /  4000
step  701  /  4000
step  751  /  4000
step  751  /  4000
step  801  /  4000
step  801  /  4000
step  801  /  4000
Chain generated!
step  1  /  4000
step  1  /  4000
step  1  /  4000
step  51  /  4000
step  101  /  4000
step  101  /  4000
step  101  /  4000
step  101  /  4000
step  101  /  4000
step  101  /  4000
step  101  /  4000
step  101  /  4000
step  101  /  4000
step  151  /  4000
step  201  /  4000
step  201  /  4000
```

```
step 251 / 4000
step 251 / 4000
step 251 / 4000
step 301 / 4000
step 301 / 4000
step 351 / 4000
step 401 / 4000
step 401 / 4000
step 451 / 4000
step 451 / 4000
step 451 / 4000
step 451 / 4000
step 451 / 4000
step 451 / 4000
step 451 / 4000
step 451 / 4000
step 501 / 4000
step 501 / 4000
step 501 / 4000
step 501 / 4000
step 501 / 4000
step 501 / 4000
step 501 / 4000
step 501 / 4000
step 501 / 4000
step 501 / 4000
step 551 / 4000
step 551 / 4000
step 551 / 4000
step 551 / 4000
step 551 / 4000
step 551 / 4000
step 551 / 4000
step 551 / 4000
step 551 / 4000
step 551 / 4000
step 551 / 4000
step 551 / 4000
step 551 / 4000
step 551 / 4000
step 551 / 4000
step 551 / 4000
step 551 / 4000
step 551 / 4000
step 551 / 4000
step 601 / 4000
step 651 / 4000
step 651 / 4000
```

```
step  651  /  4000
step  651  /  4000
step  651  /  4000
step  651  /  4000
step  651  /  4000
step  651  /  4000
step  651  /  4000
step  701  /  4000
step  701  /  4000
step  701  /  4000
Chain generated!
step  1  /  4000
step  51  /  4000
step  51  /  4000
step  101  /  4000
step  151  /  4000
step  151  /  4000
step  151  /  4000
step  151  /  4000
step  151  /  4000
step  151  /  4000
step  151  /  4000
step  201  /  4000
step  201  /  4000
step  201  /  4000
step  201  /  4000
step  201  /  4000
step  201  /  4000
step  251  /  4000
step  251  /  4000
step  251  /  4000
step  251  /  4000
step  301  /  4000
step  301  /  4000
step  301  /  4000
step  301  /  4000
step  351  /  4000
step  351  /  4000
step  351  /  4000
step  351  /  4000
step  351  /  4000
step  351  /  4000
step  351  /  4000
step  351  /  4000
step  351  /  4000
step  351  /  4000
step  401  /  4000
step  401  /  4000
```

```
step  401  /  4000
step  401  /  4000
step  451  /  4000
step  451  /  4000
step  451  /  4000
step  451  /  4000
step  501  /  4000
step  501  /  4000
step  551  /  4000
step  551  /  4000
step  551  /  4000
step  551  /  4000
step  551  /  4000
step  551  /  4000
step  601  /  4000
step  601  /  4000
step  651  /  4000
step  651  /  4000
step  651  /  4000
step  701  /  4000
step  701  /  4000
step  701  /  4000
step  701  /  4000
step  701  /  4000
step  701  /  4000
step  701  /  4000
step  701  /  4000
step  701  /  4000
step  701  /  4000
step  701  /  4000
step  701  /  4000
step  701  /  4000
step  701  /  4000
step  701  /  4000
Chain generated!
```

In [18]:
```
#full_chains = np.ones((m_half, traj_len, n_param))
print(full_chains.shape)
no_burnin_chains = full_chains[:,2*n:,:]
print(no_burnin_chains.shape)
```

```
(10, 4000, 9)
(10, 2000, 9)
```

In [19]:
```
chains = np.vstack((no_burnin_chains[:,:n,:], no_burnin_chains[:,n:,:]))
print(chains.shape)
```

```
(20, 1000, 9)
```

```
In [ ]:
```

1.3.2 3.2. Choose scalar function ψ

Hint: the log-likelihood defined before may be a good choice.

```
In [20]: psi = lambda par: log_lkh(par, observations)
```

1.3.3 3.3. Compute the between-sequence and the within-sequence variance

- **Between**:
$$B = \frac{n}{m-1}\sum_{j=1}^{m}(\bar{\psi}_j - \bar{\psi})^2,$$
where $\bar{\psi}_j = \frac{1}{n}\sum_{i=1}^{n}\psi_{ij}$ and $\bar{\psi} = \frac{1}{m}\sum_{i=1}^{m}\bar{\psi}_j$.
- **Within**:
$$W = \frac{1}{m}\sum_{j=1}^{m}s_j^2,$$
where $s_j^2 = \frac{1}{n-1}\sum_{i=1}^{n}(\psi_{ij} - \bar{\psi}_j)^2$.

Notation: $\psi_{ij} := \psi(\theta_{ij})$.

```
In [21]: def between_variance(chains):
             # chains is a m x n matrix
             m, n, _ = chains.shape
             bar_psi_j = np.empty(m)
             for j in range(m):
                 bar_psi_j[j] = np.mean(np.array([psi(th_i) for th_i in chains[j]]))
             bar_psi = np.mean(bar_psi_j)

             B = np.sum((bar_psi_j-bar_psi)**2)*n/(m-1)

             return B
```

```
In [22]: def within_variance(chains):
             # chains is a m x n matrix
             m, n, _ = chains.shape

             bar_psi_j = np.empty(m)
             s_j = np.empty(m)
             for j in range(m):
                 psi_ij = np.array([psi(th_i) for th_i in chains[j]])
                 bar_psi_j[j] = np.mean(psi_ij)
                 s_j[j] = np.sum((psi_ij-bar_psi_j[j])**2)/(n-1)
```

```
        W = np.mean(s_j)

        return W
```

```
In [23]: def var_plus_psi(chains):
             m,n, _ = chains.shape
             B = between_variance(chains)
             W = within_variance(chains)
             return (W*(n-1)+B)/n
```

```
In [24]: def var_plus_psi_fast(chains):
             m,n, _ = chains.shape
             psi_ij = np.array([[psi(th_i) for th_i in chains[j]] for j in range(m)])
             bar_psi_j = np.mean(psi_ij, axis=1)

             bar_psi = np.mean(bar_psi_j)

             B = np.sum((bar_psi_j-bar_psi)**2)*n/(m-1)

             s_j = np.sum((psi_ij-bar_psi_j.reshape(m,1))**2, axis=1)/(n-1)

             W = np.mean(s_j)

             return (W*(n-1)+B)/n, W, B
```

```
In [25]: import time
         start = time.time()
         var_p, _, _ = var_plus_psi_fast(chains)
         print(time.time()-start)
```

```
28.67379856109619
```

```
In [35]: psi_ij = np.array([[psi(th_i) for th_i in chains[j]] for j in range(m)])
```

1.3.4 3.4. First convergence criterion

If $\hat{R} \leq 1.1$ we have converged

```
In [36]: def R_hat(chains):
             W = within_variance(chains)
             var_p = var_plus_psi(chains)

             return np.sqrt(var_p/W)
```

```
In [37]: def R_hat_fast(chains):
             var_p, W, _ = var_plus_psi_fast(chains)

             return np.sqrt(var_p/W)
```

```
In [38]: print(R_hat_fast(chains))
```

```
1.0257354419531537
```

1.3.5 3.5. Second convergence criterion

We need at least $n_{eff} \approx 100$.

```
In [39]: def V_k(chains, k):
             m, n, _ = chains.shape
             Vk_j = np.zeros(m)
             for j in range(m):
                 psi_ij = np.array([psi(th_i) for th_i in chains[j]])
                 p1 = psi_ij[k:]
                 p2 = psi_ij[:n-k]

                 Vk_j[j] = np.sum((p1-p2)**2)

             Vk = np.sum(Vk_j)/(m*(n-k))

             return Vk
```

```
In [40]: # variogram ar lag k
         def V_k_fast(psi_ij, k):
             m, n = psi_ij.shape
             p1 = psi_ij[:, k:]
             p2 = psi_ij[:,:n-k]

             Vk = np.sum((p1-p2)**2)/(m*(n-k))
             return Vk
```

```
In [41]: def rho_hat_k(chains, var_p, k):
             Vk = V_k(chains, k)
             return 1-Vk/(2*var_p )
```

```
In [42]: def rho_hat_k_fast(psi_ij, var_p, k):
             Vk = V_k_fast(psi_ij, k)
             return 1-Vk/(2*var_p )
```

Optional: Find the value K such that

$$K = \min\{k \mid k \text{ is odd and } \hat{\rho}_{k+1} + \hat{\rho}_{k+2} < 0\}.$$

Otherwise, set $K = n$.
Once K is fixed compute the sequence $\hat{\rho}_1 \cdots \hat{\rho}_K$.

```
In [44]: # Find K


         def find_K(psi_ij, var_p):
```

```
        rho = lambda t: rho_hat_k_fast(psi_ij, var_p, t)

        K = 0
        while rho(K+1)+rho(K+2) >= 0 and K<n:
            if K%100 == 0:
                print(K)
            K+=1
        return K

    K = find_K(psi_ij, var_p)
    print("K = ", K)
    # Compute rho_hat_seq of length K
    rho = lambda t: rho_hat_k_fast(psi_ij, var_p, t)
    rho_hat_seq = np.array([rho(k) for k in range(K)])
```

```
0
100
K =  184
```

In [45]:
```
def n_eff_hat(m,n, rho_hat_seq):
    return m*n/(1+2*np.sum(rho_hat_seq))
```

In [46]:
```
print(n_eff_hat(m,n, rho_hat_seq))
```

```
330.3377266822796
```

In [ ]:

In [ ]: