Advanced Machine Learning Assignment: 2

Stian Grønlund, Femke van Venrooij and Monika Kazlauskaitė

October 2024

1 Introduction

Markov Chain Monte Carlo (MCMC) is a well-known method for sampling from complex probability distributions. It includes algorithms like Metropolis Hastings and Hamiltonian Monte Carlo, for several types of analyses, but can fall short when variables to be sampled from are strongly correlated, or are high-dimensional.
This report analyzes the implementation of Metropolis Hastings (MH) and Hamiltonian Monte Carlo (HMC), applied to problems involving a highly correlated bivariate Gaussian, and posterior inference for perceptron weights. The objective is to see the effect of hyper-parameter tweaking on both the acceptance rate and parameter estimates of the approximations made by these algorithms. The code was implemented in Python, making use of JAX (Bradbury et al., 2018) for numerical integration and gradient calculation for HMC.

2 Design

2.1 Metropolis Hastings

The Metropolis Hastings (MH) algorithm is used to sample from an arbitrary probability distribution, when sampling from the actual distribution is impossible or computationally intractable. MH uses a proposal density function to sample a new proposed state. Then, it decides whether to accept a "move" to this state depending on the ratio (MacKay, 2003):

$$a = \frac{P^*(x_0)P^*(x(t))}{Q(x(t); x_0)Q(x_0; x(t))}$$

and moves to the new state with probability $min(1, a)$. When gathering samples using Metropolis-Hastings, the acceptance ratio is important: too low wastes samples, while too high limits exploration to the highest density region of the distribution. Both lead to not actually exploring the distribution of interest.
The pseudo code of the MH algorithm is given in 1.

2.2 Hamiltonian Monte Carlo

Hamiltonian Monte Carlo (HMC) uses derivatives of the target density to propose transitions for probability distributions. Inspired by dynamical systems, it applies numerical integration to treat the distribution as a Hamiltonian system, using momentum and position parameters to simulate evolution. HMC views the distribution variables as position parameters and generates position vectors from a (hyper-)parameterized multivariate normal distribution. ((Team, 2021), (Bishop, 2006))
The pseudocode for HMC is given in 2.

Algorithm 1 Metropolis Hastings MCMC

```
for $r \leftarrow 1$ to $N$ do
    $x' \leftarrow$ sample from $q(x'|x^r)$
    compute $a$
    if $a \geq 1$ then
        $x^{r+1} \leftarrow x'$
    else
        draw $u \sim \text{Uniform}(0,1)$
        if $u \leq a$ then
            $x^{r+1} \leftarrow x'$
        else
            $x^{r+1} \leftarrow x^r$
        end if
    end if
end for
```

Algorithm 2 Hamiltonian Monte Carlo

```
Choose initial $x_1$
for $t \leftarrow 1$ to $T$ do
    $p_t \leftarrow$ sample from $\mathcal{N}(0, \alpha^{-1})$
    $(x_0, p_0) \leftarrow$ Run Hamiltonian dynamics from $(x_t, p_t)$
    $a \leftarrow \frac{e^{-H(x_0,p_0)}}{e^{-H(x_t,p_t)}}$
    draw $u \sim \text{Uniform}(0,1)$
    if $u \leq \min(1, a)$ then
        $(x_{t+1}, p_{t+1}) \leftarrow (x_0, p_0)$
    else
        $(x_{t+1}, p_{t+1}) \leftarrow (x_t, p_t)$
    end if
end for
```

3 Implementation

3.1 Metropolis Hastings algorithm

The implementation for Metropolis Hastings looks fairly straightforward to follow from the pseudocode. One thing to take note of is how the acceptance/rejection decision is made. We sample from a uniform distribution, $p \sim [0, 1]$, and accept if $a > p$:

```
    acceptance_prob = min(1, acceptance_ratio)
    u = np.random.uniform(0, 1, size=1)
    if u <= acceptance_prob:
        return x_proposed, 1
    else:
        return x_initial, 0
```

Additionally, the following lines illustrates how we keep track of the eventual acceptance ratio as a running sum:

```
    x_sample, num_proposed = sample_x(x_proposed,
                                      x_init,
                                      proportional_function,
                                      args)
    ...
    sum_proposed += num_proposed
    ...
all_acceptance_ratio = sum_proposed/num_iterations
```

3.2 Hamiltonian Monte Carlo

The implementation for HMC is a bit more complex, as it leans in to optimization techniques using the JAX python library. First, let's take a look at the algorithm's Hamiltonian dynamics computed using the leapfrog integrator:

```
def run_leapfrog(rho, g, x, eps, energy_fn:Callable, tau):
  def scan_step(carry, _):
    rho, g, x = carry

    rho = rho - 0.5 * eps * g
    x = x + eps*rho
    g = jax.grad(energy_fn)(x)
    rho = rho - 0.5 * eps * g

    return (rho, g, x), None
  return jax.lax.scan(scan_step, (rho, g, x), jnp.arange(tau))[0]
```

A leapfrog integrator will first compute a half-step update for the momentum, then a full-step update for the position, then a second half-step update for the momentum. Here, rho is the variable used to encode the momentum values, and x the position values.
The step of the leapfrog integrator is included within the scope of another function to make use of JAX's optimization, as function objects cannot be passed to JAX jit (just-in-time) compiled object. Other implementation details like the accept decision and the acceptance ratio are implemented similar as in Metropolis-Hastings.

4 Testing and Results

We will test these algorithms on two problems. The first we call "elongated Gaussian", and the second "Bayesian Inference for Perceptron Learning" (or Inference for short).

4.1 Elongated Gaussian

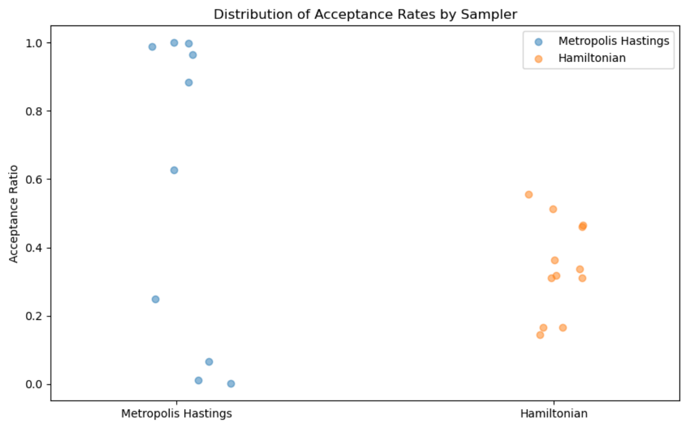
In the elongated Gaussian problem, we are given a bivariate Gaussian with mean zero, and high correlation between the variables. The Gaussian is described by the energy:

$$p(x_1, x_2) \propto \exp\left(-\frac{1}{2}x^\top A x\right)$$

with

$$A = \begin{pmatrix} 250.25 & -249.75 \\ -249.75 & 250.25 \end{pmatrix}$$

For the Metropolis-Hastings algorithm, the parameter that can be tweaked is the variance of the proposal distribution, σ. We varied sigma over the interval $[10^{-4}, 10]$, in equal increments, with 40,000 samples. The optimal value was $\sigma = 0.2154$, resulting in an acceptance ratio of 0.25. This is similar to the optimal acceptance rate suggested by Sherlock and Roberts (2009). Note the data points showing an acceptance rate of close to 1: these are cases where MH does not adequately explore the full distribution. Example plots are shown in the Appendix as 12.



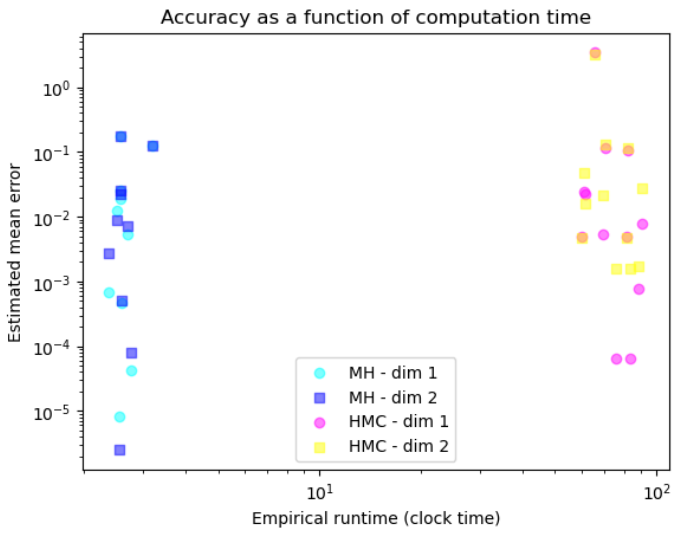


Figure 1: Left: Acceptance rates for both algorithms. Right: Log accuracy plotted against log runtimes.

For the Hamiltonian Monte Carlo algorithm, the parameters that can be tweaked are ε and τ: the size and number of steps in the leapfrog integration, respectively. We tested all combinations of the following arrays of values. For ε: $[0.1, 0.01, 0.001, 0.0001]$, and for τ: $[5, 10, 25, 50, 100, 250]$, with 1000 samples. It was quickly determined that a ε of 0.1 resulted in a non-functional acceptance rate, and it was therefore excluded from the results shown. The parameters that seemed to best explore the distribution was $\varepsilon = 0.001$ and $\tau = 50$, with an acceptance ratio of 0.465.
We present two more diagnostic plots of the results, in figure 1. The left plot shows the distribution of acceptance ratios for both algorithms. It can be seen that several of the parameters were dysfunctional. The right plot shows accuracy as a function of the computation time of the algorithms. Two things should be noted. Across the board, HMC used magnitudes more computation time than Metropolis Hastings. Therefore, we used a logarithmic scale for our x-axis on the plot, which shows the clock runtime of the algorithm. Additionally, some of our estimated mean square errors are also several orders of magnitude different from each other, and we therefore also used a logarithmic scale for our y-axis on this plot.

4.2 Bayesian Inference for Perceptron Learning

In the Inference problem, we sample from the posterior of a learning problem. In particular, we have three weights for a perceptron, and want to find the posterior value of the weights given some data;

$$P(\boldsymbol{w} \mid D, \alpha) = \frac{P(D \mid \boldsymbol{w}) P(\boldsymbol{w} \mid \alpha)}{P(D \mid \alpha)} \tag{1}$$

Also given are the equations that describe the distributions for the likelihood in this Bayes' rule equation: $P(D \mid \boldsymbol{w}) = \exp(-G(\boldsymbol{w}))$, where

$$G(\boldsymbol{w}) = -\sum_n \left[t^{(n)} \ln y(x^{(n)}; \boldsymbol{w}) + (1 - t^{(n)}) \ln(1 - y(x^{(n)}; \boldsymbol{w})) \right]$$

and for the prior: $P(\boldsymbol{w} \mid \alpha) = \frac{1}{Z_W(\alpha)} \exp(-\alpha E_W)$, with

$$E_W(\boldsymbol{w}) = \frac{1}{2} \sum_i w_i^2$$

being a regularizing prior, and $1/Z_W(\alpha) = (\alpha/2\pi)^{K/2}$, with $K =$ the number of parameters in the vector $\boldsymbol{w}$ (MacKay, 2003). Given this, we find our "energy" function for our sampling procedure;

$$M(\boldsymbol{w}) = G(\boldsymbol{w}) + \alpha E_W(\boldsymbol{w})$$

We implemented each of these equations as functions within Python, and then provided our sampling algorithms with the function M:

```
def y_fn(x, w):
    wx = x @ w.T
    return 1 / (1 + jnp.exp(-wx))

def G_calc(x, w, labels):
    y = y_fn(x,w)
    G = -labels * jnp.log(y) + (1 - labels) * jnp.log(1 - y)
    return jnp.sum(G)

def E_calc(w):
    return 1 / 2 * jnp.sum(w ** 2)

def M(w, x, labels, a):
    g = G_calc(x, w, labels)
    e = E_calc(w)
    m = g + a*e
    k = len(w)
    z = jnp.power((a/2*jnp.pi), k/2)
    return -m*z
```

We also kept track of the values of $M(w)$ and $G(w)$ over time, so that we could plot the ”energy” of the system and the log likelihood (”loss”) of the sampler over time.
For this exercise, we also used the plots of both $M(w)$ and the values of the sampled variables over time to estimate the burn-in period of the algorithm. The burn-in period is the period before the algorithm reaches ”stationarity” - namely before it actually finds likely values of the variables.

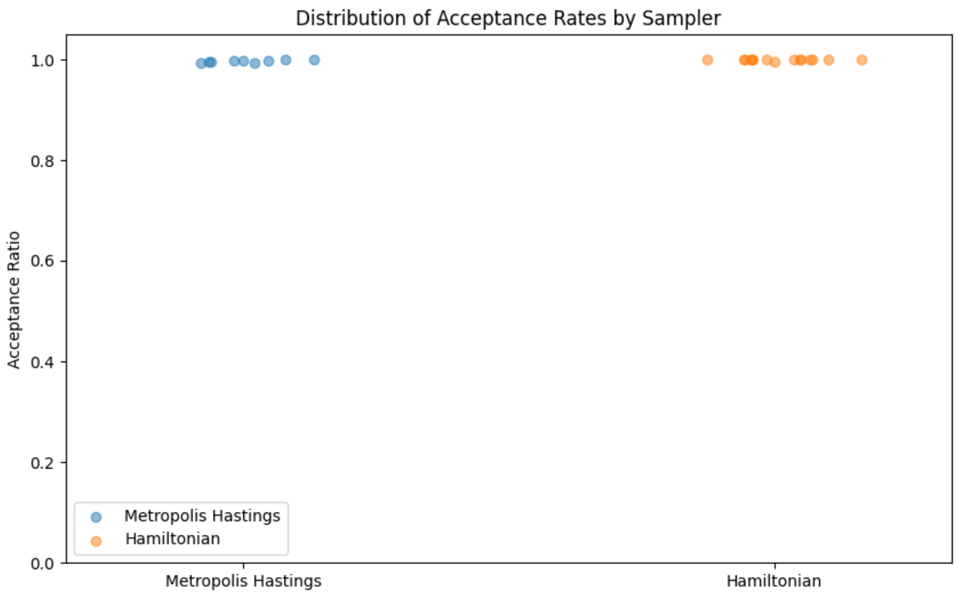


Figure 2: Acceptance rates for both algorithms. Note the highly strange pattern of acceptance rates.

For the Metropolis-Hastings algorithm, we varied sigma over the interval $[10^{-4}, 1]$, in equal increments, with 10,000 samples. The optimal value was $\sigma = 1$, resulting in an acceptance ratio of 0.9991. This is indicative that the σ value was too small. However, when we increased it further, we ran into severe numerical overflow issues.
For Hamiltonian Monte Carlo we tested all combinations of the following arrays of values. For ε: $[0.01, 0.001, 0.0005]$, and for τ: $[5, 10, 25, 50, 100]$. The optimal values were $\varepsilon = 0.001$ and $\tau = 100$, which resulted in an acceptance ratio of 1.
Both of the algorithms seem to draw out the general same ”shape” of distribution of w_1 and w_2, even if that distribution is mirrored compared to each other (see figure 5 and figure 6).
We have here also a wider array of plots, in 11, recreating plots related to perceptron learning from MacKay (2003), figure 41.5.
We can use these plots to estimate the burn-in period of the algorithms. First, for Metropolis-Hastings, we can see that it appears to reach stationarity after a similar number of samples, although it’s a bit hard to read given our numerical overflow issues. But we can see that the graph of $M(w)$ stays around ~ -70 from iteration ~ 100 until the end. As for HMC, it’s not that customary to use burn-in given the Markov Chain’s quick movement towards the real distribution, due to the use of Hamiltonian Dynamics. However, we can see from the plots that after only about 100 or so steps, the $M(w)$ graph flattens out, indicating that the sampling process has reached stationarity.

5 Conclusion

Overall, Metropolis Hastings seems to have come out on top in our testing. It is a lot faster than HMC, without seeming to sacrifice a lot on accuracy.

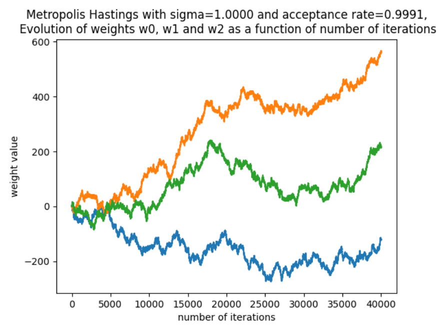


Figure 3: Weights over iterations MH

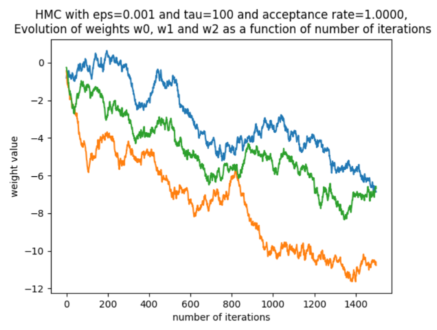


Figure 4: Weights over iterations, HMC

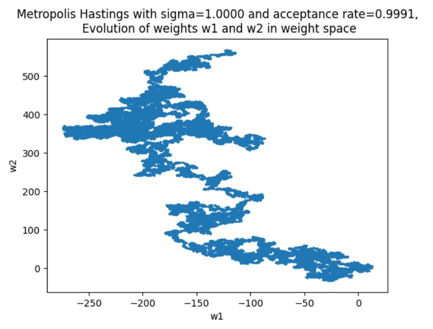


Figure 5: Distribution of w_1, w_2 MH

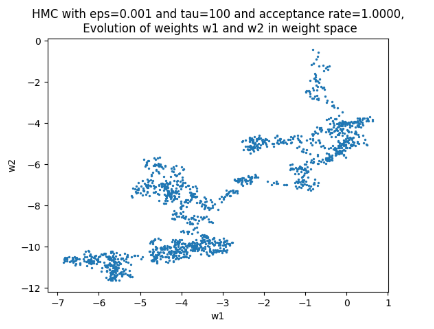


Figure 6: Distribution of w_1, w_2 HMC

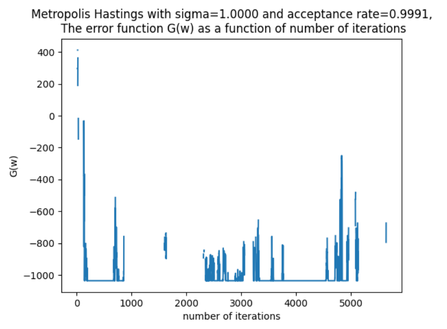


Figure 7: G(w) over iterations, MH

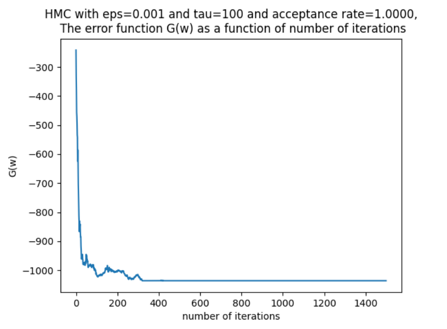


Figure 8: G(w) over iterations, HMC

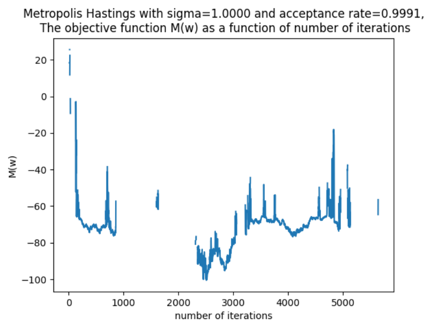


Figure 9: M(w) over iterations, MH

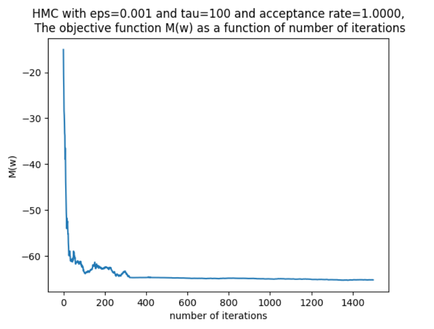


Figure 10: M(w) over iterations, HMC

Figure 11: Plots recreating (41.5) from MacKay (2003)

These results are actually so stark that I'm starting to believe we have some wild mistakes in our HMC algorithm, which is supposed to be competitive with Metropolis Hastings, and even preferred for problems with a larger state-space or more complicated energy functions. This is further documented by our strange (for HMC) acceptance rate values.
However, our results - especially MH for the Perceptron Inference task - are so strange that we're really quite not sure what to make of it. Overall, the one we're left with is that either hyperparameter tuning in MCMC algorithms are even more important than we previously thought, or we did something wrong somewhere. Perhaps a bit of both.
Some caveats must be taken with our approach. First, using clock-time as a run-time measure is looked down upon, as it is sensitive to details of the system being used, among other things. A better approach would define some comparable "fundamental" operation(s) being done in both algorithms, and compare run-time complexity as a function of those (which are in turn a function of input variables).

6 Appendix

The code for the plots, algorithms in this report, and more not included, can be found at https://github.com/meraxion/cds-ml/tree/main/aml_as-02.
Running the elongated_gaussian.py and bayes_inf_perceptron_learning .py files, respectively, will produce the corresponding plots, while the implementations of the algorithms can be found in metropolis_hastings .py and hamiltonian_mc.py.

6.1 Extra Plots

Figure 12 shows two plots related to the elongated Gaussian exercise. On the left is the samples gathered from Metropolis Hastings for $\sigma = 0.11$, and on the right the same for $\sigma = 0.0001$. As can be seen in the latter case, because the proposal distribution has too small variance, we don't ever move far away from the starting condition - and don't explore the whole distribution.

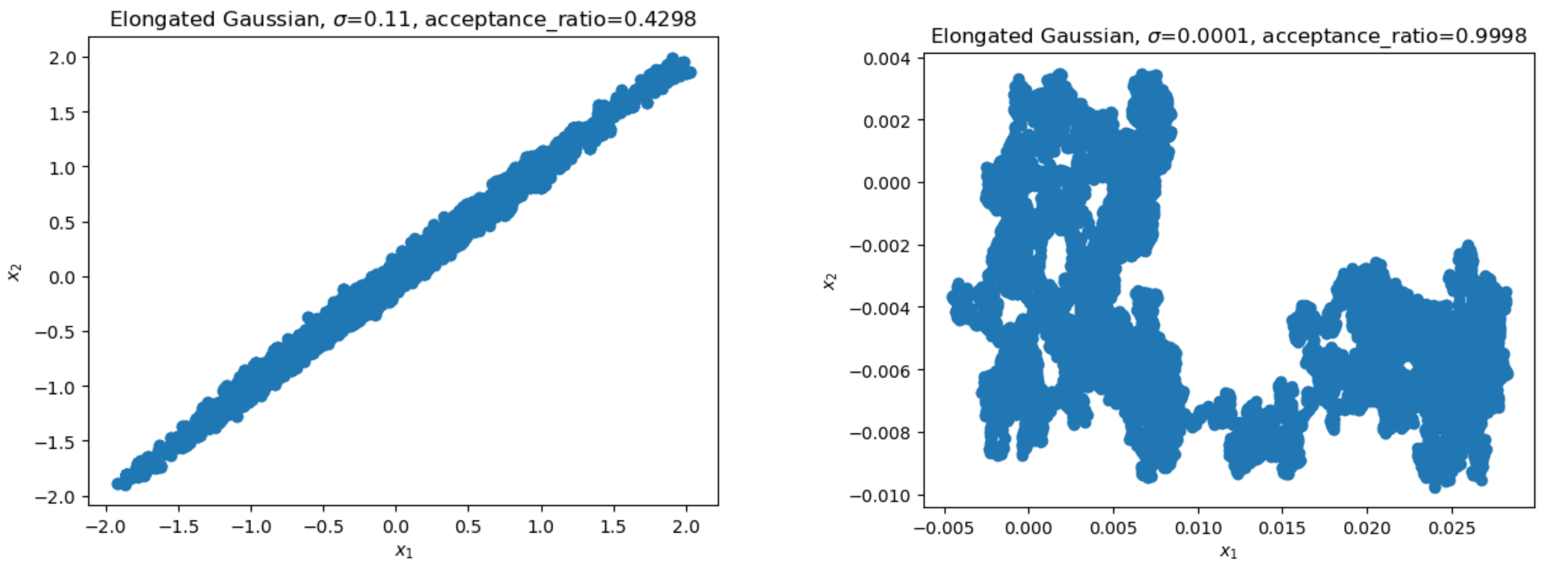


Figure 12: Left: Distribution of samples gathered with $\sigma = 0.11$.
Right: Distribution of samples gathered with $\sigma = 0.0001$, both with Metropolis Hastings

References

Bishop, C. M. (2006). *Pattern recognition and machine learning*. Springer.

Bradbury, J., Frostig, R., Hawkins, P., Johnson, M. J., Leary, C., Maclaurin, D., Necula, G., Paszke, A., VanderPlas, J., Wanderman-Milne, S., & Zhang, Q. (2018). *JAX: Composable transformations of Python+NumPy programs* (Version 0.3.13). http://github.com/jax-ml/jax

MacKay, D. J. C. (2003). *Information theory, inference, and learning algorithms*. Cambridge University Press.

Sherlock, C., & Roberts, G. (2009). Optimal scaling of the random walk Metropolis on elliptically symmetric unimodal targets. *Bernoulli*, *15*(3). https://doi.org/10.3150/08-BEJ176

Team, S. D. (2021). *Stan Modeling Language Users Guide and Reference Manual* (Version 2.35). https://mc-stan.org