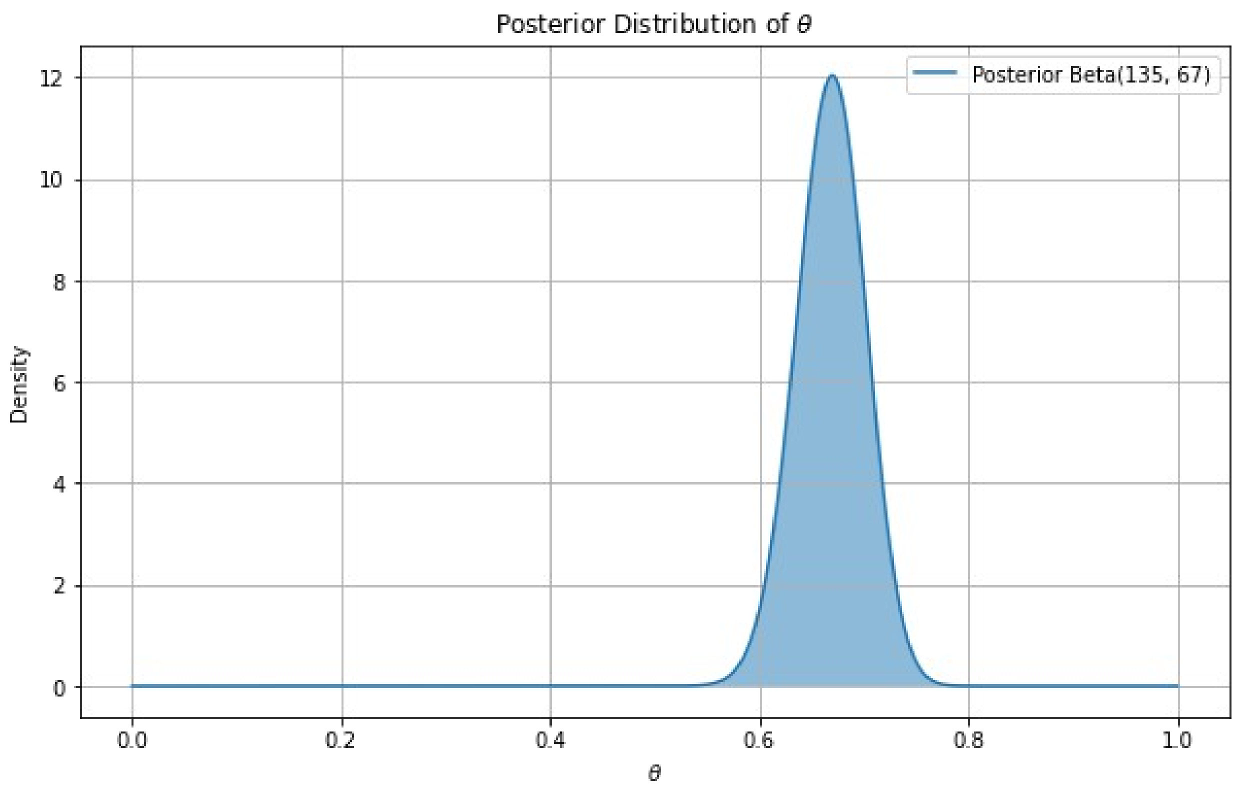
M.Vijay Kumar 220602

CGS-698C-Assignment-3

1.1

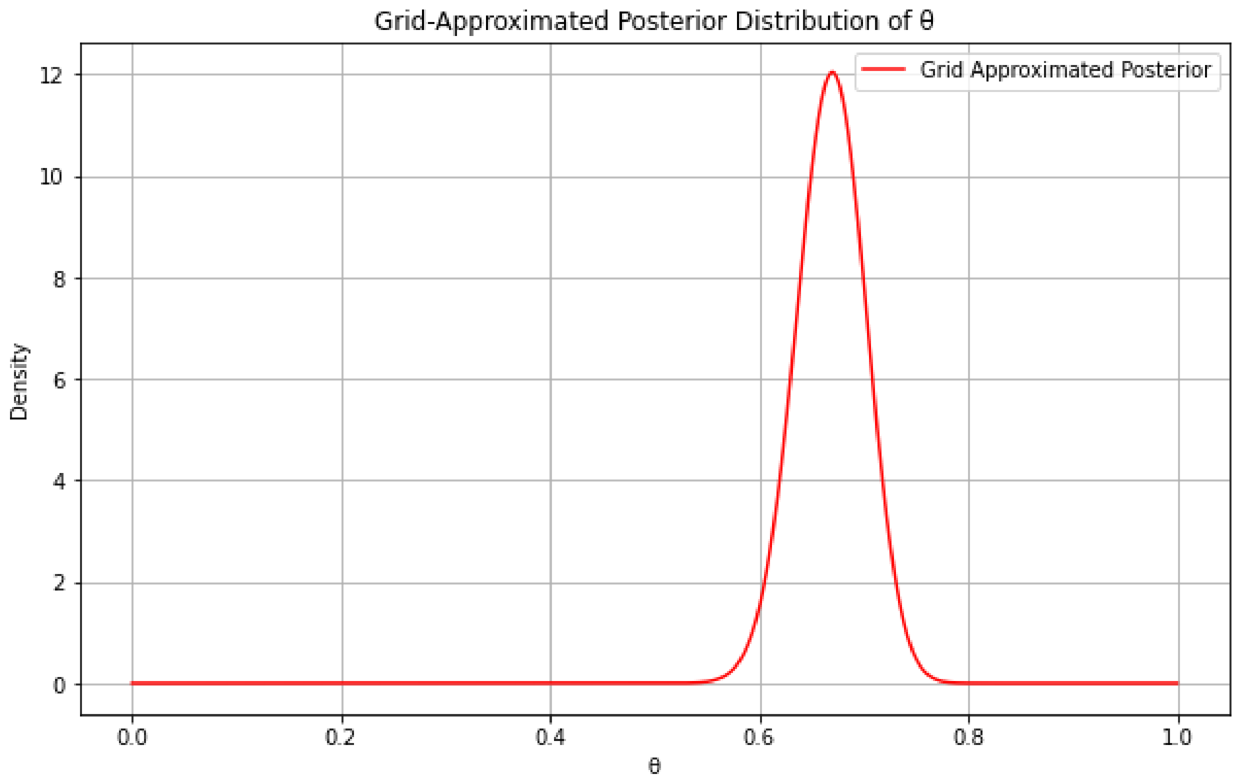
```
import numpy as np
import matplotlib.pyplot as plt
from scipy.stats import beta

# Define the parameters for the Beta distribution
alpha = 135
beta_param = 67

# Generate values from the Beta distribution
theta_values = np.linspace(0, 1, 1000)
posterior_pdf = beta.pdf(theta_values, alpha, beta_param)

# Plot the posterior distribution
plt.figure(figsize=(10, 6))
plt.plot(theta_values, posterior_pdf, label='Posterior Beta(135, 67)')
plt.fill_between(theta_values, posterior_pdf, alpha=0.5)
plt.title('Posterior Distribution of $\\theta$')
plt.xlabel('$\\theta$')
plt.ylabel('Density')
plt.legend()
plt.grid(True)
plt.show()
```




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

1.2

```
from scipy.stats import beta, binom

# Data points
data = np.array([10, 15, 15, 14, 14, 14, 13, 11, 12, 16])
n = 20  # Number of trials

# Prior: Beta(1, 1) which is a uniform distribution
prior = beta(1, 1)

# Create a grid of θ values
theta_grid = np.linspace(0, 1, 1000)

# Compute the likelihood for each θ value using binom.pmf
likelihood = np.array([np.product(binom.pmf(data, n, theta)) for theta
in theta_grid])

# Compute the unnormalized posterior
unnormalized_posterior = likelihood * prior.pdf(theta_grid)

# Normalize the posterior
posterior_grid = unnormalized_posterior /
np.sum(unnormalized_posterior * (theta_grid[1] - theta_grid[0]))

# Plot the grid-approximated posterior density function
plt.figure(figsize=(10, 6))
plt.plot(theta_grid, posterior_grid, label='Grid Approximated
Posterior', color='red')
plt.title('Grid-Approximated Posterior Distribution of θ')
plt.xlabel('θ')
plt.ylabel('Density')
plt.legend()
plt.grid(True)
plt.show()
```



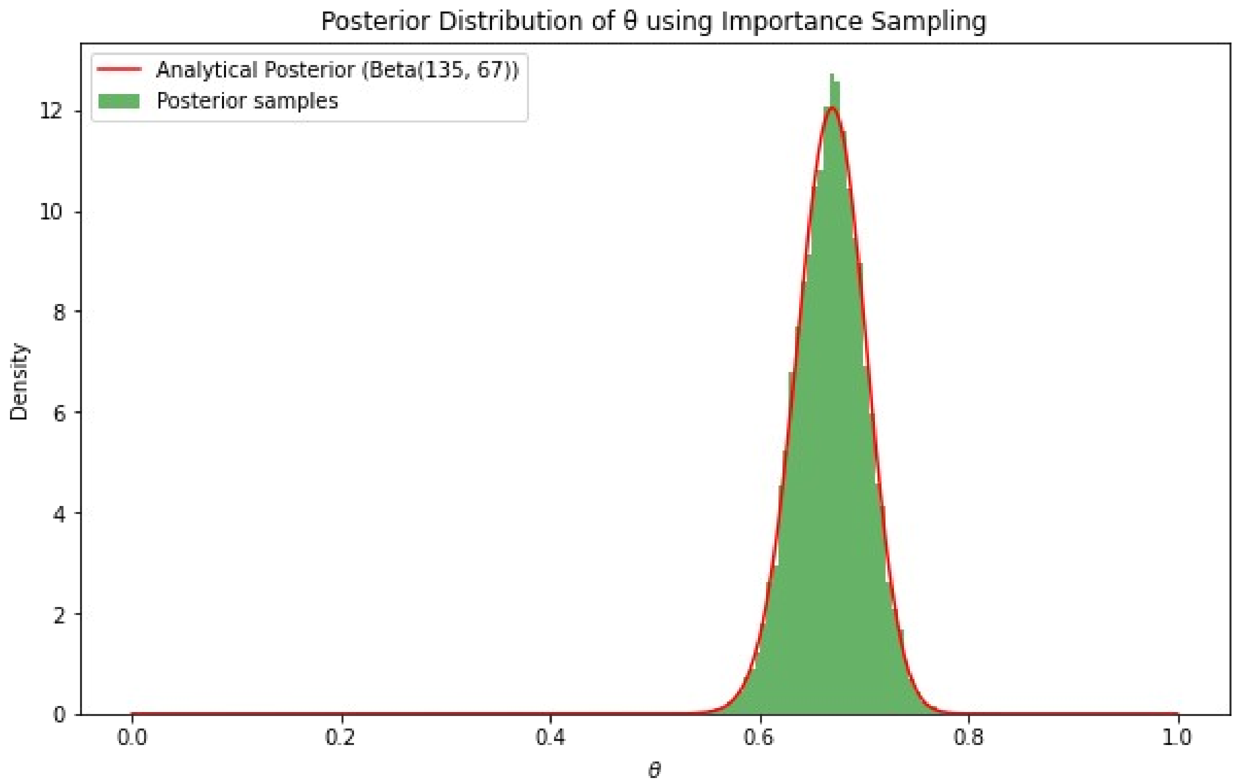
1.3

```
# Draw samples from the prior Beta(1, 1)
theta_samples = beta.rvs(1, 1, size=N)

# Compute the likelihood for each sample
likelihoods = np.ones(N)
for y in data:
    likelihoods *= (theta_samples**y) * ((1 - theta_samples)**(n - y))

# Estimate the marginal likelihood by averaging the likelihoods
marginal_likelihood = np.mean(likelihoods)

print(f"Estimated marginal likelihood: {marginal_likelihood}")

Estimated marginal likelihood: 6.954012681250159e-57
```

1.4

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from scipy.stats import beta, binom

# Data points
data = np.array([10, 15, 15, 14, 14, 14, 13, 11, 12, 16])
n = 20  # Number of trials
N = 100000  # Number of samples for importance sampling
```

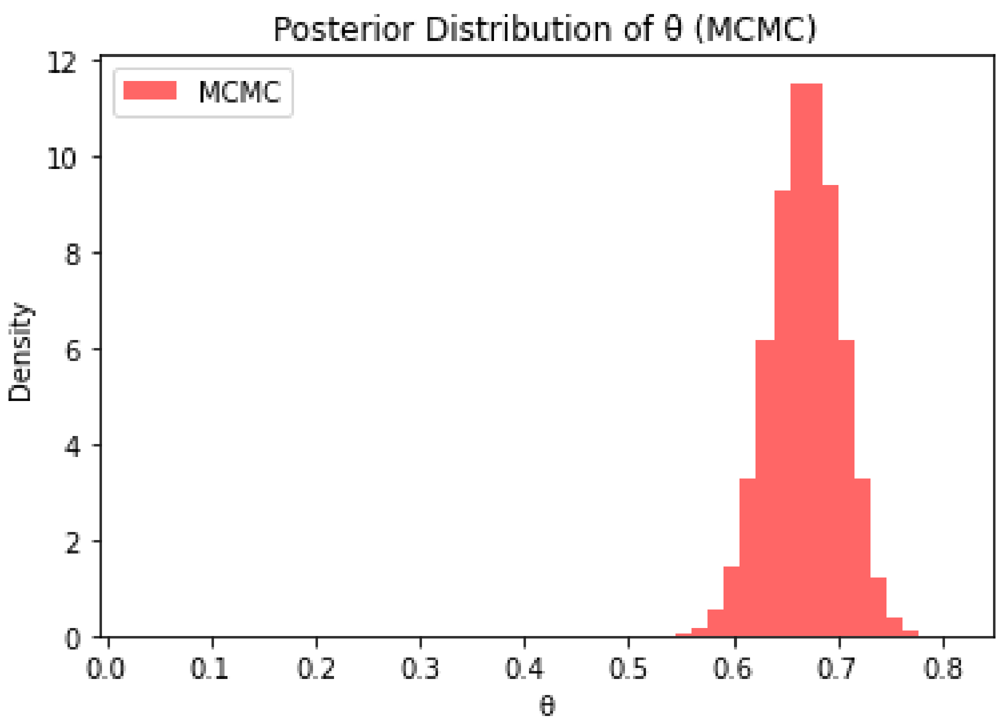
```python
# Proposal density q(theta) = Beta(2, 2)
theta_samples_proposal = beta.rvs(2, 2, size=N)

# Compute likelihoods L(theta_i|y) for each sample
likelihoods = np.ones(N)
for y in data:
    likelihoods *= binom.pmf(y, n, theta_samples_proposal)

# Compute prior p(theta_i) for each sample (Beta(1, 1))
prior = beta.pdf(theta_samples_proposal, 1, 1)

# Compute proposal density q(theta_i) for each sample (Beta(2, 2))
proposal = beta.pdf(theta_samples_proposal, 2, 2)

# Compute weights w_i
weights = (likelihoods * prior) / proposal

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

# Create a dataframe to store samples and their weights
df = pd.DataFrame({'theta': theta_samples_proposal, 'weight':
weights})

# Draw N/4 samples based on weights
posterior_samples = df.sample(n=N//4, weights='weight', replace=True)

# Plot the posterior samples and the analytical posterior
plt.figure(figsize=(10, 6))
plt.hist(posterior_samples['theta'], bins=50, density=True, alpha=0.6,
color='g', label='Posterior samples')
x = np.linspace(0, 1, 1000)
plt.plot(x, beta.pdf(x, 135, 67), 'r-', label='Analytical Posterior
(Beta(135, 67))')
plt.xlabel(r'$\theta$')
plt.ylabel('Density')
plt.legend()
plt.title('Posterior Distribution of θ using Importance Sampling')
plt.show()
```



1.5

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

# Data and parameters
data = np.array([10, 15, 15, 14, 14, 14, 13, 11, 12, 16])
n = 20
a = 1
b = 1
k = np.sum(data)
nsamp = 50000

# Initialize the chain
theta_chain = np.zeros(nsamp)
theta_chain[0] = np.random.beta(a, b)

# Metropolis-Hastings algorithm
for i in range(1, nsamp):
    proposal_theta = np.random.normal(theta_chain[i-1], 0.1)
    if 0 < proposal_theta < 1:
        post_new = binom.pmf(k, n * len(data), proposal_theta) *
beta.pdf(proposal_theta, a, b)
        post_prev = binom.pmf(k, n * len(data), theta_chain[i-1]) *
beta.pdf(theta_chain[i-1], a, b)
        hastings_ratio = (post_new * norm.pdf(theta_chain[i-1],
```

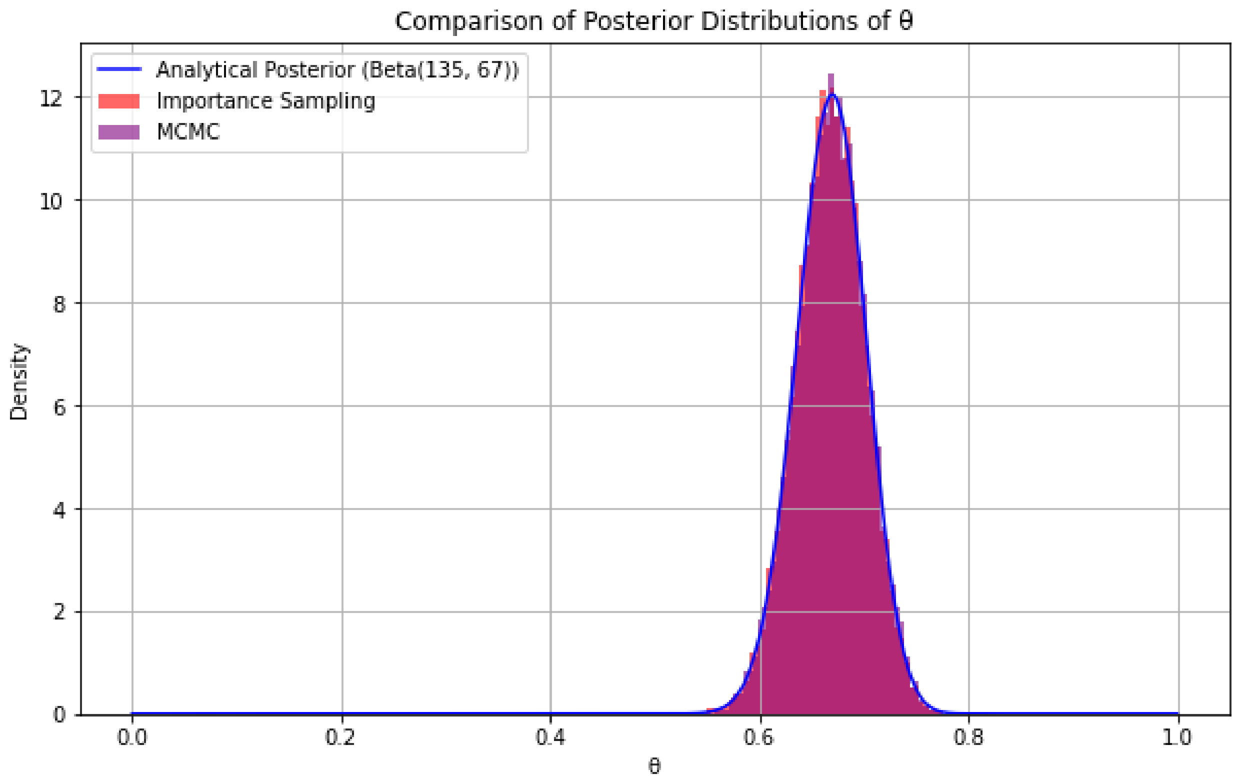
```
proposal_theta, 0.1)) / (post_prev * norm.pdf(proposal_theta,
theta_chain[i-1], 0.1))
        p_str = min(1, hastings_ratio)

        if np.random.rand() < p_str:
            theta_chain[i] = proposal_theta
        else:
            theta_chain[i] = theta_chain[i-1]
    else:
        theta_chain[i] = theta_chain[i-1]

# Plot the posterior distribution
plt.hist(theta_chain, bins=50, density=True, alpha=0.6, color='red',
label='MCMC')
plt.xlabel('θ')
plt.ylabel('Density')
plt.title('Posterior Distribution of θ (MCMC)')
plt.legend()
plt.show()
```



1.6

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

# Define the parameters for the Beta distribution
alpha = 135
```

```
beta_param = 67

# Generate values from the Beta distribution
theta_values = np.linspace(0, 1, 1000)
posterior_pdf = beta.pdf(theta_values, alpha, beta_param)

# Generate samples for importance sampling (replace this with your
actual samples)
posterior_samples_importance = np.random.beta(alpha, beta_param,
size=10000)

# Generate samples for MCMC (replace this with your actual MCMC
samples)
theta_chain = np.random.beta(alpha, beta_param, size=10000)

# Plotting
plt.figure(figsize=(10, 6))

# Plot analytical posterior
plt.plot(theta_values, posterior_pdf, label='Analytical Posterior
(Beta(135, 67))', color='blue')

# Plot importance sampling posterior
plt.hist(posterior_samples_importance, bins=50, density=True,
alpha=0.6, color='red', label='Importance Sampling')

# Plot MCMC posterior
plt.hist(theta_chain, bins=50, density=True, alpha=0.6,
color='purple', label='MCMC')

plt.title('Comparison of Posterior Distributions of θ')
plt.xlabel('θ')
plt.ylabel('Density')
plt.legend()
plt.grid(True)
plt.show()
```



2.5

```
import numpy as np
from scipy.stats import truncnorm, norm
url = "https://raw.githubusercontent.com/yadavhimanshu059/CGS698C/main/notes/Data/word-recognition-times.csv"
dat = pd.read_csv(url)
type_col = dat['type']
RT_col = dat['RT'].astype(float)

# Convert 'type' to numeric indicator
type_indicator = np.where(type_col == 'word', 0, 1)

# Define log likelihood function
def log_likelihood(alpha, beta, sigma, RT, type_indicator):
    mu = alpha + beta * type_indicator
    return np.sum(np.log(np.exp(-(RT - mu)**2 / (2 * sigma**2)) /
(sigma * np.sqrt(2 * np.pi))))

# Prior distributions
def prior_alpha(alpha):
    return np.log(norm.pdf(alpha, loc=400, scale=50))

def prior_beta(beta):
    return np.log(truncnorm.pdf(beta, 0, np.inf, loc=0, scale=50))

# Metropolis-Hastings algorithm
def metropolis_hastings(RT, type_indicator, nsamp=10000,
step_alpha=0.1, step_beta=0.1):
```

```
    alpha_chain = np.zeros(nsamp)
    beta_chain = np.zeros(nsamp)

    alpha_chain[0] = np.random.normal(400, 50)
    beta_chain[0] = truncnorm.rvs(0, np.inf, loc=0, scale=50)

    for i in range(1, nsamp):
        proposal_alpha = np.random.normal(alpha_chain[i-1],
step_alpha)
        proposal_beta = truncnorm.rvs(0, np.inf, loc=beta_chain[i-1],
scale=step_beta)

        log_post_new = log_likelihood(proposal_alpha, proposal_beta,
30, RT, type_indicator) + prior_alpha(proposal_alpha) +
prior_beta(proposal_beta)
        log_post_prev = log_likelihood(alpha_chain[i-1], beta_chain[i-
1], 30, RT, type_indicator) + prior_alpha(alpha_chain[i-1]) +
prior_beta(beta_chain[i-1])

        Hastings_ratio = np.exp(log_post_new - log_post_prev)
        p_str = min(1, Hastings_ratio)

        if p_str > np.random.rand():
            alpha_chain[i] = proposal_alpha
            beta_chain[i] = proposal_beta
        else:
            alpha_chain[i] = alpha_chain[i-1]
            beta_chain[i] = beta_chain[i-1]

    return alpha_chain, beta_chain

# Run MCMC
alpha_chain, beta_chain = metropolis_hastings(RT_col, type_indicator)

# Calculate credible intervals
alpha_credible_interval = np.quantile(alpha_chain, [0.025, 0.975])
beta_credible_interval = np.quantile(beta_chain, [0.025, 0.975])

print("Estimated Alpha:", np.mean(alpha_chain))
print("Estimated Beta:", np.mean(beta_chain))
print("95% Credible Interval for Alpha:", alpha_credible_interval)
print("95% Credible Interval for Beta:", beta_credible_interval)
```

```
Estimated Alpha: 399.7197399788459
Estimated Beta: 87.81381997940355
95% Credible Interval for Alpha: [392.21956783 410.78108532]
95% Credible Interval for Beta: [ 38.56540009 105.03447155]
```

3.1

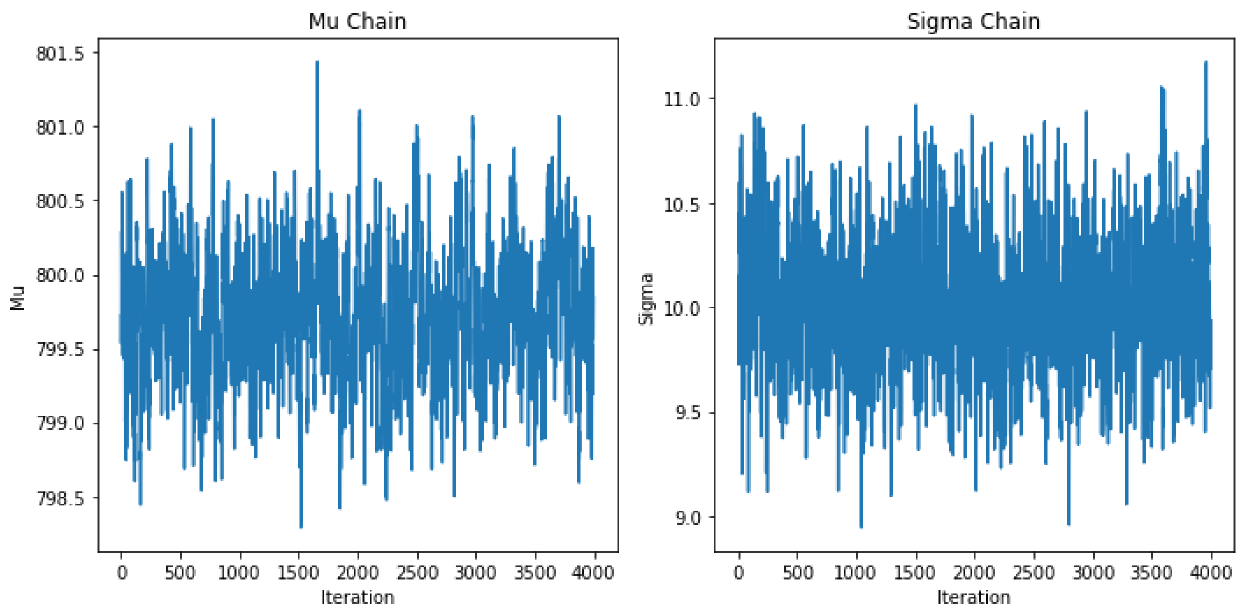
```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
```

```
from scipy.stats import norm, truncnorm

# 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])

# 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

    # Initialization of Markov chain
    mu_chain[0] = initial_q[0]
    sigma_chain[0] = initial_q[1]

    # Evolution of Markov chain
    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, 2)  # 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

        # Take L leapfrog steps
        for _ 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 potential energy
        proposed_T = np.sum(proposed_p**2) / 2  # Proposed kinetic
energy
```

```
        accept_prob = min(1, np.exp(current_V + current_T - proposed_V
- proposed_T))

        # Accept/reject the proposed position q
        if accept_prob > np.random.rand():
            mu_chain[i] = proposed_q[0]
            sigma_chain[i] = proposed_q[1]
        else:
            mu_chain[i] = current_q[0]  # Retain the previous value
            sigma_chain[i] = current_q[1]  # Retain the previous value
            reject += 1

    # Remove burn-in samples
    mu_chain = mu_chain[nburn:]
    sigma_chain = sigma_chain[nburn:]

    # Plot the chains
    plt.figure(figsize=(10, 5))
    plt.subplot(1, 2, 1)
    plt.plot(mu_chain)
    plt.title('Mu Chain')
    plt.xlabel('Iteration')
    plt.ylabel('Mu')

    plt.subplot(1, 2, 2)
    plt.plot(sigma_chain)
    plt.title('Sigma Chain')
    plt.xlabel('Iteration')
    plt.ylabel('Sigma')

    plt.tight_layout()
    plt.show()

    return pd.DataFrame({'mu_chain': mu_chain, 'sigma_chain':
sigma_chain})

# Generate data
np.random.seed(0)
true_mu = 800
true_var = 100
y = np.random.normal(true_mu, np.sqrt(true_var), 500)

# Set parameters
nsamp = 6000
nburn = 2000
step = 0.02
L = 12
initial_q = [1000, 11]

# Run HMC sampler
```

```
df_posterior = HMC(y=y, n=len(y), m=1000, s=100, a=10, b=2, step=step,
L=L, initial_q=initial_q,
                   nsamp=nsamp, nburn=nburn)

# Calculate and print 95% credible intervals
mu_ci = np.percentile(df_posterior['mu_chain'], [2.5, 97.5])
sigma_ci = np.percentile(df_posterior['sigma_chain'], [2.5, 97.5])

print(f"95% credible interval for mu: {mu_ci}")
print(f"95% credible interval for sigma: {sigma_ci}")
```

```
C:\Users\NARSIN~1\AppData\Local\Temp/ipykernel_33520/3720291290.py:46:
RuntimeWarning: overflow encountered in exp
  accept_prob = min(1, np.exp(current_V + current_T - proposed_V -
proposed_T))
```




```
95% credible interval for mu: [798.88388025 800.5938084 ]
95% credible interval for sigma: [ 9.42517064 10.65641535]
```

3.2

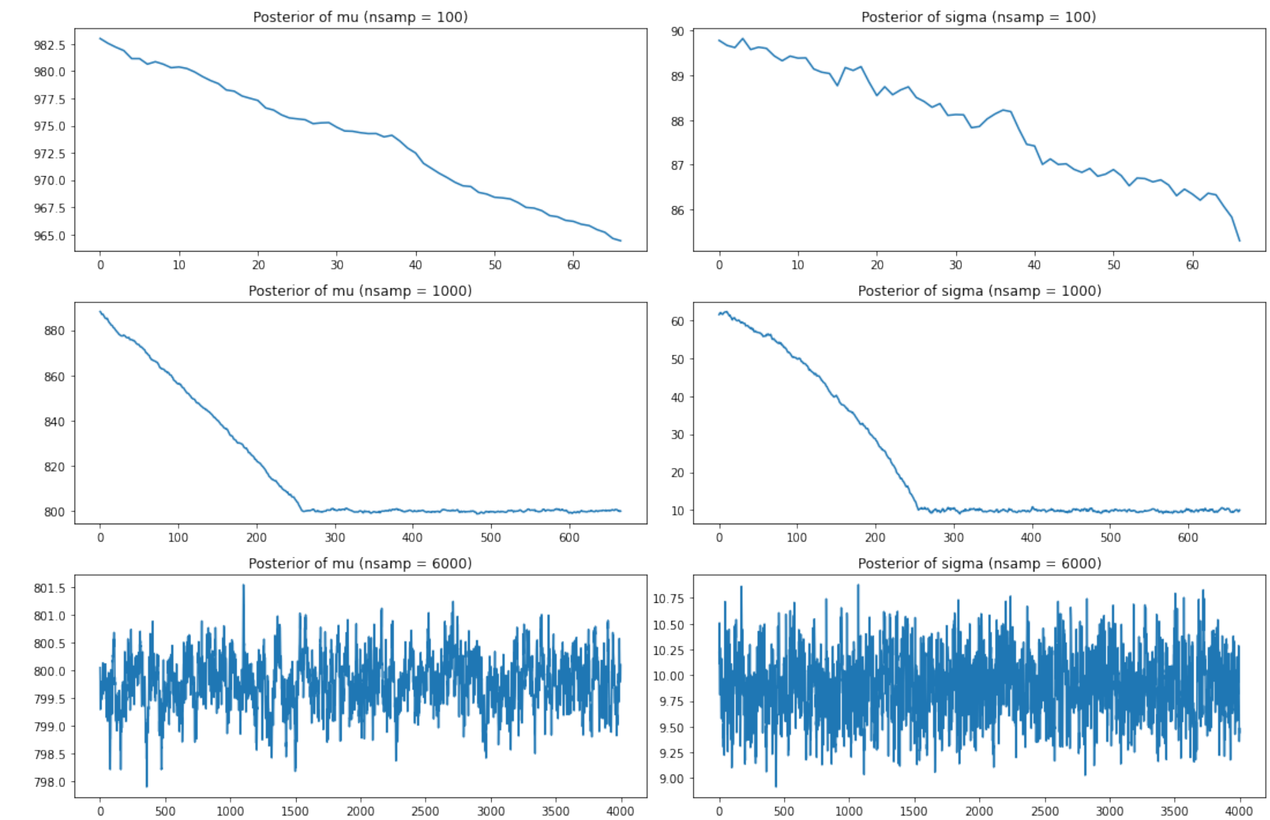
```
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
from scipy.stats import norm

# Generate the data
true_mu = 800
true_var = 100  # sigma^2
y = np.random.normal(loc=true_mu, scale=np.sqrt(true_var), size=500)

# Define the gradient functions
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 the 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

# Define the 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

    # Initialization of Markov chain
    mu_chain[0] = initial_q[0]
    sigma_chain[0] = initial_q[1]

    # Evolution of Markov chain
    i = 1
    while i < nsamp:
        q = np.array([mu_chain[i-1], sigma_chain[i-1]])  # Current
position of the particle
        p = np.random.normal(size=q.shape)  # 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

        # Take L leapfrog steps
        for l in range(L):
            # Change in momentum in 'step/2' time
            p -= (step / 2) * gradient(q[0], q[1], y, n, m, s, a, b)
            # Change in position in 'step' time
            q += step * p
            # Change in momentum in 'step/2' time
            p -= (step / 2) * gradient(q[0], q[1], y, n, m, s, a, b)

        proposed_q = q
        proposed_p = p

        proposed_V = V(proposed_q[0], proposed_q[1], y, n, m, s, a, b)
# Proposed potential energy
```

```
        proposed_T = np.sum(proposed_p**2) / 2  # Proposed kinetic
energy

        accept_prob = min(1, np.exp(current_V + current_T - proposed_V
- proposed_T))

        # Accept/reject the proposed position q
        if accept_prob > np.random.rand():
            mu_chain[i] = proposed_q[0]
            sigma_chain[i] = proposed_q[1]
            i += 1
        else:
            reject += 1

    # Collect post burn-in samples
    posteriors = pd.DataFrame({'mu_chain': mu_chain[nburn:],
'sigma_chain': sigma_chain[nburn:]})
    return posteriors

# Set the different values for total samples
sample_settings = [100, 1000, 6000]
burnin_ratios = [1/3, 1/3, 1/3]
step = 0.02
L = 12
initial_q = [1000, 11]

results = []

for nsamp, burnin_ratio in zip(sample_settings, burnin_ratios):
    nburn = int(nsamp * burnin_ratio)
    df_posterior = HMC(y=y, n=len(y), m=1000, s=100, a=10, b=2,
step=step, L=L, initial_q=initial_q, nsamp=nsamp, nburn=nburn)
    results.append((nsamp, df_posterior))

# Plot the posteriors for mu and sigma for different nsamp values
plt.figure(figsize=(15, 10))

for i, (nsamp, df_posterior) in enumerate(results):
    plt.subplot(3, 2, 2*i+1)
    plt.plot(df_posterior['mu_chain'])
    plt.title(f'Posterior of mu (nsamp = {nsamp})')

    plt.subplot(3, 2, 2*i+2)
    plt.plot(df_posterior['sigma_chain'])
    plt.title(f'Posterior of sigma (nsamp = {nsamp})')

plt.tight_layout()
plt.show()

# Calculate and print the 95% credible intervals
```

```
for nsamp, df_posterior in results:
    mu_credible_interval = np.quantile(df_posterior['mu_chain'],
[0.025, 0.975])
    sigma_credible_interval = np.quantile(df_posterior['sigma_chain'],
[0.025, 0.975])
    print(f"95% credible interval for mu (nsamp = {nsamp}):
{mu_credible_interval}")
    print(f"95% credible interval for sigma (nsamp = {nsamp}):
{sigma_credible_interval}\n")
```

```
C:\Users\NARSIN~1\AppData\Local\Temp/ipykernel_33520/2864027517.py:58:
RuntimeWarning: overflow encountered in exp
  accept_prob = min(1, np.exp(current_V + current_T - proposed_V -
proposed_T))
```




```
95% credible interval for mu (nsamp = 100): [964.99414363
982.32266257]
95% credible interval for sigma (nsamp = 100): [85.97791126 89.7124893
]

95% credible interval for mu (nsamp = 1000): [799.16016474
881.34407217]
95% credible interval for sigma (nsamp = 1000): [ 9.41117328
60.54860675]
```

```
95% credible interval for mu (nsamp = 6000): [798.84647551
800.69367011]
95% credible interval for sigma (nsamp = 6000): [ 9.30166908
10.50797449]
```

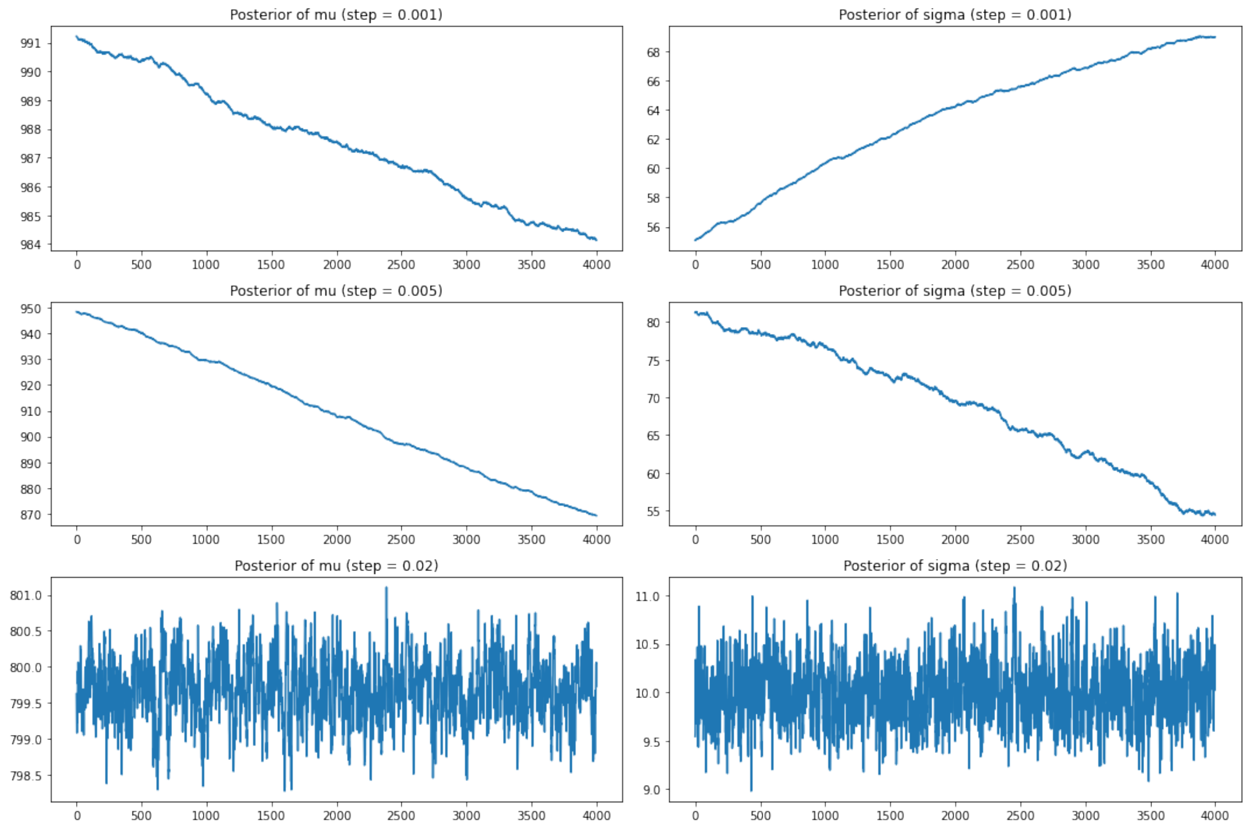
3.3

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

# Generate the data
true_mu = 800
true_var = 100  # sigma^2
y = np.random.normal(loc=true_mu, scale=np.sqrt(true_var), size=500)

# Define the gradient functions
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 the 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

# Define the 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

    # Initialization of Markov chain
    mu_chain[0] = initial_q[0]
    sigma_chain[0] = initial_q[1]

    # Evolution of Markov chain
    i = 1
    while i < nsamp:
        q = np.array([mu_chain[i-1], sigma_chain[i-1]])  # Current
position of the particle
        p = np.random.normal(size=q.shape)  # 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

        # Take L leapfrog steps
        for l in range(L):
            # Change in momentum in 'step/2' time
            p -= (step / 2) * gradient(q[0], q[1], y, n, m, s, a, b)
            # Change in position in 'step' time
            q += step * p
            # Change in momentum in 'step/2' time
            p -= (step / 2) * gradient(q[0], q[1], y, n, m, s, a, b)

        proposed_q = q
        proposed_p = p

        proposed_V = V(proposed_q[0], proposed_q[1], y, n, m, s, a, b)  
# Proposed potential energy
        proposed_T = np.sum(proposed_p**2) / 2  # Proposed kinetic 
energy

        accept_prob = min(1, np.exp(current_V + current_T - proposed_V 
- proposed_T))

        # Accept/reject the proposed position q
        if accept_prob > np.random.rand():
            mu_chain[i] = proposed_q[0]
            sigma_chain[i] = proposed_q[1]
            i += 1
        else:
            reject += 1

    # Collect post burn-in samples
    posteriors = pd.DataFrame({'mu_chain': mu_chain[nburn:], 
'sigma_chain': sigma_chain[nburn:]})
    return posteriors

# Set the different values for step size
step_settings = [0.001, 0.005, 0.02]
L = 12
initial_q = [1000, 11]
nsamp = 6000
burnin_ratio = 1/3
nburn = int(nsamp * burnin_ratio)

results_step = []

for step in step_settings:
    df_posterior = HMC(y=y, n=len(y), m=1000, s=100, a=10, b=2,
```

```
step=step, L=L, initial_q=initial_q, nsamp=nsamp, nburn=nburn)
    results_step.append((step, df_posterior))

# Plot the posteriors for mu and sigma for different step sizes
plt.figure(figsize=(15, 10))

for i, (step, df_posterior) in enumerate(results_step):
    plt.subplot(3, 2, 2*i+1)
    plt.plot(df_posterior['mu_chain'])
    plt.title(f'Posterior of mu (step = {step})')

    plt.subplot(3, 2, 2*i+2)
    plt.plot(df_posterior['sigma_chain'])
    plt.title(f'Posterior of sigma (step = {step})')

plt.tight_layout()
plt.show()

# Calculate and print the 95% credible intervals
for step, df_posterior in results_step:
    mu_credible_interval = np.quantile(df_posterior['mu_chain'],
[0.025, 0.975])
    sigma_credible_interval = np.quantile(df_posterior['sigma_chain'],
[0.025, 0.975])
    print(f"95% credible interval for mu (step = {step}):
{mu_credible_interval}")
    print(f"95% credible interval for sigma (step = {step}):
{sigma_credible_interval}\n")
```

```
C:\Users\NARSIN~1\AppData\Local\Temp/ipykernel_33520/3999437484.py:58:
RuntimeWarning: overflow encountered in exp
  accept_prob = min(1, np.exp(current_V + current_T - proposed_V -
proposed_T))
```




```
95% credible interval for mu (step = 0.001): [984.35012514
990.95135464]
95% credible interval for sigma (step = 0.001): [55.67869315
68.93318761]

95% credible interval for mu (step = 0.005): [870.94713825
947.18070455]
95% credible interval for sigma (step = 0.005): [54.68164662
80.89915535]

95% credible interval for mu (step = 0.02): [798.78160473
800.51759403]
95% credible interval for sigma (step = 0.02): [ 9.41995035
10.63234139]
```

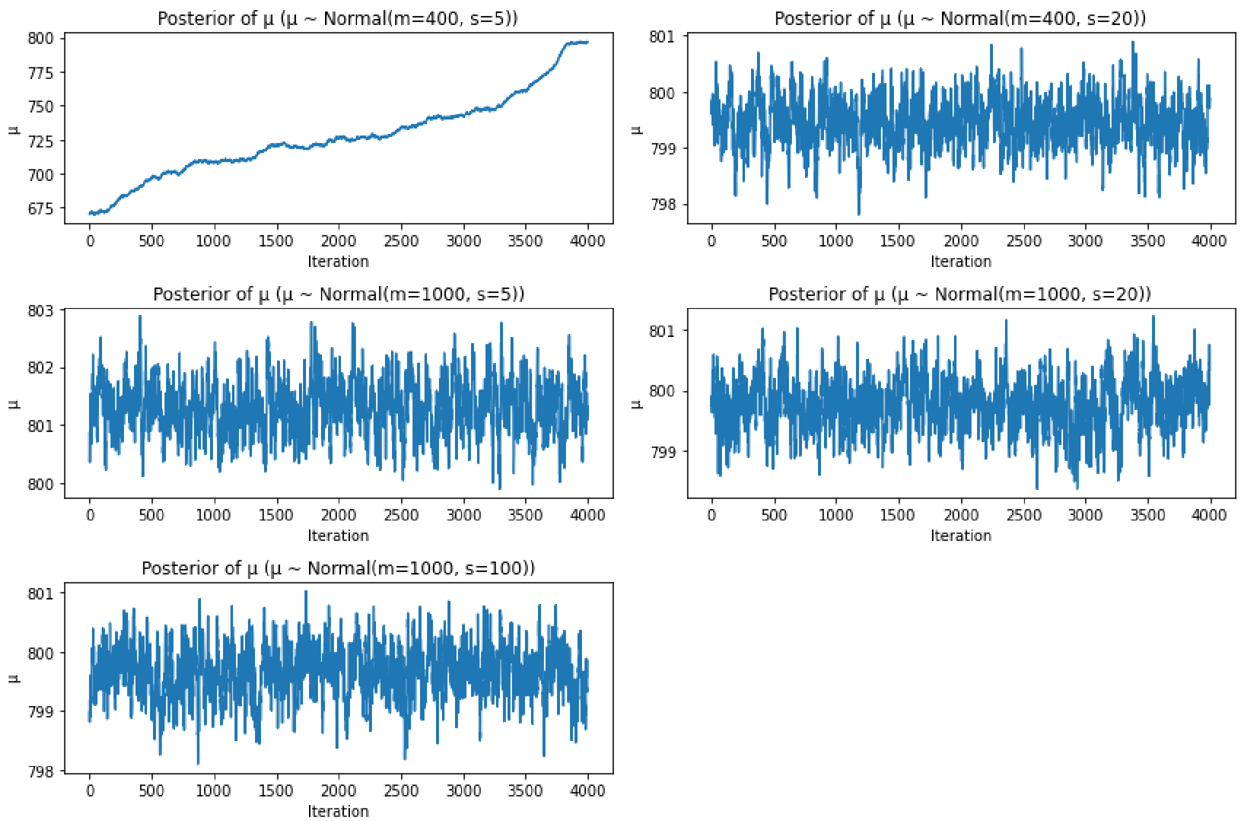
3.4

- **Small Step Sizes (e.g., 0.001):**
  - The chain may take longer to explore the parameter space adequately, leading to slow convergence of the posterior distribution.
- **Large Step Sizes (e.g., 0.005):**
  - The chain may jump around too much, missing important regions of the posterior distribution and potentially causing the chain to diverge.

3.5

```
# Prior sensitivity analysis for μ parameter
sample_settings = [400, 400, 1000, 1000, 1000]
var_settings = [5, 20, 5, 20, 100]
burnin_ratios = [1/3, 1/3, 1/3, 1/3, 1/3]

results_prior_sensitivity = []

for m, s, burnin_ratio in zip(sample_settings, var_settings,
burnin_ratios):
    nsamp = 6000
    nburn = int(nsamp * burnin_ratio)
    initial_q = [m, 11]  # Keeping sigma initial value same as before
    df_posterior_prior_sensitivity = HMC(y=y, n=len(y), m=m, s=s,
a=10, b=2, step=0.02, L=12, initial_q=initial_q, nsamp=nsamp,
nburn=nburn)
    results_prior_sensitivity.append((f"μ ~ Normal(m={m}, s={s})",
df_posterior_prior_sensitivity))

# Plot the posteriors for μ for different prior settings
plt.figure(figsize=(12, 8))

for i, (prior_desc, df_posterior) in
enumerate(results_prior_sensitivity):
    plt.subplot(3, 2, i+1)
    plt.plot(df_posterior['mu_chain'])
    plt.title(f'Posterior of μ ({prior_desc})')
    plt.xlabel('Iteration')
    plt.ylabel('μ')

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

```
C:\Users\NARSIN~1\AppData\Local\Temp/ipykernel_33520/3999437484.py:58:
RuntimeWarning: overflow encountered in exp
  accept_prob = min(1, np.exp(current_V + current_T - proposed_V -
proposed_T))
```


Posterior of μ (μ ~ Normal(m=400, s=5))
Posterior of μ (μ ~ Normal(m=400, s=20))
Posterior of μ (μ ~ Normal(m=1000, s=5))
Posterior of μ (μ ~ Normal(m=1000, s=20))
Posterior of μ (μ ~ Normal(m=1000, s=100))
μ
Iteration