1 Metropolis Algorithm

Lab Objective: *Understand the basic principles of the Metropolis algorithm and apply these ideas to the Ising Model.*

The Metropolis Algorithm

Sampling from a given probability distribution is an important task in many different applications found throughout the sciences. When these distributions are complicated, as is often the case when modeling real-world problems, direct sampling methods can become difficult, as they might involve computing high-dimensional integrals. The Metropolis algorithm is an effective method to sample from many distributions, requiring only that we be able to evaluate the probability density function up to a constant of proportionality. In particular, the Metropolis algorithm does not require us to compute difficult high-dimensional integrals, such as those that are found in the denominator of Bayesian posterior distributions.

The Metropolis algorithm is an MCMC sampling method which generates a sequence of random variables, similar to Gibbs sampling. These random variables form a Markov Chain whose invariant distribution is equal to the distribution from which we wish to sample. Suppose that $h : \mathbb{R}^n \to \mathbb{R}$ is the probability density function of distribution, and suppose that $f(\boldsymbol{\theta}) = c \cdot h(\boldsymbol{\theta})$ for some nonzero constant c (in practice, we assume that f is an easy function to evaluate, while h is difficult). Let $Q : \mathbb{R}^n \times \mathbb{R}^n \to \mathbb{R}$ be a symmetric *proposal function* (so that $Q(\cdot, \mathbf{y})$ is a probability density function for all $\mathbf{y} \in \mathbb{R}^n$, and $Q(\mathbf{x}, \mathbf{y}) = Q(\mathbf{y}, \mathbf{x})$ for all $\mathbf{x}, \mathbf{y} \in \mathbb{R}^n$) and let $A : \mathbb{R}^n \times \mathbb{R}^n \to \mathbb{R}$ be an *acceptance function* defined by

$$A(\mathbf{x}, \mathbf{y}) = \min\left(1, \frac{f(\mathbf{x})}{f(\mathbf{y})}\right).$$

We can combine these functions in such a way so as to sample from the aforementioned Markov Chain by following Algorithm 1.1. The Metropolis algorithm can be interpreted as follows: given our current state $\mathbf{y}$, we propose a new state according to the distribution $Q(\cdot, \mathbf{y})$. We then accept or reject it according to A. We continue by repeating the process. So long as Q defines an irreducible, aperiodic, and non-null recurrent Markov chain, we will have a Markov chain whose unique invariant distribution will have density h. Furthermore, given any initial state, the chain will converge to this invariant distribution. Note that for numerical reasons, it is often wise to make calculations of the acceptance functions in log space:

$$\log A(\mathbf{x}, \mathbf{y}) = \min(0, \log f(\mathbf{x}) - \log f(\mathbf{y})).$$

Algorithm 1.1 Metropolis Algorithm

procedure METROPOLIS ALGORITHM
  Choose initial point $\mathbf{x}_0$.
  for $t = 1, 2, \ldots$ **do**
    Draw $\mathbf{x}' \sim Q(\cdot, \mathbf{x}_{t-1})$
    Draw $a \sim \text{unif}(0, 1)$
    if $a \leq A(\mathbf{x}', \mathbf{x}_{t-1})$ **then**
      $\mathbf{x}_t = \mathbf{x}'$
    else
      $\mathbf{x}_t = \mathbf{x}_{t-1}$
  Return $\mathbf{x}_1, \mathbf{x}_2, \mathbf{x}_3, \ldots$

Let's apply the Metropolis algorithm to a simple example of Bayesian analysis. Consider the problem of computing the posterior distribution over the mean μ and variance σ^2 of a normal distribution for which we have N data points $y_1, \ldots, y_N$. For concreteness, we use the data in `examscores.csv` and we assume the prior distributions

$$\mu \sim N(\mu_0 = 80, \sigma_0^2 = 16)$$
$$\sigma^2 \sim IG(\alpha = 3, \beta = 50).$$

In this situation, we wish to sample from the posterior distribution

$$p(\mu, \sigma^2 \,|\, y_1, \ldots, y_N) = \frac{p(\mu)p(\sigma^2)\prod_{i=1}^{N} N(y_i \,|\, \mu, \sigma^2)}{\int_{-\infty}^{\infty}\int_{0}^{\infty} p(\mu)p(\sigma^2)\prod_{i=1}^{N} N(y_i \,|\, \mu, \sigma^2)\, d\sigma^2 d\mu}.$$

However, we can conveniently calculate only the numerator of this expression. Since the denominator is simply a constant with respect to μ and σ^2, the numerator can serve as the function f in the Metropolis algorithm, and the denominator can serve as the constant c.

We choose our proposal function to be based on a bivariate Normal distribution:

$$Q(x, y) = N(x \,|\, y, sI),$$

where I is the 2×2 identity matrix and s is some positive scalar.

```
>>> import numpy as np
>>> from scipy import stats
>>> from math import sqrt, exp, log
>>> from matplotlib import pyplot as plt

# Load in the data and initialize hyperparameters.
>>> scores = np.load("examscores.npy")
>>> alpha = 3
>>> beta = 50
>>> mu0 = 80
>>> sig20 = 16

# Initialize the prior distributions.
>>> muprior = stats.norm(loc=mu0, scale=sqrt(sig20))
>>> sig2prior = stats.invgamma(alpha, scale=beta)
```

```
>>> def proposal(y, s):
...     """The proposal function Q(x,y) = N(x|y,sI)."""
...     return stats.multivariate_normal.rvs(mean=y, cov=s*np.eye(len(y)))
...
>>> def propLogDensity(x):
...     """Calculate the log of the proportional density."""
...     logprob = muprior.logpdf(x[0]) + sig2prior.logpdf(x[1])
...     logprob += stats.norm.logpdf(scores, loc=x[0], scale=sqrt(x[1])).sum()
...     return logprob    # ^this is where the scores are used.
...
>>> def acceptance(x, y):
...     return min(0, propLogDensity(x) - propLogDensity(y))
...
```

We are now ready to code up the Metropolis algorithm using these functions. We will keep track of the samples generated by the algorithm, along with the proportional log densities of the samples and the proportion of proposed samples that were accepted.

```
def metropolis(x0, s, n_samples):
    """Use the Metropolis algorithm to sample from posterior.

    Parameters:
        x0 ((2,) ndarray): The first entry is mu, the second entry is sigma^2.
        s (float): The standard deviation parameter for the proposal function.
        n_samples (int): The number of samples to generate.

    Returns:
        draws ((n_samples, 2) ndarray): The MCMC samples.
        logprobs ((n_samples,) ndarray): The log density of the samples.
        accept_rate (float): The proportion of accepted proposed samples.
    """
    accept_counter = 0
    draws = np.empty((n_samples,2))
    logprob = np.empty(n_samples)
    x = x0.copy()
    for i in range(n_samples):
        xprime = proposal(x,s)
        a = np.random.uniform()
        if log(a) <= acceptance(xprime, x):
            accept_counter += 1
            x = xprime
        draws[i] = x
        logprob[i] = propLogDensity(x)
    return draws, logprob, accept_counter/n_samples
```

Now let's sample from the posterior. We take initial guesses $\mu = 40$ and $\sigma^2 = 10$ and set $s = 20$.

```
# Draw 10,000 samples.
```

```
>>> draws, lprobs, rate = metropolis(np.array([40., 10.]), 20., 10000)
>>> print("Acceptance Rate:", rate)
Acceptance Rate: 0.3531
```

We can evaluate the quality of our results by plotting the log probabilities, the μ samples, the σ^2 samples, and kernel density estimators for the marginal posterior distributions of μ and σ^2.

```
# Plot the first 500 log probabilities.
>>> plt.plot(lprobs[:500])
>>> plt.show()

>>> fig, axes = plt.subplots(2, 2)
# Plot the mu samples.
>>> axes[0,0].plot(draws[:,0])
>>> axes[0,0].set_title(r"$\mu$ samples")

# Plot the sigma2 samples.
>>> axes[1,0].plot(draws[:,1])
>>> axes[1,0].set_title(r"$\sigma^2$ samples")

# Build and plot KDE for posterior mu.
>>> mu_kernel = stats.gaussian_kde(draws[50:,0])
>>> x_min = draws[50:,0].min() - 1
>>> x_max = draws[50:,0].max() + 1
>>> x = np.linspace(x_min, x_max, 200)
>>> axes[0,1].plot(x, mu_kernel(x))
>>> axes[0,1].set_title(r"$\mu$ posterior")

# Build and plot KDE for posterior sigma2.
>>> sig_kernel = stats.gaussian_kde(draws[50:,1])
>>> x_min, x_max = 20, 200
>>> x = np.linspace(x_min, x_max, 200)
>>> axes[1,1].plot(x, sig_kernel(x))
>>> axes[1,1].set_title(r"$\sigma^2$ posterior")
>>> plt.show()
```

The results should be similar to Figures 1.2 and 1.1.

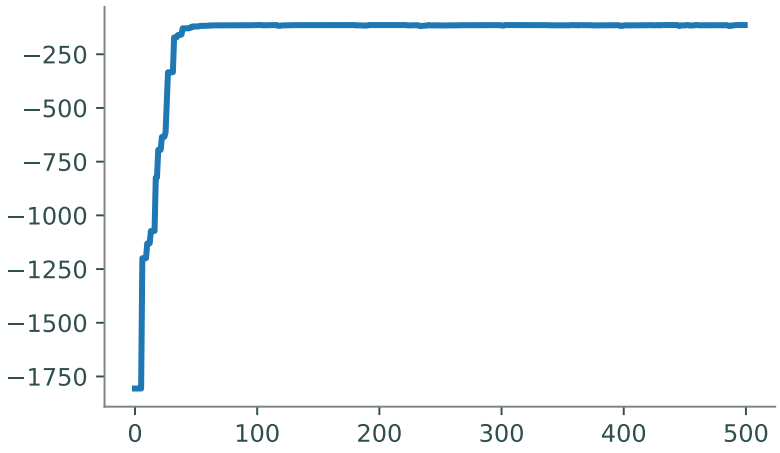


Figure 1.1: Log densities of the first 500 Metropolis samples.

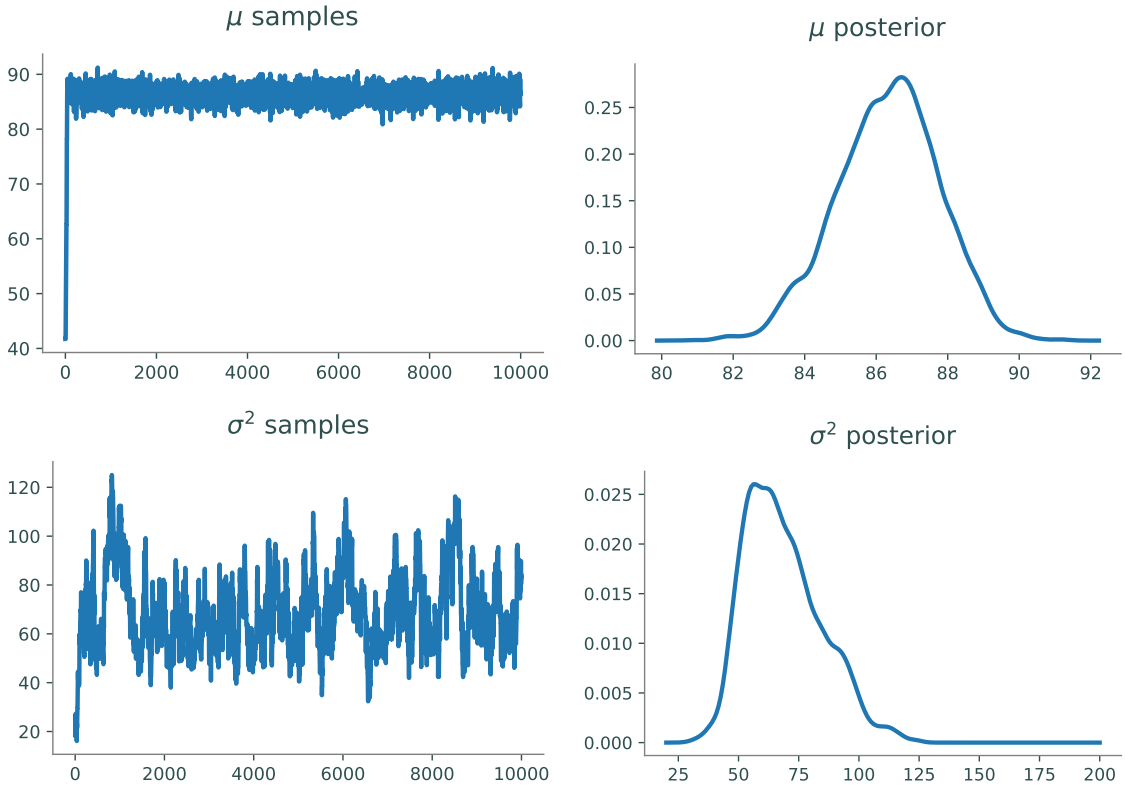


Figure 1.2: Metropolis samples and KDEs for the marginal posterior distribution of μ (top row) and σ^2 (bottom row).

The Ising Model

In statistical mechanics, the Ising model describes how atoms interact in ferromagnetic material. Assume we have some lattice Λ of sites. We say $i \sim j$ if i and j are adjacent sites. Each site i in our lattice is assigned an associated *spin* $\sigma_i \in \{\pm 1\}$. A *state* in our Ising model is a particular spin configuration $\sigma = (\sigma_k)_{k \in \Lambda}$. If $L = |\Lambda|$, then there are 2^L possible states in our model. If L is large, the state space becomes huge, which is why MCMC sampling methods (in particular the Metropolis algorithm) are so useful in calculating model estimations.

With any spin configuration σ, there is an associated energy

$$H(\sigma) = -J \sum_{i \sim j} \sigma_i \sigma_j$$

where $J > 0$ for ferromagnetic materials, and $J < 0$ for antiferromagnetic materials. Throughout this lab, we will assume $J = 1$, leaving the energy equation to be $H(\sigma) = -\sum_{i \sim j} \sigma_i \sigma_j$ where the interaction from each pair is added only once.

We will consider a lattice that is a 100×100 square grid. The adjacent sites for a given site are those directly above, below, to the left, and to the right of the site, so to speak. For sites on the edge of the grid, we assume it wraps around. In other words, a site at the farthest left side of the grid is adjacent to the corresponding site on the farthest right side. Thus, a single spin configuration can be represented as a 100×100 array, with entries of ± 1.

Problem 1. Write a function that accepts an integer n and returns a random spin configuration for an $n \times n$ lattice (as an $n \times n$ NumPy array of 1s and -1s).
(Hint: `np.random.binomial()` or `scipy.stats.bernoulli()` may be helpful.)

Test your function with $n = 100$, plotting the spin configuration via `plt.imshow()` with `cmap="gray"`. It should look fairly random, as in Figure 1.3.

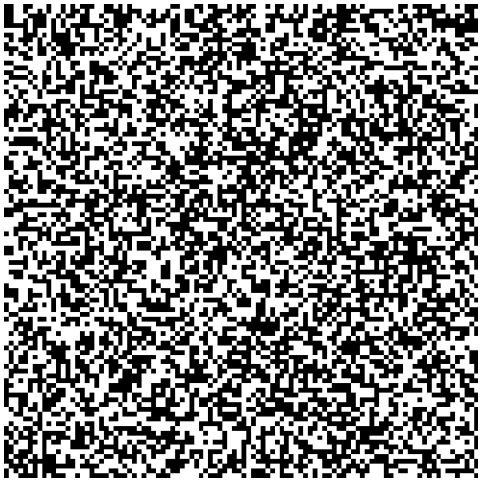


Figure 1.3: Spin configuration from random initialization.

Problem 2. Write a function that accepts a spin configuration σ for a lattice as a NumPy array. Compute the energy $H(\sigma)$ of the spin configuration. Be careful to not double count site pair interactions!
(Hint: `np.roll()` may be helpful.)

Different spin configurations occur with different probabilities, depending on the energy of the spin configuration and $\beta > 0$, a quantity inversely proportional to the temperature. More specifically, for a given β, we have

$$\mathbb{P}_\beta(\sigma) = \frac{e^{-\beta H(\sigma)}}{Z_\beta}$$

where $Z_\beta = \sum_\sigma e^{-\beta H(\sigma)}$. Because there are $2^{100\cdot 100} = 2^{10000}$ possible spin configurations for our particular lattice, computing this sum is infeasible. However, the numerator is quite simple, provided we can efficiently compute the energy $H(\sigma)$ of a spin configuration. Thus the ratio of the probability densities of two spin configurations is simple:

$$\frac{\mathbb{P}_\beta(\sigma^*)}{\mathbb{P}_\beta(\sigma)} = \frac{e^{-\beta H(\sigma^*)}}{e^{-\beta H(\sigma)}} = e^{\beta(H(\sigma)-H(\sigma^*))}$$

The simplicity of this ratio should lead us to think that a Metropolis algorithm might be an appropriate way by which to sample from the spin configuration probability distribution, in which case the acceptance probability would be

$$A(\sigma^*, \sigma) = \begin{cases} 1 & \text{if } H(\sigma^*) < H(\sigma) \\ e^{\beta(H(\sigma)-H(\sigma^*))} & \text{otherwise.} \end{cases} \tag{1.1}$$

By choosing our transition matrix Q cleverly, we can also make it easy to compute the energy for any proposed spin configuration. We restrict our possible proposals to only those spin configurations in which we have flipped the spin at exactly one lattice site, i.e. we choose a lattice site i and flip its spin. Thus, there are only L possible proposal spin configurations σ^* given σ, each being proposed with probability $\frac{1}{L}$, and such that $\sigma^*_j = \sigma_j$ for all $j \neq i$, and $\sigma^*_i = -\sigma_i$. Note that we would never actually write out this matrix (it would be $2^{10000} \times 2^{10000}$). Computing the proposed site's energy is simple: if the spin flip site is i, then we have

$$H(\sigma^*) = H(\sigma) + 2\sum_{j:j\sim i} \sigma_i \sigma_j. \tag{1.2}$$

Problem 3. Write a function that accepts an integer n and chooses a pair of indices (i, j) where $0 \leq i, j \leq n-1$. Each possible pair should have an equal probability $\frac{1}{n^2}$ of being chosen.

Problem 4. Write a function that accepts a spin configuration σ, its energy $H(\sigma)$, and integer indices i and j. Use (1.2) to compute the energy of the new spin configuration σ^*, which is σ but with the spin flipped at the (i, j)th entry of the corresponding lattice. Do not explicitly construct the new lattice for σ^*.

Problem 5. Write a function that accepts a float β and spin configuration energies $H(\sigma)$ and $H(\sigma^*)$. Using (1.1), calculate whether or not the new spin configuration σ^* should be accepted (return `True` or `False`). Consider doing the calculations in log space.

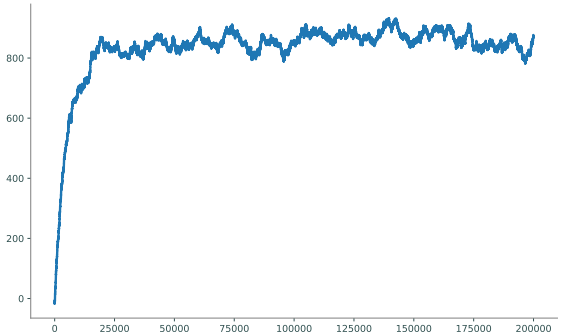
To track the convergence of the Markov chain, we would like to look at the probabilities of each sample at each time. However, this would require us to compute the denominator Z_β, which is generally the reason we have to use a Metropolis algorithm to begin with. We can get away with examining only $-\beta H(\sigma)$. We should see this value increase as the algorithm proceeds, and it should converge once we are sampling from the correct distribution. Note that we don't expect these values to converge to a specific value, but rather to a restricted range of values.

Problem 6. Write a function that accepts a float $\beta > 0$ and integers n, `n_samples`, and `burn_in`. Initialize an $n \times n$ lattice for a spin configuration σ using Problem 1. Use the Metropolis algorithm to (potentially) update the lattice `burn_in` times.

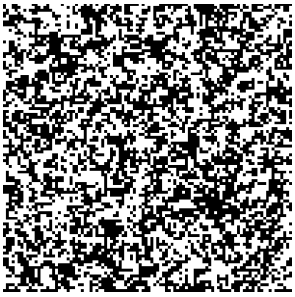
1. Use Problem 3 to choose a site for possibly flipping the spin, thus defining a potential new configuration σ^*.
2. Use Problem 4 to calculate the energy $H(\sigma^*)$ of the proposed configuration.
3. Use Problem 5 to accept or reject the proposed configuration. If it is accepted, set $\sigma = \sigma^*$ by flipping the spin at the indicated site.
4. Track $-\beta H(\sigma)$ at each iteration (independent of acceptance).

After the burn-in period, continue the iteration `n_samples` times, also recording every 100th sample (to prevent memory failure). Return the samples, the sequence of weighted energies $-\beta H(\sigma)$, and the acceptance rate.

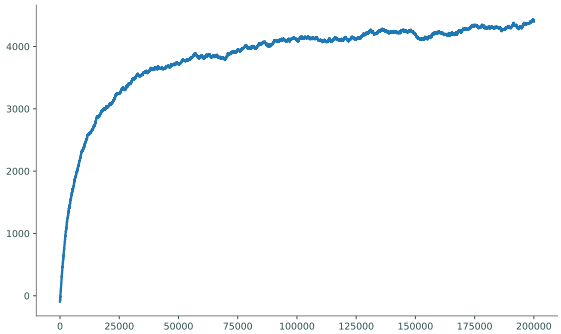
Test your sampler on a 100×100 grid with 200000 total iterations, with `n_samples` large enough so that you will keep 50 samples, for $\beta = 0.2, 0.4, 1$. Plot the proportional log probabilities, as well as a late sample from each test. How does the ferromagnetic material behave differently with differing temperatures? Recall that β is an inverse function of temperature. You should see more structure with lower temperature, as illustrated in Figure 1.4.



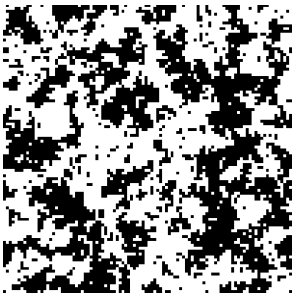
(a) Proportional log probs when $\beta = 0.2$.



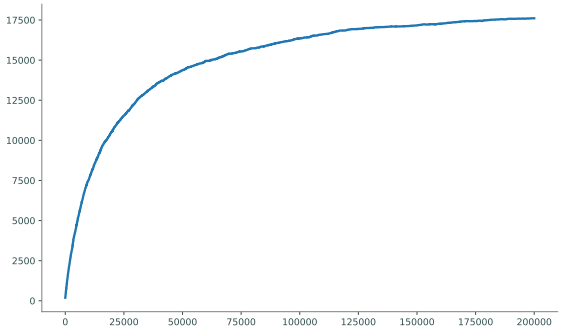
(b) Spin configuration sample when $\beta = 0.2$.



(c) Proportional log probs when $\beta = 0.4$.



(d) Spin configuration sample when $\beta = 0.4$.



(e) Proportional log probs when $\beta = 1$.

(f) Spin configuration sample when $\beta = 1$.

Figure 1.4