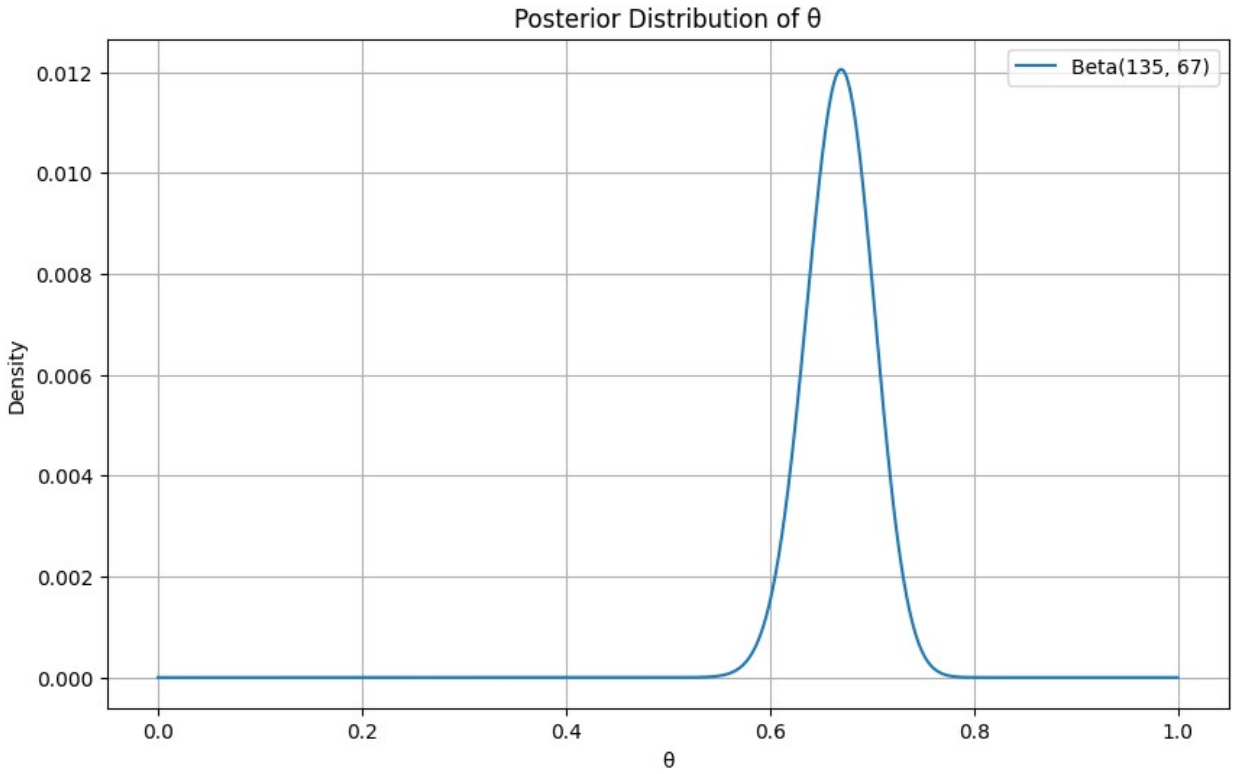
Assignment 3

Roshan Patil (220760)

Part 1: ***Estimating the posterior distribution using different computational methods***

1)

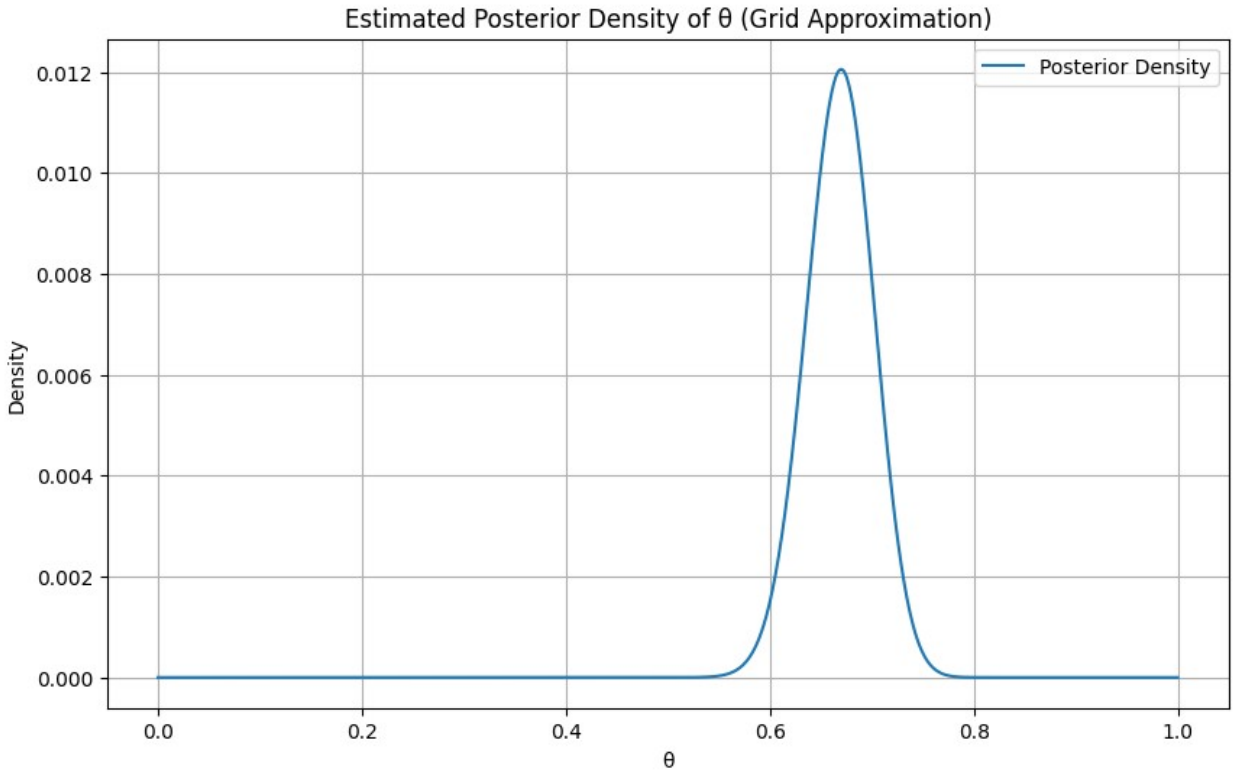
```
import numpy as np
import matplotlib.pyplot as plt

alpha = 135
beta = 67

theta_values = np.linspace(0, 1, 1000)
pdf_values = np.power(theta_values, alpha - 1) * np.power(1 -
theta_values, beta - 1)
pdf_values /= np.sum(pdf_values)  # normalize the PDF values to sum to
1

plt.figure(figsize=(10, 6))
plt.plot(theta_values, pdf_values, label=f'Beta({alpha}, {beta})')
plt.title('Posterior Distribution of θ')
plt.xlabel('θ')
plt.ylabel('Density')
plt.legend()
plt.grid(True)
plt.show()
```



2)

```python
import numpy as np
import matplotlib.pyplot as plt

theta_values = np.linspace(0, 1, 1000)  # grid of 1000 points between 0 and 1

data = np.array([10, 15, 15, 14, 14, 14, 13, 11, 12, 16])
n = 20
log_likelihood = data.sum() * np.log(theta_values) + (n * len(data) - data.sum()) * np.log(1 - theta_values)
likelihood = np.exp(log_likelihood - np.max(log_likelihood))  # subtracting the maximum to avoid overflow
prior = np.ones_like(theta_values)  # uniform prior Beta(1, 1)
unnormalized_posterior = likelihood * prior
posterior = unnormalized_posterior / np.sum(unnormalized_posterior)

plt.figure(figsize=(10, 6))
plt.plot(theta_values, posterior, label='Posterior Density')
plt.title('Estimated Posterior Density of θ (Grid Approximation)')
plt.xlabel('θ')
plt.ylabel('Density')
plt.legend()
plt.grid(True)
plt.show()
```

```
<ipython-input-1-f04f566954e7>:8: RuntimeWarning: divide by zero
encountered in log
  log_likelihood = data.sum() * np.log(theta_values) + (n * len(data)
- data.sum()) * np.log(1 - theta_values)
```



3)

```python
import numpy as np

num_samples = 100000

theta_samples = np.random.beta(1, 1, size=num_samples)

data = np.array([10, 15, 15, 14, 14, 14, 13, 11, 12, 16])
n = 20
likelihoods = np.prod(np.power(theta_samples[:, np.newaxis],
data.sum()) * np.power(1 - theta_samples[:, np.newaxis], n * len(data)
- data.sum()), axis=1)

# Estimate the marginal likelihood using Monte Carlo integration
marginal_likelihood = np.mean(likelihoods)

print(f"Estimated Marginal Likelihood (Monte Carlo):
{marginal_likelihood}")
```

```
Estimated Marginal Likelihood (Monte Carlo): 6.778336010218077e-57
```

4)

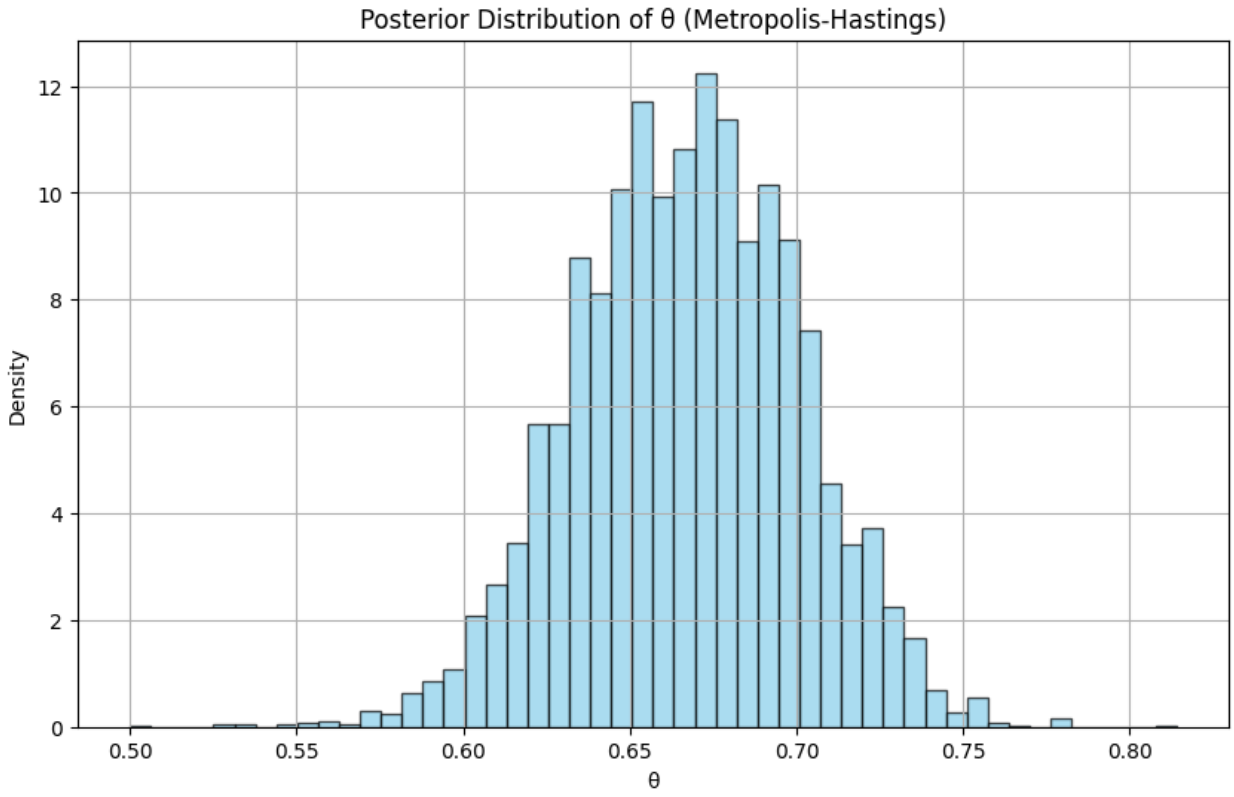
```
import numpy as np
import pandas as pd


N = 10000  # Total number of samples
M = N // 4  # Number of samples to select based on weights

theta_proposal = np.random.beta(2, 2, size=N)

data = np.array([10, 15, 15, 14, 14, 14, 13, 11, 12, 16])
n = 20
likelihoods = np.prod(np.power(theta_proposal[:, np.newaxis],
data.sum()) * np.power(1 - theta_proposal[:, np.newaxis], n *
len(data) - data.sum()), axis=1)

prior = np.ones_like(theta_proposal)
proposal_density = np.random.beta(2, 2, size=N)

weights = likelihoods * prior / proposal_density
weights /= np.sum(weights)

samples_df = pd.DataFrame({'theta': theta_proposal, 'weight':
weights})
selected_samples = samples_df.sample(n=M, weights='weight',
replace=True)['theta'].values

print("Selected samples from the posterior distribution:")
print(selected_samples)
```

```
Selected samples from the posterior distribution:
[0.6639267  0.67287244 0.69952913 ... 0.65640454 0.68801523
0.72044039]
```

5)

```
import numpy as np
import matplotlib.pyplot as plt

data = np.array([10, 15, 15, 14, 14, 14, 13, 11, 12, 16])
n = 20
alpha_prior = 1
beta_prior = 1

def log_posterior(theta, data, n, alpha_prior, beta_prior):
    if theta < 0 or theta > 1:
        return -np.inf  # log(0) for theta outside [0, 1] is -inf
    else:
        likelihood = np.prod(theta**data.sum() * (1-
```

```
theta)**(n*len(data)-data.sum()))
        prior = theta**(alpha_prior-1) * (1-theta)**(beta_prior-1)
        return np.log(likelihood * prior)

# Metropolis-Hastings algorithm
def metropolis_hastings(log_posterior, theta0, n_samples,
proposal_std=0.1):
    samples = [theta0]
    current_theta = theta0

    for _ in range(n_samples):
        proposed_theta = np.random.normal(current_theta, proposal_std)
        log_alpha = log_posterior(proposed_theta, data, n,
alpha_prior, beta_prior) - log_posterior(current_theta, data, n,
alpha_prior, beta_prior)
        # Accept or reject the proposed theta
        if np.log(np.random.uniform(0, 1)) < log_alpha:
            current_theta = proposed_theta
        samples.append(current_theta)
    return np.array(samples)

np.random.seed(42)

theta0 = 0.5  # starting value
n_samples = 10000  # number of samples to draw

samples = metropolis_hastings(log_posterior, theta0, n_samples)

# Plot the posterior distribution of theta
plt.figure(figsize=(10, 6))
plt.hist(samples, bins=50, density=True, color='skyblue',
edgecolor='black', alpha=0.7)
plt.title('Posterior Distribution of θ (Metropolis-Hastings)')
plt.xlabel('θ')
plt.ylabel('Density')
plt.grid(True)
plt.show()
```

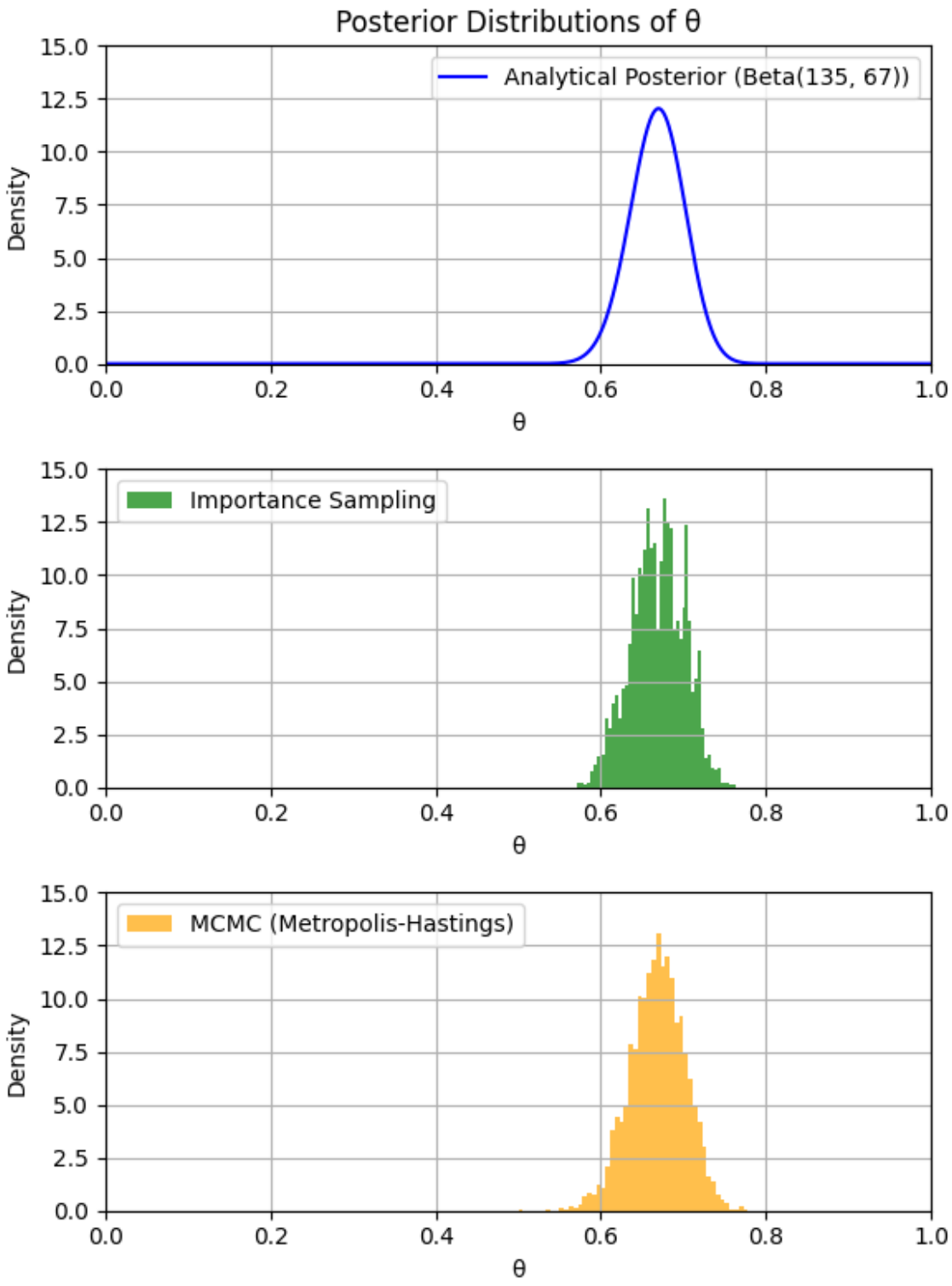


6)

```
import numpy as np
import matplotlib.pyplot as plt
from scipy.stats import beta

data = np.array([10, 15, 15, 14, 14, 14, 13, 11, 12, 16])
n = 20

analytical_theta = np.linspace(0, 1, 1000)
analytical_posterior = beta.pdf(analytical_theta, 135, 67)

# Importance Sampling function
def importance_sampling_posterior(num_samples):
    theta_samples = np.random.beta(2, 2, size=num_samples)

    likelihoods = np.prod(np.power(theta_samples[:, np.newaxis],
data.sum()) * np.power(1 - theta_samples[:, np.newaxis], n * len(data)
- data.sum()), axis=1)
    prior = theta_samples**(1-1) * (1-theta_samples)**(1-1)  # Beta(1,
1) prior
    proposal = np.random.beta(2, 2, size=num_samples)
    unnormalized_posterior = likelihoods * prior / proposal

    # Normalize weights
    weights = unnormalized_posterior / np.sum(unnormalized_posterior)
```

```
    posterior_samples = np.random.choice(theta_samples,
size=num_samples//4, replace=True, p=weights)

    return posterior_samples

# Metropolis-Hastings function
def metropolis_hastings_posterior(theta0, n_samples):
    def log_posterior(theta, data, n):
        if theta < 0 or theta > 1:
            return -np.inf  # log(0) for theta outside [0, 1] is -inf
        else:
            likelihood = np.prod(theta**data.sum() * (1-
theta)**(n*len(data)-data.sum()))
            prior = theta**(1-1) * (1-theta)**(1-1)
            return np.log(likelihood * prior)

    samples = [theta0]
    current_theta = theta0

    for _ in range(n_samples):
        proposed_theta = np.random.normal(current_theta, 0.1)
        log_alpha = log_posterior(proposed_theta, data, n) -
log_posterior(current_theta, data, n)

        if np.log(np.random.uniform(0, 1)) < log_alpha:
            current_theta = proposed_theta

        samples.append(current_theta)

    return np.array(samples)

# Number of samples for Importance Sampling and MCMC
num_samples = 10000

# Obtain posterior distributions
importance_samples = importance_sampling_posterior(num_samples)
mcmc_samples = metropolis_hastings_posterior(0.5, num_samples)

# Plotting
plt.figure(figsize=(6, 8))

# Analytical Posterior
plt.subplot(3, 1, 1)
plt.plot(analytical_theta, analytical_posterior, label='Analytical
Posterior (Beta(135, 67))', color='blue')
plt.xlim([0, 1])
plt.ylim([0, 15])
plt.title('Posterior Distributions of θ')
plt.xlabel('θ')
```

```
plt.ylabel('Density')
plt.legend()
plt.grid(True)

# Importance Sampling Posterior
plt.subplot(3, 1, 2)
plt.hist(importance_samples, bins=50, density=True, label='Importance
Sampling', alpha=0.7, color='green')
plt.xlim([0, 1])
plt.ylim([0, 15])
plt.xlabel('θ')
plt.ylabel('Density')
plt.legend()
plt.grid(True)

# MCMC Posterior
plt.subplot(3, 1, 3)
plt.hist(mcmc_samples, bins=50, density=True, label='MCMC (Metropolis-
Hastings)', alpha=0.7, color='orange')
plt.xlim([0, 1])
plt.ylim([0, 15])
plt.xlabel('θ')
plt.ylabel('Density')
plt.legend()
plt.grid(True)

plt.tight_layout()
plt.show()
```



Part 2: ***Writing your own sampler for Bayesian inference***

2.5.1 and *2.5.2*

```
import pandas as pd
import numpy as np
from scipy.stats import norm

# Load the data
url =
"https://raw.githubusercontent.com/yadavhimanshu059/CGS698C/main/notes
/Data/word-recognition-times.csv"
dat = pd.read_csv(url)

# Likelihood function
def log_likelihood(alpha, beta, sigma, RT, type):
    mu = alpha + beta * type
    return np.sum(norm.logpdf(RT, loc=mu, scale=sigma))

# Priors
def log_prior_alpha(alpha):
    return norm.logpdf(alpha, loc=400, scale=50)

def log_prior_beta(beta):
    if beta > 0:
        return norm.logpdf(beta, loc=0, scale=50)
    else:
        return -np.inf  # log(0) for beta <= 0 is -inf

# Metropolis-Hastings algorithm
def metropolis_hastings(RT, type, initial_values, n_iter,
proposal_sd):
    alpha = initial_values[0]
    beta = initial_values[1]
    sigma = 30

    samples = np.zeros((n_iter, 2))
    samples[0, :] = initial_values
    accept = 0

    for t in range(1, n_iter):
        # Propose new values
        alpha_proposal = np.random.normal(alpha, proposal_sd[0])
        beta_proposal = np.random.normal(beta, proposal_sd[1])

        # Compute log-probability of current and proposed values
        log_prob_current = log_likelihood(alpha, beta, sigma, RT,
type) + log_prior_alpha(alpha) + log_prior_beta(beta)
        log_prob_proposal = log_likelihood(alpha_proposal,
beta_proposal, sigma, RT, type) + log_prior_alpha(alpha_proposal) +
log_prior_beta(beta_proposal)
```

```
        # Accept or reject the proposal
        log_ratio = log_prob_proposal - log_prob_current
        if np.log(np.random.uniform(0, 1)) < log_ratio:
            alpha = alpha_proposal
            beta = beta_proposal
            accept += 1

        samples[t, :] = [alpha, beta]

    acceptance_rate = accept / n_iter
    return samples, acceptance_rate

# Parameters for MCMC
initial_values = [400, 1]  # starting values for alpha and beta
n_iter = 50000
proposal_sd = [10, 0.1]  # standard deviations for proposal
distributions

# Filter data for words (type = 0) and non-words (type = 1)
RT_words = dat['RT'][dat['type'] == 0].values
RT_nonwords = dat['RT'][dat['type'] == 1].values

# Run MCMC for words and non-words separately
np.random.seed(123)  # for reproducibility
samples_words, acceptance_rate_words = metropolis_hastings(RT_words,
np.zeros_like(RT_words), initial_values, n_iter, proposal_sd)
samples_nonwords, acceptance_rate_nonwords =
metropolis_hastings(RT_nonwords, np.ones_like(RT_nonwords),
initial_values, n_iter, proposal_sd)

# Burn-in (optional): Remove initial samples to reduce impact of
starting values
burn_in = 1000
samples_words = samples_words[burn_in:, :]
samples_nonwords = samples_nonwords[burn_in:, :]

# Calculate means of the posterior samples for alpha and beta
posterior_mean_words = np.mean(samples_words, axis=0)
posterior_mean_nonwords = np.mean(samples_nonwords, axis=0)

# Print final estimates of alpha and beta
print("Estimated parameters (words):")
print("Alpha:", posterior_mean_words[0])
print("Beta:", posterior_mean_words[1])
print("\n")

print("Estimated parameters (non-words):")
print("Alpha:", posterior_mean_nonwords[0])
print("Beta:", posterior_mean_nonwords[1])
print("\n")
```

```
# 2.5.2
# Calculate 95% credible intervals
credible_interval_words = np.percentile(samples_words, [2.5, 97.5],
axis=0)
credible_interval_nonwords = np.percentile(samples_nonwords, [2.5,
97.5], axis=0)

# Print results
print("95% Credible Interval for alpha (words):",
credible_interval_words[:, 0])
print("95% Credible Interval for beta (words):",
credible_interval_words[:, 1])
print("\n")

print("95% Credible Interval for alpha (non-words):",
credible_interval_nonwords[:, 0])
print("95% Credible Interval for beta (non-words):",
credible_interval_nonwords[:, 1])
```

```
Estimated parameters (words):
Alpha: 402.75344233538794
Beta: 6.440139362507579


Estimated parameters (non-words):
Alpha: 402.64799543843105
Beta: 7.373797459804245


95% Credible Interval for alpha (words): [303.0641298  497.55582721]
95% Credible Interval for beta (words): [ 0.27797191 27.21340885]


95% Credible Interval for alpha (non-words): [304.64225165
498.95580373]
95% Credible Interval for beta (non-words): [ 0.92800065 18.24094115]
```

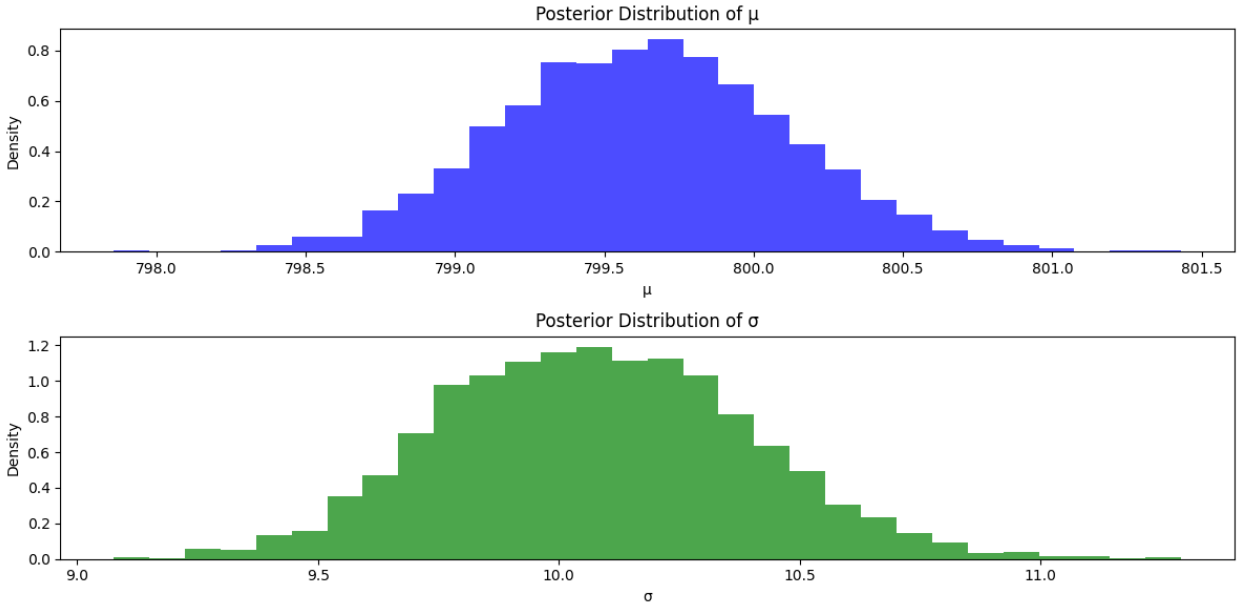
Part 3: ***Hamiltonian Monte Carlo sampler***

Exercise 3.1

```
import numpy as np
import matplotlib.pyplot as plt
from scipy.stats import norm

# Generate data
np.random.seed(123)
true_mu = 800
true_var = 100
```

```
y = np.random.normal(loc=true_mu, scale=np.sqrt(true_var), size=500)

# Define gradient function
def gradient(mu, sigma, y, n, m, s, a, b):
    grad_mu = (((n * mu) - np.sum(y)) / (sigma ** 2)) + ((mu - m) / (s
** 2))
    grad_sigma = (n / sigma) - (np.sum((y - mu) ** 2) / (sigma ** 3))
+ ((sigma - a) / (b ** 2))
    return np.array([grad_mu, grad_sigma])

# Define potential energy function
def V(mu, sigma, y, n, m, s, a, b):
    nlpd = -(np.sum(norm.logpdf(y, loc=mu, scale=sigma)) +
             norm.logpdf(mu, loc=m, scale=s) +
             norm.logpdf(sigma, loc=a, scale=b))
    return nlpd

# HMC sampler with log-scale acceptance probability
def HMC(y, n, m, s, a, b, step, L, initial_q, nsamp, nburn):
    mu_chain = np.zeros(nsamp)
    sigma_chain = np.zeros(nsamp)
    reject = 0

    mu_chain[0] = initial_q[0]
    sigma_chain[0] = initial_q[1]

    for i in range(1, nsamp):
        q = np.array([mu_chain[i - 1], sigma_chain[i - 1]])  # Current
position of the particle
        p = np.random.normal(0, 1, size=len(q))  # Generate random
momentum at the current position

        current_q = q.copy()
        current_p = p.copy()
        current_V = V(current_q[0], current_q[1], y, n, m, s, a, b)  #
Current potential energy
        current_T = np.sum(current_p ** 2) / 2  # Current kinetic
energy

        for l in range(L):
            p -= (step / 2) * gradient(q[0], q[1], y, n, m, s, a, b)
            q += step * p
            p -= (step / 2) * gradient(q[0], q[1], y, n, m, s, a, b)

        proposed_q = q.copy()
        proposed_p = p.copy()
        proposed_V = V(proposed_q[0], proposed_q[1], y, n, m, s, a, b)
        proposed_T = np.sum(proposed_p ** 2) / 2

        # Calculate log acceptance probability
```

```
        log_accept_prob = (current_V + current_T) - (proposed_V +
proposed_T)

        if np.log(np.random.uniform(0, 1)) < log_accept_prob:
            mu_chain[i] = proposed_q[0]
            sigma_chain[i] = proposed_q[1]
        else:
            mu_chain[i] = mu_chain[i - 1]
            sigma_chain[i] = sigma_chain[i - 1]
            reject += 1

    return np.vstack((mu_chain[nburn:], sigma_chain[nburn:])).T


# Parameters for HMC sampler
m = 1000
s = 100
a = 10
b = 2
step = 0.02
L = 12
initial_q = np.array([1000, 11])
nsamp = 6000
nburn = 2000

# Run HMC sampler
samples = HMC(y=y, n=len(y), m=m, s=s, a=a, b=b, step=step, L=L,
initial_q=initial_q, nsamp=nsamp, nburn=nburn)

# Plotting
plt.figure(figsize=(12, 6))

plt.subplot(2, 1, 1)
plt.hist(samples[:, 0], bins=30, density=True, color='blue',
alpha=0.7)
plt.title('Posterior Distribution of μ')
plt.xlabel('μ')
plt.ylabel('Density')

plt.subplot(2, 1, 2)
plt.hist(samples[:, 1], bins=30, density=True, color='green',
alpha=0.7)
plt.title('Posterior Distribution of σ')
plt.xlabel('σ')
plt.ylabel('Density')

plt.tight_layout()
plt.show()
```



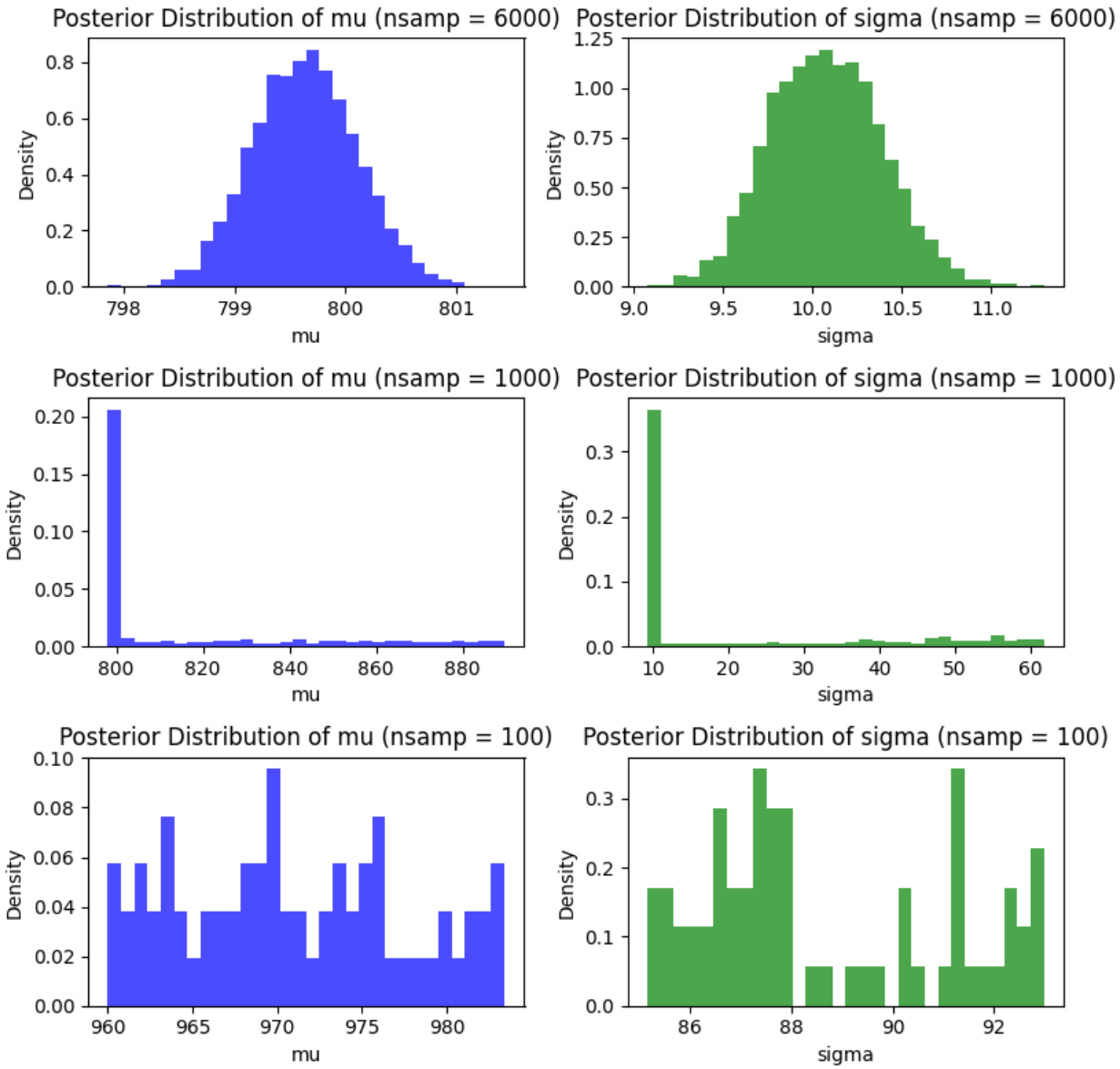
Exercise 3.2

```
import numpy as np
import matplotlib.pyplot as plt
from scipy.stats import norm

# Generate data
np.random.seed(123)
true_mu = 800
true_var = 100
y = np.random.normal(loc=true_mu, scale=np.sqrt(true_var), size=500)

# Define gradient function
def gradient(mu, sigma, y, n, m, s, a, b):
    grad_mu = (((n * mu) - np.sum(y)) / (sigma ** 2)) + ((mu - m) / (s
** 2))
    grad_sigma = (n / sigma) - (np.sum((y - mu) ** 2) / (sigma ** 3))
+ ((sigma - a) / (b ** 2))
    return np.array([grad_mu, grad_sigma])

# Define potential energy function
def V(mu, sigma, y, n, m, s, a, b):
    nlpd = -(np.sum(norm.logpdf(y, mu, sigma)) + norm.logpdf(mu, m, s)
+ norm.logpdf(sigma, a, b))
    return nlpd

# HMC sampler with log-scale acceptance probability
def HMC(y, n, m, s, a, b, step, L, initial_q, nsamp):
    nburn = nsamp // 3  # Set burn-in samples as one-third of nsamp

    mu_chain = np.zeros(nsamp)
```

```
    sigma_chain = np.zeros(nsamp)
    reject = 0

    mu_chain[0] = initial_q[0]
    sigma_chain[0] = initial_q[1]

    for i in range(1, nsamp):
        q = np.array([mu_chain[i - 1], sigma_chain[i - 1]])  # Current
position of the particle
        p = np.random.normal(0, 1, size=len(q))  # Generate random
momentum at the current position

        current_q = q.copy()
        current_p = p.copy()
        current_V = V(current_q[0], current_q[1], y, n, m, s, a, b)  #
Current potential energy
        current_T = np.sum(current_p ** 2) / 2  # Current kinetic
energy

        for l in range(L):
            p -= (step / 2) * gradient(q[0], q[1], y, n, m, s, a, b)
            q += step * p
            p -= (step / 2) * gradient(q[0], q[1], y, n, m, s, a, b)

        proposed_q = q.copy()
        proposed_p = p.copy()
        proposed_V = V(proposed_q[0], proposed_q[1], y, n, m, s, a, b)
        proposed_T = np.sum(proposed_p ** 2) / 2

        # Calculate log acceptance probability
        log_accept_prob = (current_V + current_T) - (proposed_V +
proposed_T)

        if np.log(np.random.uniform(0, 1)) < log_accept_prob:
            mu_chain[i] = proposed_q[0]
            sigma_chain[i] = proposed_q[1]
        else:
            mu_chain[i] = mu_chain[i - 1]
            sigma_chain[i] = sigma_chain[i - 1]
            reject += 1

    return np.vstack((mu_chain[nburn:], sigma_chain[nburn:])).T

# Parameters for HMC sampler
m = 1000
s = 100
a = 10
b = 2
L = 12
initial_q = np.array([1000, 11])
```

```
step = 0.02

# Run HMC sampler for nsamp = 6000
nsamp = 6000
samples_6000 = HMC(y=y, n=len(y), m=m, s=s, a=a, b=b, step=step, L=L,
initial_q=initial_q, nsamp=nsamp)

# Run HMC sampler for nsamp = 1000
nsamp = 1000
samples_1000 = HMC(y=y, n=len(y), m=m, s=s, a=a, b=b, step=step, L=L,
initial_q=initial_q, nsamp=nsamp)

# Run HMC sampler for nsamp = 100
nsamp = 100
samples_100 = HMC(y=y, n=len(y), m=m, s=s, a=a, b=b, step=step, L=L,
initial_q=initial_q, nsamp=nsamp)

# Plotting the posteriors
plt.figure(figsize=(8, 8))

# Plot for nsamp = 6000
plt.subplot(3, 2, 1)
plt.hist(samples_6000[:, 0], bins=30, density=True, color='blue',
alpha=0.7)
plt.title('Posterior Distribution of mu (nsamp = 6000)')
plt.xlabel('mu')
plt.ylabel('Density')

plt.subplot(3, 2, 2)
plt.hist(samples_6000[:, 1], bins=30, density=True, color='green',
alpha=0.7)
plt.title('Posterior Distribution of sigma (nsamp = 6000)')
plt.xlabel('sigma')
plt.ylabel('Density')

# Plot for nsamp = 1000
plt.subplot(3, 2, 3)
plt.hist(samples_1000[:, 0], bins=30, density=True, color='blue',
alpha=0.7)
plt.title('Posterior Distribution of mu (nsamp = 1000)')
plt.xlabel('mu')
plt.ylabel('Density')

plt.subplot(3, 2, 4)
plt.hist(samples_1000[:, 1], bins=30, density=True, color='green',
alpha=0.7)
plt.title('Posterior Distribution of sigma (nsamp = 1000)')
plt.xlabel('sigma')
plt.ylabel('Density')
```

```
# Plot for nsamp = 100
plt.subplot(3, 2, 5)
plt.hist(samples_100[:, 0], bins=30, density=True, color='blue',
alpha=0.7)
plt.title('Posterior Distribution of mu (nsamp = 100)')
plt.xlabel('mu')
plt.ylabel('Density')

plt.subplot(3, 2, 6)
plt.hist(samples_100[:, 1], bins=30, density=True, color='green',
alpha=0.7)
plt.title('Posterior Distribution of sigma (nsamp = 100)')
plt.xlabel('sigma')
plt.ylabel('Density')

plt.tight_layout()
plt.show()
```

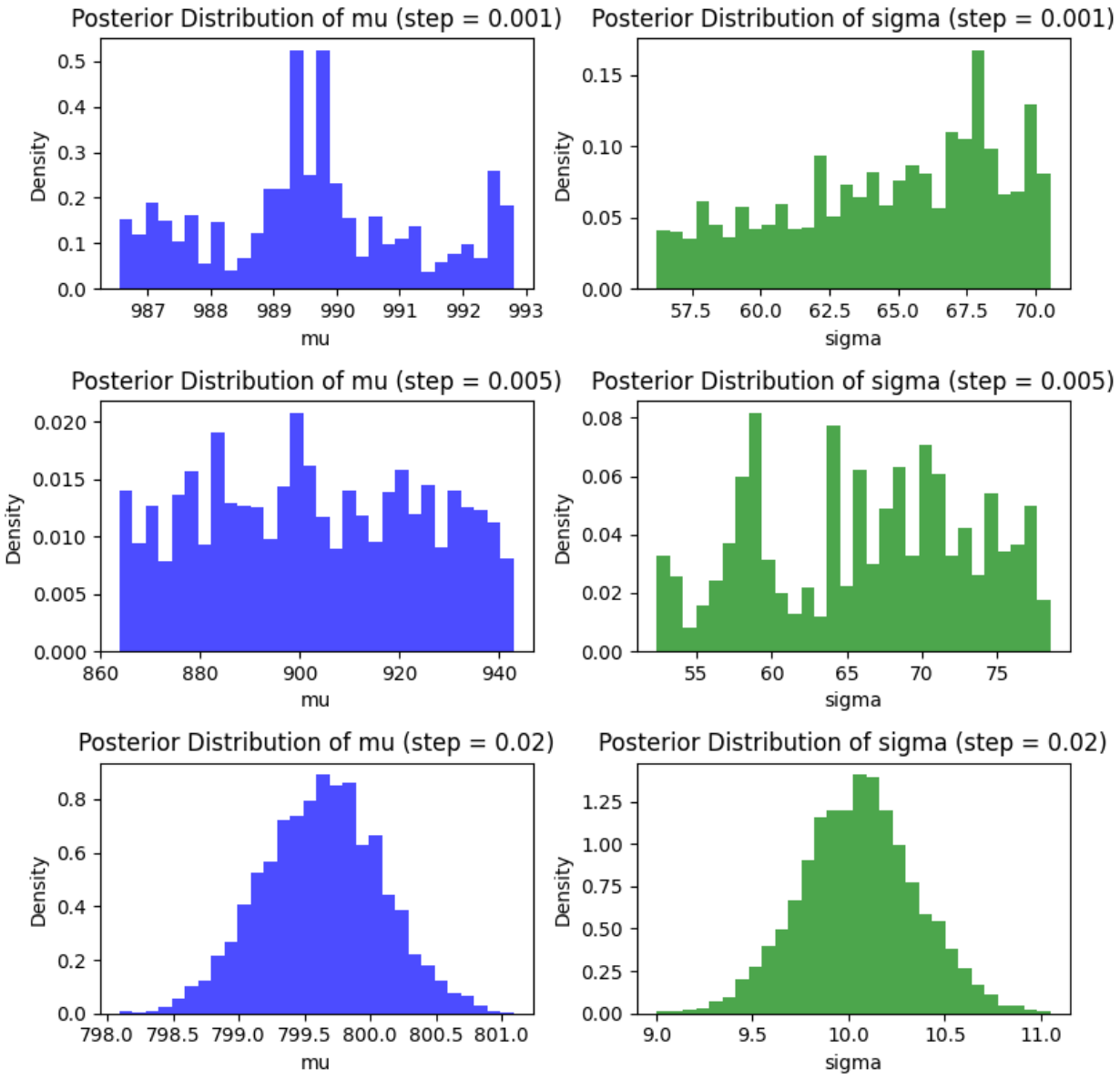


Exercise 3.3

```
import numpy as np
import matplotlib.pyplot as plt
from scipy.stats import norm

# Generate data
np.random.seed(123)
true_mu = 800
true_var = 100
y = np.random.normal(loc=true_mu, scale=np.sqrt(true_var), size=500)

# Define gradient function
def gradient(mu, sigma, y, n, m, s, a, b):
```

```
    grad_mu = (((n * mu) - np.sum(y)) / (sigma ** 2)) + ((mu - m) / (s
** 2))
    grad_sigma = (n / sigma) - (np.sum((y - mu) ** 2) / (sigma ** 3))
+ ((sigma - a) / (b ** 2))
    return np.array([grad_mu, grad_sigma])

# Define potential energy function
def V(mu, sigma, y, n, m, s, a, b):
    nlpd = -(np.sum(norm.logpdf(y, mu, sigma)) + norm.logpdf(mu, m, s)
+ norm.logpdf(sigma, a, b))
    return nlpd

# HMC sampler with log-scale acceptance probability
def HMC(y, n, m, s, a, b, step, L, initial_q, nsamp):
    nburn = nsamp // 3  # Set burn-in samples as one-third of nsamp

    mu_chain = np.zeros(nsamp)
    sigma_chain = np.zeros(nsamp)
    reject = 0

    mu_chain[0] = initial_q[0]
    sigma_chain[0] = initial_q[1]

    for i in range(1, nsamp):
        q = np.array([mu_chain[i - 1], sigma_chain[i - 1]])  # Current
position of the particle
        p = np.random.normal(0, 1, size=len(q))  # Generate random
momentum at the current position

        current_q = q.copy()
        current_p = p.copy()
        current_V = V(current_q[0], current_q[1], y, n, m, s, a, b)  #
Current potential energy
        current_T = np.sum(current_p ** 2) / 2  # Current kinetic
energy

        for l in range(L):
            p -= (step / 2) * gradient(q[0], q[1], y, n, m, s, a, b)
            q += step * p
            p -= (step / 2) * gradient(q[0], q[1], y, n, m, s, a, b)

        proposed_q = q.copy()
        proposed_p = p.copy()
        proposed_V = V(proposed_q[0], proposed_q[1], y, n, m, s, a, b)
        proposed_T = np.sum(proposed_p ** 2) / 2

        # Calculate log acceptance probability
        log_accept_prob = (current_V + current_T) - (proposed_V +
proposed_T)
```

```
        if np.log(np.random.uniform(0, 1)) < log_accept_prob:
            mu_chain[i] = proposed_q[0]
            sigma_chain[i] = proposed_q[1]
        else:
            mu_chain[i] = mu_chain[i - 1]
            sigma_chain[i] = sigma_chain[i - 1]
            reject += 1

    return np.vstack((mu_chain[nburn:], sigma_chain[nburn:])).T

# Parameters for HMC sampler
m = 1000
s = 100
a = 10
b = 2
L = 12
initial_q = np.array([1000, 11])
nsamp = 6000

# List of step sizes to compare
steps = [0.001, 0.005, 0.02]

# Run HMC sampler for each step size
posteriors = []
for step_size in steps:
    samples = HMC(y=y, n=len(y), m=m, s=s, a=a, b=b, step=step_size,
L=L, initial_q=initial_q, nsamp=nsamp)
    posteriors.append(samples)

# Plotting the posteriors for each step size
plt.figure(figsize=(8, 8))

for i, step_size in enumerate(steps):
    plt.subplot(3, 2, 2*i + 1)
    plt.hist(posteriors[i][:, 0], bins=30, density=True, color='blue',
alpha=0.7)
    plt.title(f'Posterior Distribution of mu (step = {step_size})')
    plt.xlabel('mu')
    plt.ylabel('Density')

    plt.subplot(3, 2, 2*i + 2)
    plt.hist(posteriors[i][:, 1], bins=30, density=True,
color='green', alpha=0.7)
    plt.title(f'Posterior Distribution of sigma (step = {step_size})')
    plt.xlabel('sigma')
    plt.ylabel('Density')

plt.tight_layout()
plt.show()
```

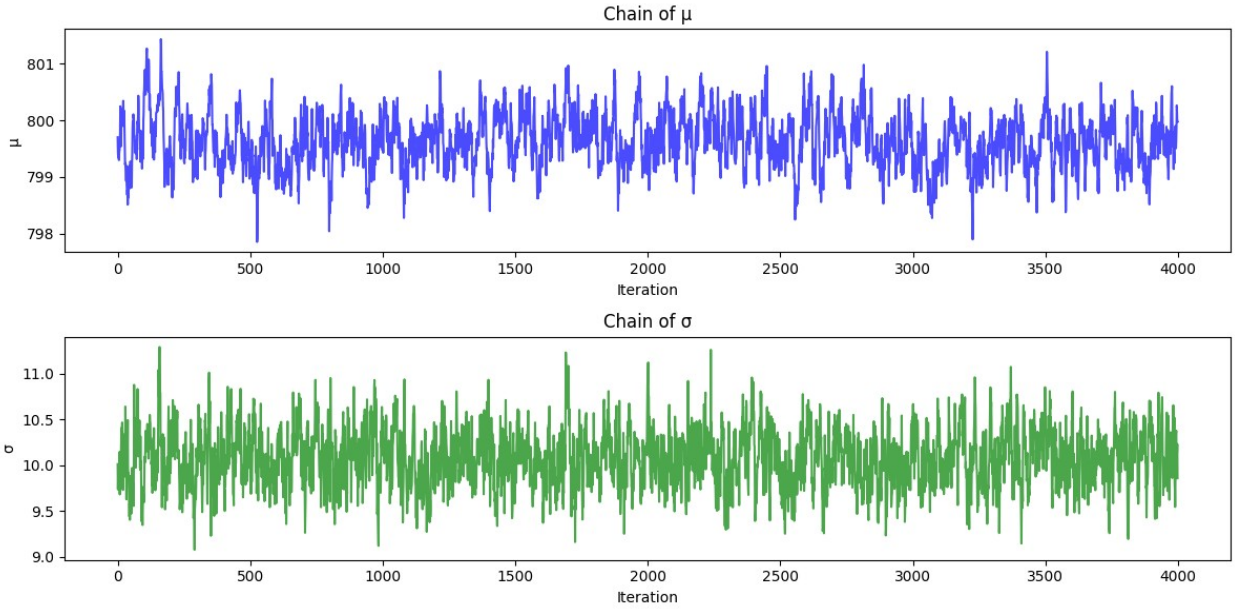


Exercise 3.4

```
import numpy as np
import matplotlib.pyplot as plt
from scipy.stats import norm

# Generate data
np.random.seed(123)
true_mu = 800
true_var = 100
y = np.random.normal(loc=true_mu, scale=np.sqrt(true_var), size=500)

# Define gradient function
def gradient(mu, sigma, y, n, m, s, a, b):
```

```
    grad_mu = (((n * mu) - np.sum(y)) / (sigma ** 2)) + ((mu - m) / (s
** 2))
    grad_sigma = (n / sigma) - (np.sum((y - mu) ** 2) / (sigma ** 3))
+ ((sigma - a) / (b ** 2))
    return np.array([grad_mu, grad_sigma])

# Define potential energy function
def V(mu, sigma, y, n, m, s, a, b):
    nlpd = -(np.sum(norm.logpdf(y, mu, sigma)) + norm.logpdf(mu, m, s)
+ norm.logpdf(sigma, a, b))
    return nlpd

# HMC sampler with log-scale acceptance probability
def HMC(y, n, m, s, a, b, step, L, initial_q, nsamp):
    nburn = nsamp // 3  # Set burn-in samples as one-third of nsamp

    mu_chain = np.zeros(nsamp)
    sigma_chain = np.zeros(nsamp)
    reject = 0

    mu_chain[0] = initial_q[0]
    sigma_chain[0] = initial_q[1]

    for i in range(1, nsamp):
        q = np.array([mu_chain[i - 1], sigma_chain[i - 1]])  # Current
position of the particle
        p = np.random.normal(0, 1, size=len(q))  # Generate random
momentum at the current position

        current_q = q.copy()
        current_p = p.copy()
        current_V = V(current_q[0], current_q[1], y, n, m, s, a, b)  #
Current potential energy
        current_T = np.sum(current_p ** 2) / 2  # Current kinetic
energy

        for l in range(L):
            p -= (step / 2) * gradient(q[0], q[1], y, n, m, s, a, b)
            q += step * p
            p -= (step / 2) * gradient(q[0], q[1], y, n, m, s, a, b)

        proposed_q = q.copy()
        proposed_p = p.copy()
        proposed_V = V(proposed_q[0], proposed_q[1], y, n, m, s, a, b)
        proposed_T = np.sum(proposed_p ** 2) / 2

        # Calculate log acceptance probability
        log_accept_prob = (current_V + current_T) - (proposed_V +
proposed_T)
```

```
        if np.log(np.random.uniform(0, 1)) < log_accept_prob:
            mu_chain[i] = proposed_q[0]
            sigma_chain[i] = proposed_q[1]
        else:
            mu_chain[i] = mu_chain[i - 1]
            sigma_chain[i] = sigma_chain[i - 1]
            reject += 1

    return mu_chain[nburn:], sigma_chain[nburn:], reject

# Parameters for HMC sampler
m = 1000
s = 100
a = 10
b = 2
L = 12
initial_q = np.array([1000, 11])
nsamp = 6000
step = 0.02

# Run HMC sampler
mu_chain, sigma_chain, reject = HMC(y=y, n=len(y), m=m, s=s, a=a, b=b,
step=step, L=L, initial_q=initial_q, nsamp=nsamp)

# Plotting the chains
plt.figure(figsize=(12, 6))

plt.subplot(2, 1, 1)
plt.plot(mu_chain, color='blue', alpha=0.7)
plt.title('Chain of μ')
plt.xlabel('Iteration')
plt.ylabel('μ')

plt.subplot(2, 1, 2)
plt.plot(sigma_chain, color='green', alpha=0.7)
plt.title('Chain of σ')
plt.xlabel('Iteration')
plt.ylabel('σ')

plt.tight_layout()
plt.show()

print(f"Number of rejections: {reject}")
```




```
Number of rejections: 3
```

The issue I see is the following-

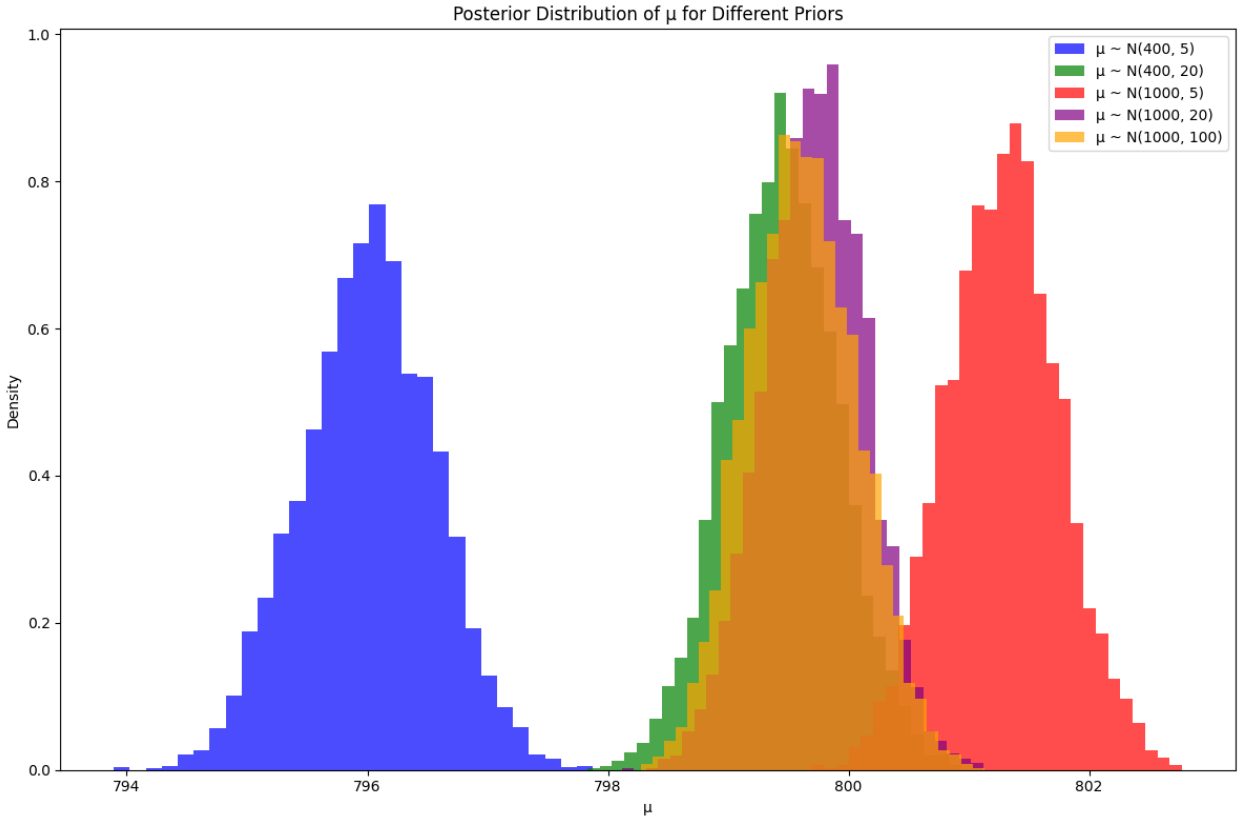
Non-convergence: Chains do not stabilize or exhibit a trend away from the initial values.

Exercise 3.5

```
import numpy as np
import matplotlib.pyplot as plt
from scipy.stats import norm

# Generate data (same as in Exercise 3.1)
np.random.seed(123)
true_mu = 800
true_var = 100
y = np.random.normal(loc=true_mu, scale=np.sqrt(true_var), size=500)

# Define gradient function
def gradient(mu, sigma, y, n, m, s, a, b):
    grad_mu = (((n * mu) - np.sum(y)) / (sigma ** 2)) + ((mu - m) / (s
** 2))
    grad_sigma = (n / sigma) - (np.sum((y - mu) ** 2) / (sigma ** 3))
+ ((sigma - a) / (b ** 2))
    return np.array([grad_mu, grad_sigma])

# Define potential energy function
def V(mu, sigma, y, n, m, s, a, b):
    nlpd = -(np.sum(norm.logpdf(y, mu, sigma)) + norm.logpdf(mu, m, s)
+ norm.logpdf(sigma, a, b))
    return nlpd
```

```
# HMC sampler
def HMC(y, n, m, s, a, b, step, L, initial_q, nsamp, nburn):
    mu_chain = np.zeros(nsamp)
    sigma_chain = np.zeros(nsamp)
    reject = 0

    mu_chain[0] = initial_q[0]
    sigma_chain[0] = initial_q[1]

    for i in range(1, nsamp):
        q = np.array([mu_chain[i - 1], sigma_chain[i - 1]])  # Current
position of the particle
        p = np.random.normal(0, 1, size=len(q))  # Generate random
momentum at the current position

        current_q = q.copy()
        current_p = p.copy()
        current_V = V(current_q[0], current_q[1], y, n, m, s, a, b)  #
Current potential energy
        current_T = np.sum(current_p ** 2) / 2  # Current kinetic
energy

        for l in range(L):
            p -= (step / 2) * gradient(q[0], q[1], y, n, m, s, a, b)
            q += step * p
            p -= (step / 2) * gradient(q[0], q[1], y, n, m, s, a, b)

        proposed_q = q.copy()
        proposed_p = p.copy()
        proposed_V = V(proposed_q[0], proposed_q[1], y, n, m, s, a, b)
        proposed_T = np.sum(proposed_p ** 2) / 2

        log_accept_prob = current_V + current_T - proposed_V -
proposed_T
        if np.log(np.random.uniform(0, 1)) < log_accept_prob:
            mu_chain[i] = proposed_q[0]
            sigma_chain[i] = proposed_q[1]
        else:
            mu_chain[i] = mu_chain[i - 1]
            sigma_chain[i] = sigma_chain[i - 1]
            reject += 1

    return mu_chain[nburn:], sigma_chain[nburn:], reject

# Parameters for HMC sampler (same as in Exercise 3.1)
m_values = [400, 400, 1000, 1000, 1000]
s_values = [5, 20, 5, 20, 100]
step = 0.02
L = 12
```

```
initial_q = np.array([1000, 11])
nsamp = 6000
nburn = nsamp // 3  # Burn-in samples set to approximately nsamp / 3

# Run HMC sampler for different priors on μ
posteriors = []
for m, s in zip(m_values, s_values):
    mu_chain, sigma_chain, reject = HMC(y=y, n=len(y), m=m, s=s, a=10,
b=2, step=step, L=L, initial_q=initial_q, nsamp=nsamp, nburn=nburn)
    posteriors.append(mu_chain)

# Plotting posterior distributions
plt.figure(figsize=(12, 8))
colors = ['blue', 'green', 'red', 'purple', 'orange']
labels = [f'μ ~ N({m}, {s})' for m, s in zip(m_values, s_values)]

for i, posterior in enumerate(posteriors):
    plt.hist(posterior, bins=30, density=True, alpha=0.7,
color=colors[i], label=labels[i])

plt.title('Posterior Distribution of μ for Different Priors')
plt.xlabel('μ')
plt.ylabel('Density')
plt.legend()
plt.tight_layout()
plt.show()
```


Posterior Distribution of μ for Different Priors
μ ~ N(400, 5)
μ ~ N(400, 20)
μ ~ N(1000, 5)
μ ~ N(1000, 20)
μ ~ N(1000, 100)
Density
μ
0.0
0.2
0.4
0.6
0.8
1.0
794
796
798
800
802