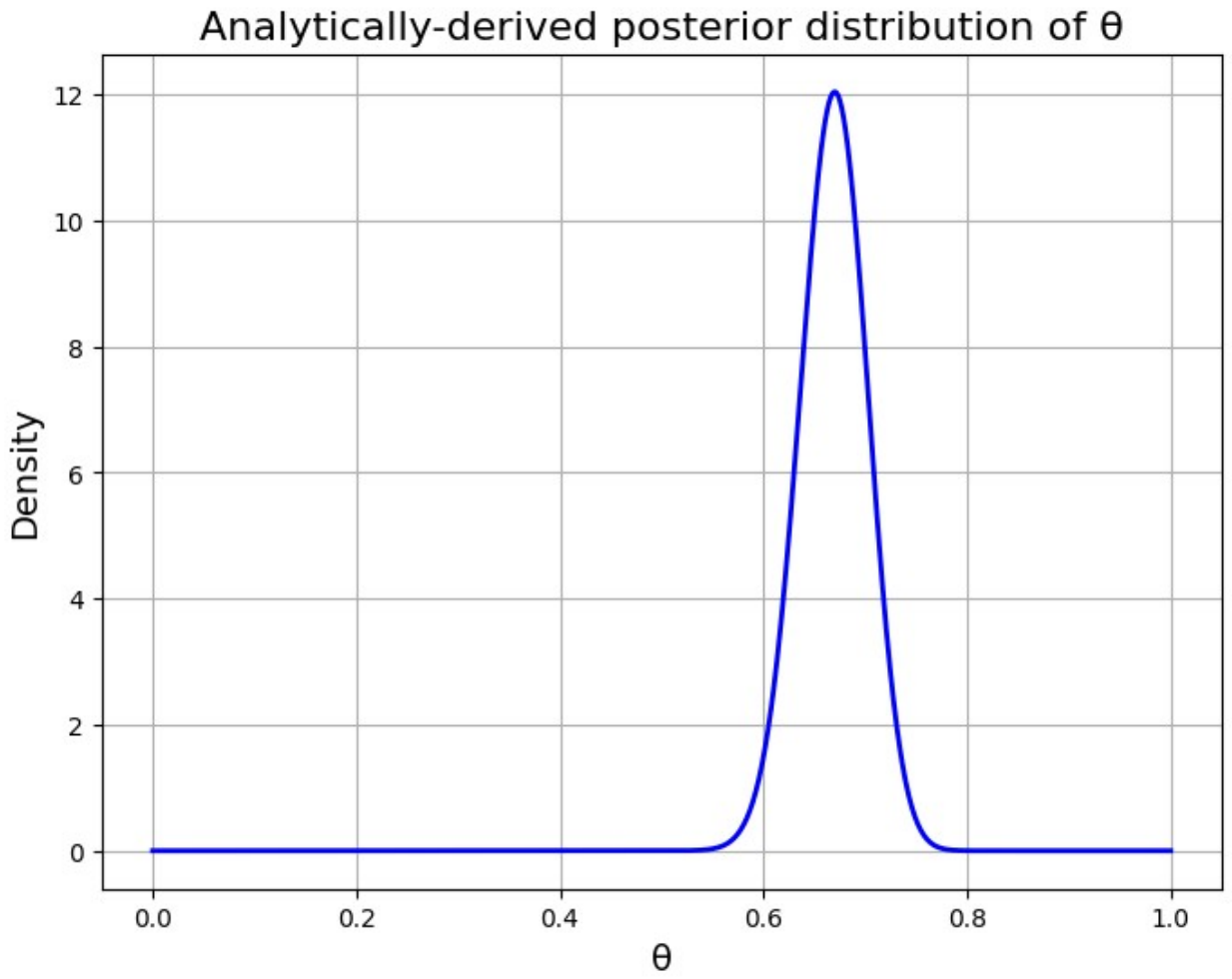
# Assignment 3

*Arpita Chaurasia 220207*

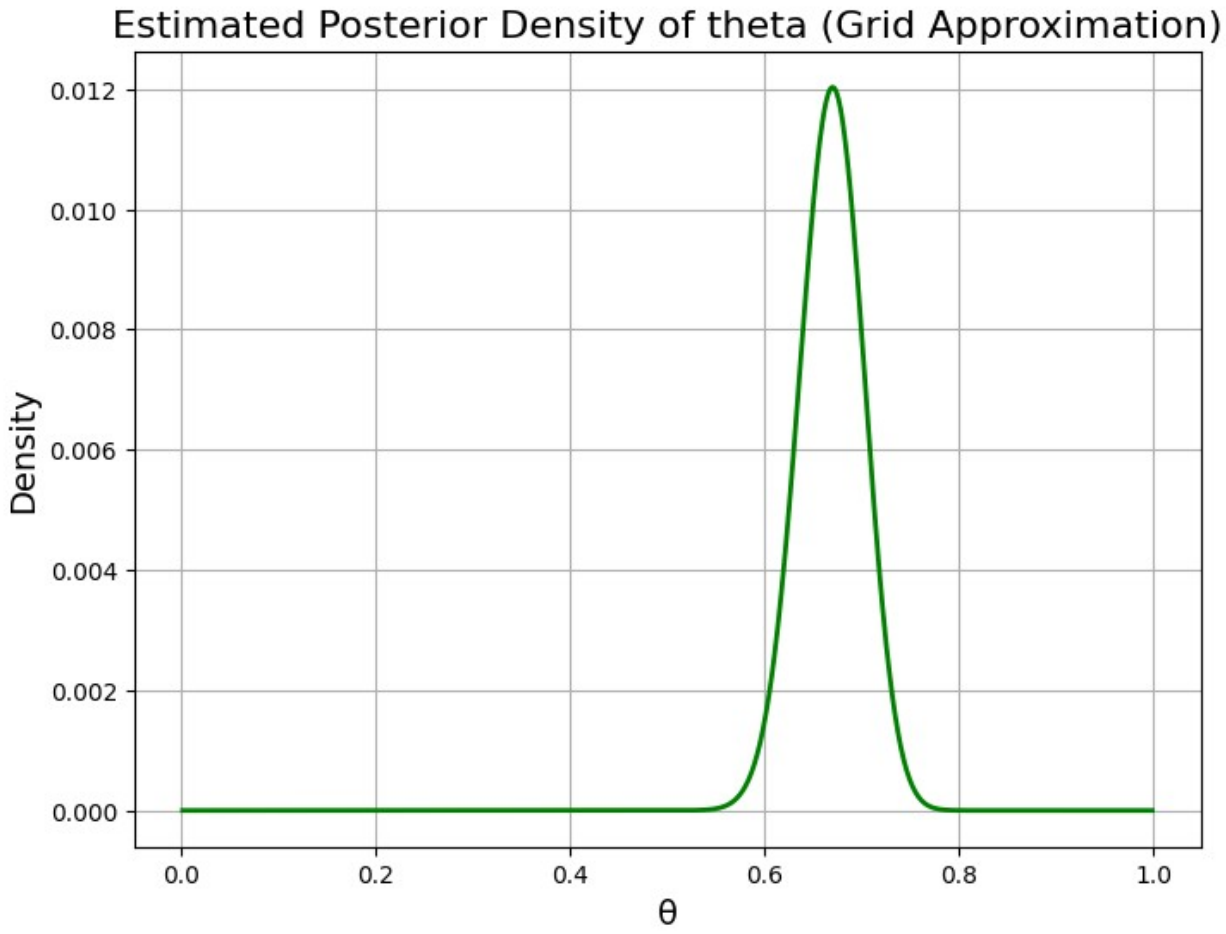
**Part 1: : Estimating the posterior distribution using different computational methods**

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

# 1 Analytically-derived posterior distribution of theta
alpha = 135
beta_param = 67

theta_values = np.linspace(0, 1, 1000)

posterior_dens = beta.pdf(theta_values, alpha, beta_param)

plt.figure(figsize=(8, 6))
plt.plot(theta_values, posterior_dens, color='blue', lw=2)
plt.xlabel('θ', fontsize=14)
plt.ylabel('Density', fontsize=14)
plt.title('Analytically-derived posterior distribution of θ',
fontsize=16)
plt.grid(True)
plt.show()
```




```
# 2 grid approximation
import numpy as np
import matplotlib.pyplot as plt
from scipy.stats import beta

#theta_grid = np.linspace(0, 1, 1000)
theta_grid = np.linspace(0.001, 0.999, 1000)

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

n=20

log_likelihood = theta_data.sum() * np.log(theta_grid) + (n *
len(theta_data) - theta_data.sum()) * np.log(1 - theta_grid)
likelihood = np.exp(log_likelihood - np.max(log_likelihood))  #
subtracting the maximum to avoid overflow
prior = np.ones_like(theta_grid)  # uniform prior Beta(1, 1)

unnormalized_posterior = likelihood * prior
posterior_norm = unnormalized_posterior /
```

```
np.sum(unnormalized_posterior)

plt.figure(figsize=(8, 6))
plt.plot(theta_grid, posterior_norm, color='green', lw=2)
plt.xlabel('θ', fontsize=14)
plt.ylabel('Density', fontsize=14)
plt.title('Estimated Posterior Density of theta (Grid Approximation)',
fontsize=16)
plt.grid(True)
plt.show()
```




```
#3
import numpy as np
import matplotlib.pyplot as plt
from scipy.stats import beta

N = 100000

theta_samples = np.random.beta(1, 1, size=N) # prior = beta(1,1)
```

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

likelihood_samples = np.prod(np.power(theta_samples[:, np.newaxis],
data.sum()) * np.power(1 - theta_samples[:, np.newaxis], 20 *
len(data) - data.sum()), axis=1)

marginal_likelihood = np.mean(likelihood_samples)

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

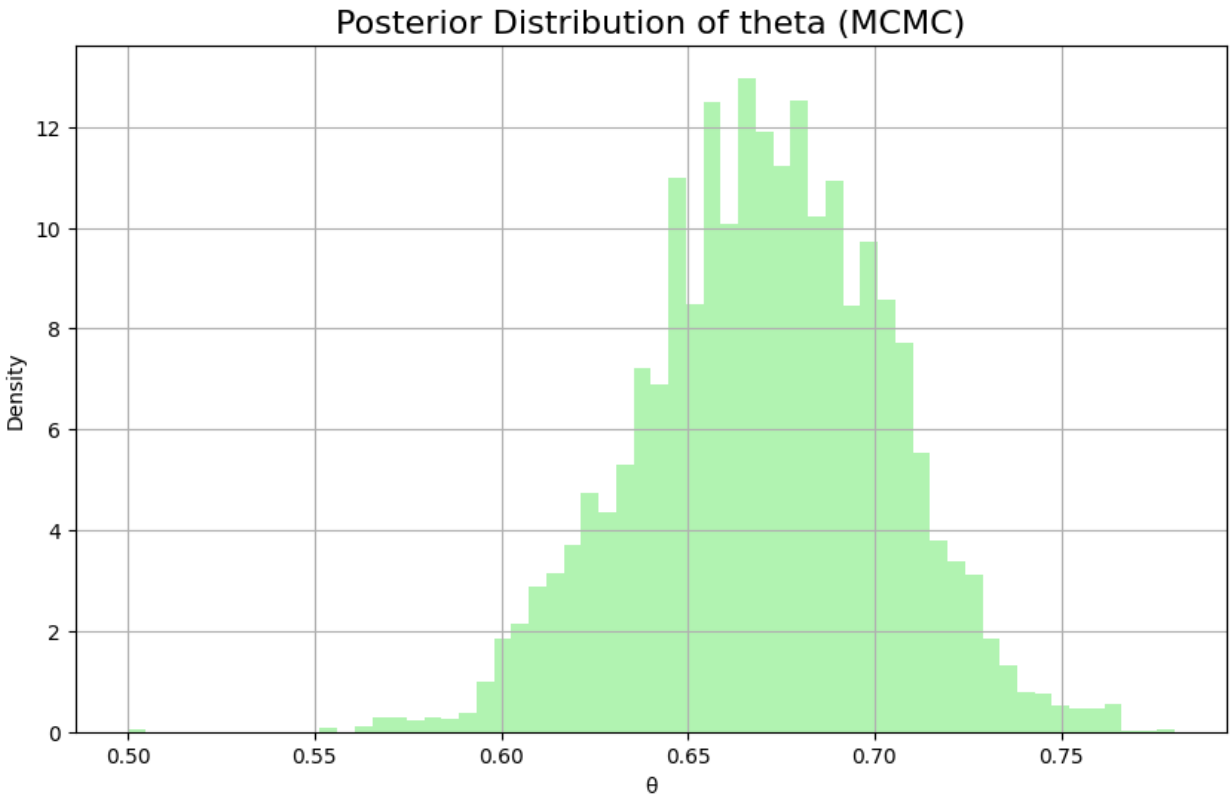
```
Estimated marginal likelihood: 6.937212802688169e-57
```

```
# 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)

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

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

weights = likelihoods * prior / proposal_dens
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:
[0.63926057 0.73011217 0.67921151 ... 0.65308111 0.64467406
0.61514354]
```

```
#5
import numpy as np
import matplotlib.pyplot as plt
```

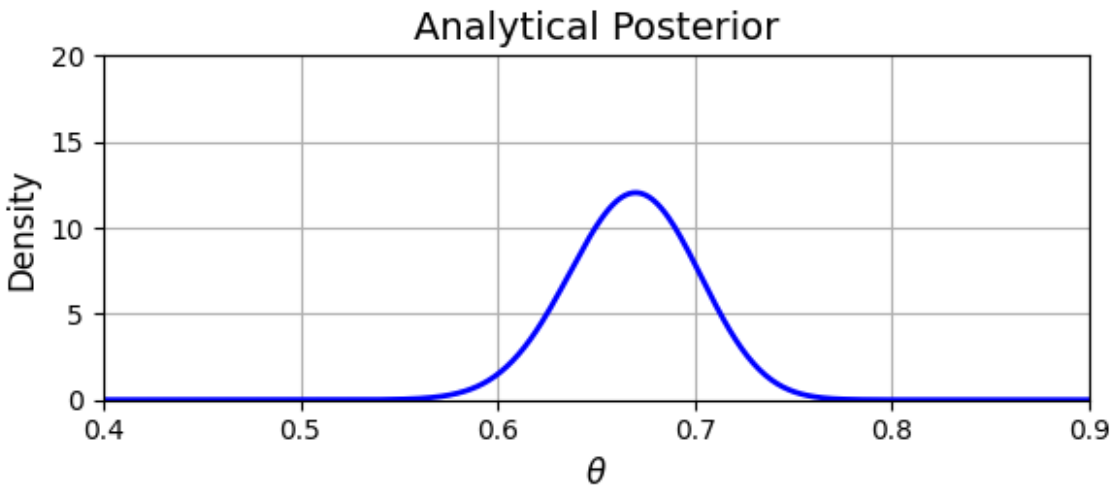
```
theta_data = np.array([10, 15, 15, 14, 14, 14, 13, 11, 12, 16])
n = 20

alpha_prior = 1
beta_prior = 1

def log_posterior(theta, theta_data, n, alpha_prior, beta_prior):
    if theta < 0 or theta > 1:
        return -np.inf
    else:
        likelihood = np.prod(theta**theta_data.sum() * (1-
theta)**(n*len(theta_data)-theta_data.sum()))
        prior = theta**(alpha_prior-1) * (1-theta)**(beta_prior-1)
        return np.log(likelihood * prior)

def metropolis_hastings(log_posterior, theta0, n_samples,
proposal_std=0.1):
    samples = [theta_init]
    current_theta = theta_init

    for _ in range(n_samples):
        proposed_theta = np.random.normal(current_theta, proposal_std)
        log_alpha = log_posterior(proposed_theta, theta_data, n,
alpha_prior, beta_prior) - log_posterior(current_theta, theta_data, n,
alpha_prior, beta_prior)

        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(123)

theta_init = 0.5
n_samples = 10000

samples = metropolis_hastings(log_posterior, theta_init, n_samples)

plt.figure(figsize=(10, 6))
plt.hist(samples, bins=60, density=True, color='lightgreen',
alpha=0.7)
plt.title('Posterior Distribution of theta (MCMC)', fontsize=16)
plt.xlabel('θ')
plt.ylabel('Density')
plt.grid(True)
plt.show()
```

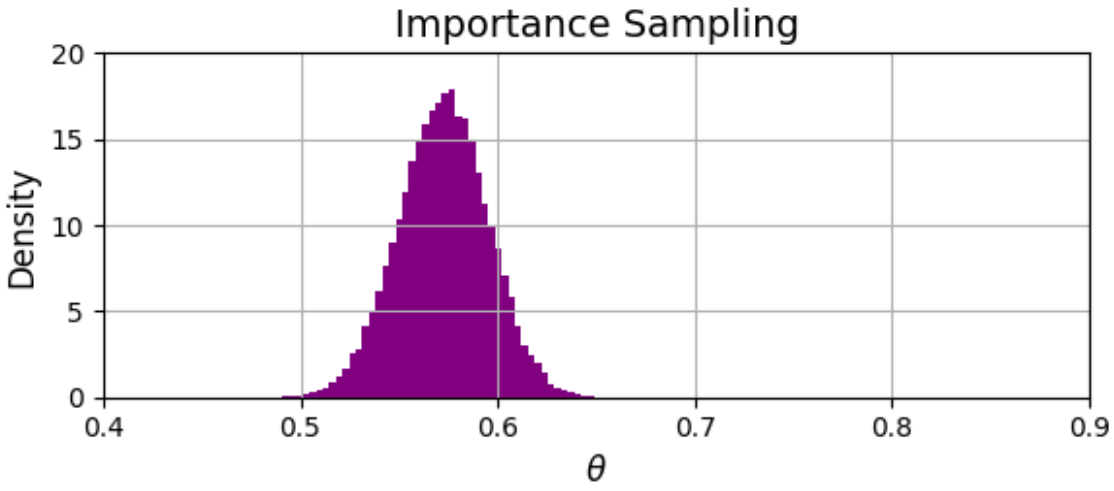


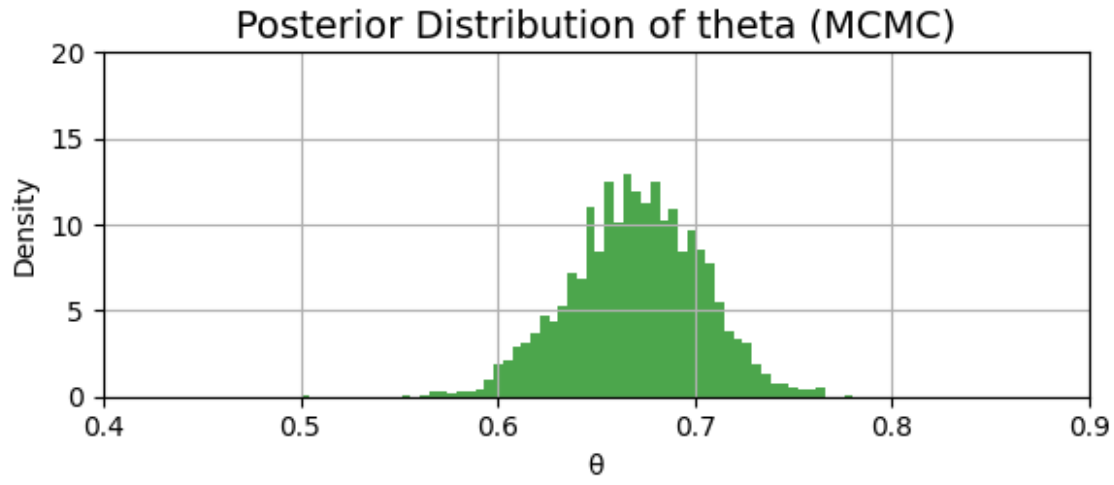

```
import numpy as np
import matplotlib.pyplot as plt
from scipy.stats import beta

# 1 Analytically-derived posterior distribution of theta
alpha = 135
beta_param = 67

theta_values = np.linspace(0, 1, 1000)

posterior_dens = beta.pdf(theta_values, alpha, beta_param)

#importance sampling
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)

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

prior = np.ones_like(theta_proposal)
```

```
proposal_dens = np.random.beta(2, 2, size=N)

weights = likelihoods * prior / proposal_dens
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
#MCMC - metropolis hastings
n = 20

alpha_prior = 1
beta_prior = 1

def log_posterior(theta, theta_data, n, alpha_prior, beta_prior):
    if theta < 0 or theta > 1:
        return -np.inf
    else:
        likelihood = np.prod(theta**theta_data.sum() * (1-
theta)**(n*len(theta_data)-theta_data.sum()))
        prior = theta**(alpha_prior-1) * (1-theta)**(beta_prior-1)
        return np.log(likelihood * prior)

def metropolis_hastings(log_posterior, theta0, n_samples,
proposal_std=0.1):
    samples = [theta_init]
    current_theta = theta_init

    for _ in range(n_samples):
        proposed_theta = np.random.normal(current_theta, proposal_std)
        log_alpha = log_posterior(proposed_theta, theta_data, n,
alpha_prior, beta_prior) - log_posterior(current_theta, theta_data, n,
alpha_prior, beta_prior)

        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(123)

theta_init = 0.5
n_samples = 10000

samples = metropolis_hastings(log_posterior, theta_init, n_samples)

# Plotting all distributions for comparison
plt.figure(figsize=(6,8))
```

```
# Analytical posterior
plt.subplot(3, 1, 1)
plt.plot(theta_values, posterior_dens, color='blue', lw=2)
plt.xlim([0.4, 0.9])
plt.ylim([0, 20])
plt.title('Analytical Posterior', fontsize=14)
plt.xlabel(r'$\theta$', fontsize=12)
plt.ylabel('Density', fontsize=12)
plt.grid(True)

# Importance sampling
plt.subplot(3, 1, 2)
plt.hist(posterior_samples_importance, bins=60, density=True,
color='purple')
plt.xlim([0.4, 0.9])
plt.ylim([0, 20])
plt.title('Importance Sampling', fontsize=14)
plt.xlabel(r'$\theta$', fontsize=12)
plt.ylabel('Density', fontsize=12)
plt.grid(True)

# MCMC
plt.subplot(3, 1, 3)
plt.hist(samples, bins=60, density=True, color='green', alpha=0.7)
plt.xlim([0.4, 0.9])
plt.ylim([0, 20])
plt.title('Posterior Distribution of theta (MCMC)', fontsize=14)
plt.xlabel('θ')
plt.ylabel('Density')
plt.grid(True)
plt.tight_layout()
plt.show()
```




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

```
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**

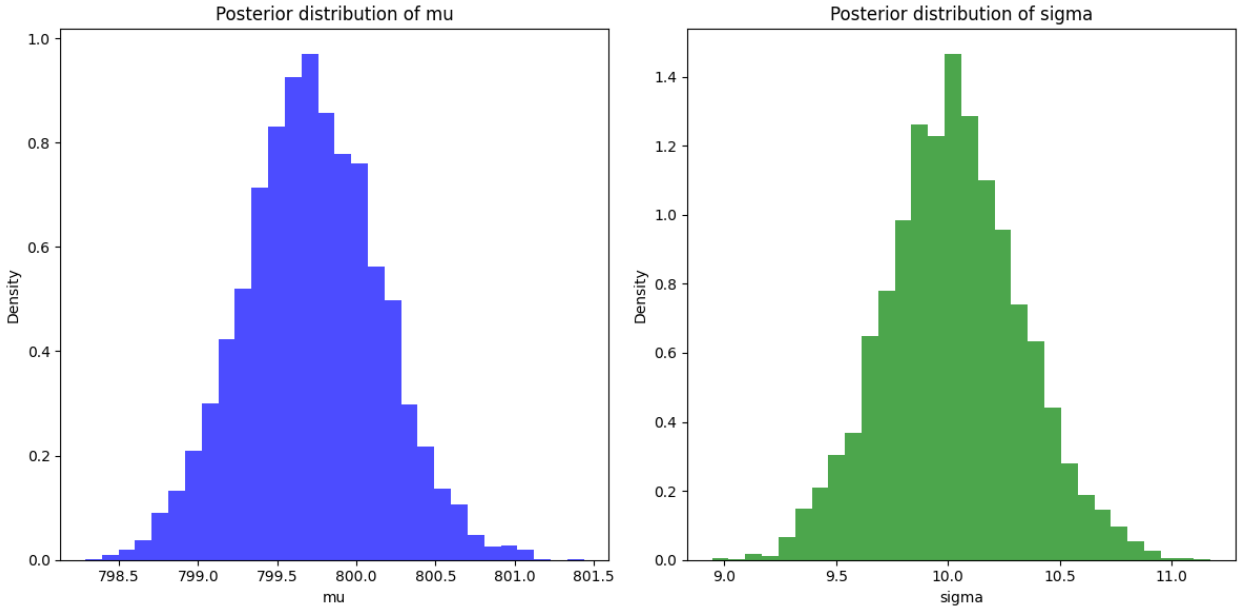
Part 3.1

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

# Data generation
np.random.seed(0)
true_mu = 800
true_var = 100
y = np.random.normal(loc=true_mu, scale=np.sqrt(true_var), size=500)

# 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
    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(0, 1, size=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 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
        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

    posteriors = pd.DataFrame({'mu': mu_chain[nburn:], 'sigma':
sigma_chain[nburn:], 'sample_id': np.arange(nsamp - nburn)})
    return posteriors

# Parameters for Exercise 3.1
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)

# Plot posterior distributions
import matplotlib.pyplot as plt

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

plt.subplot(1, 2, 1)
plt.hist(df_posterior['mu'], bins=30, density=True, alpha=0.7,
color='blue')
plt.title('Posterior distribution of mu')
plt.xlabel('mu')
plt.ylabel('Density')

plt.subplot(1, 2, 2)
plt.hist(df_posterior['sigma'], bins=30, density=True, alpha=0.7,
color='green')
plt.title('Posterior distribution of sigma')
plt.xlabel('sigma')
plt.ylabel('Density')

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

```
<ipython-input-3-8982cc51fb3c>:53: RuntimeWarning: overflow
encountered in exp
  accept_prob = min(1, np.exp(current_V + current_T - proposed_V -
proposed_T))
```




Part 3.2 and 3.3

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

# Data generation
np.random.seed(0)
true_mu = 800
true_var = 100
y = np.random.normal(loc=true_mu, scale=np.sqrt(true_var), size=500)

# 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
```

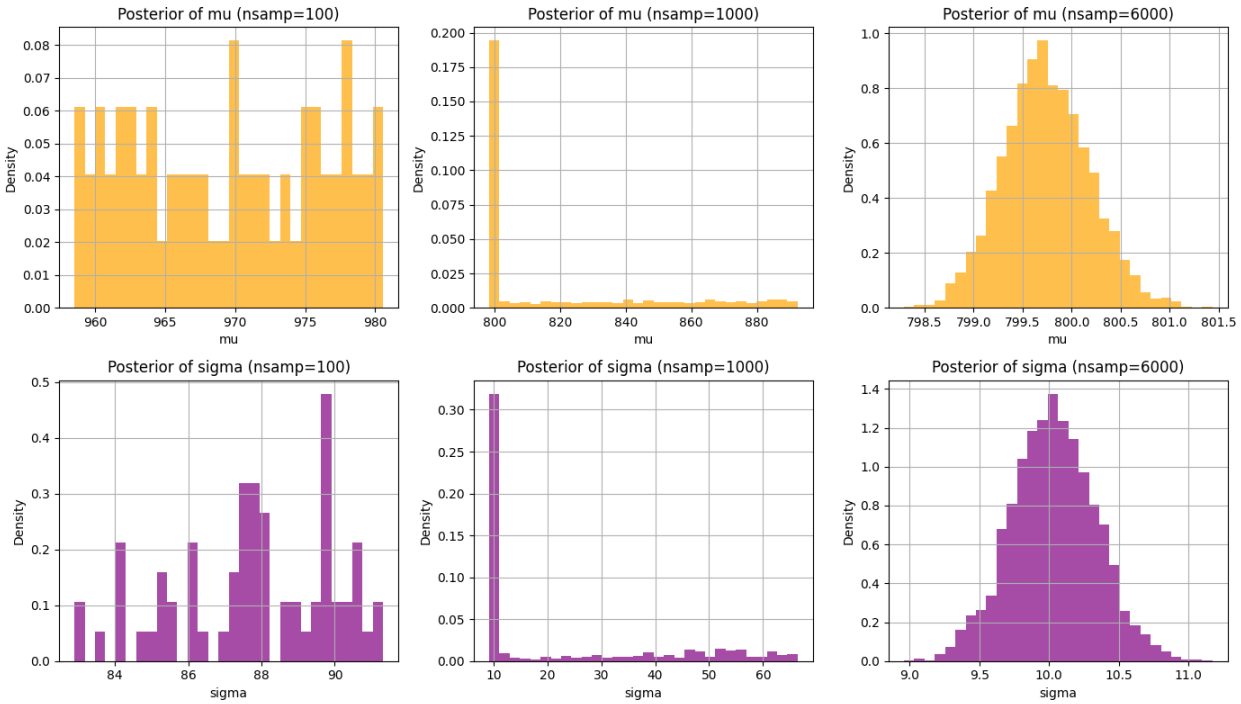
```
    # Initialization of Markov chain
    mu_chain[0] = initial_q[0]
    sigma_chain[0] = initial_q[1]

    # Evolution of Markov chain
    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(0, 1, size=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 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
        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

    posteriors = pd.DataFrame({'mu': mu_chain[nburn:], 'sigma':
sigma_chain[nburn:], 'sample_id': np.arange(nsamp - nburn)})
    return posteriors

# Exercise 3.2: Posteriors sensitivity to total number of samples
def evaluate_posterior_sensitivity(nsamp_values):
    results = []
    for nsamp in nsamp_values:
```

```
        nburn = nsamp // 3
        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(df_posterior)
    return results

# Exercise 3.3: Posteriors sensitivity to step-size parameter
def evaluate_step_sensitivity(step_values):
    results = []
    for step_val in step_values:
        df_posterior = HMC(y=y, n=len(y), m=1000, s=100, a=10, b=2,
step=step_val, L=L, initial_q=initial_q, nsamp=nsamp, nburn=nburn)
        results.append(df_posterior)
    return results

# Evaluate for different nsamp values (Exercise 3.2)
nsamp_values = [100, 1000, 6000]
results_nsamp = evaluate_posterior_sensitivity(nsamp_values)

# Plot posteriors for different nsamp values (Exercise 3.2)
print("Part 3.2")
plt.figure(figsize=(14, 8))

for i, nsamp_val in enumerate(nsamp_values):
    plt.subplot(2, 3, i + 1)
    plt.hist(results_nsamp[i]['mu'], bins=30, density=True, alpha=0.7,
color='orange')
    plt.title(f'Posterior of mu (nsamp={nsamp_val})')
    plt.xlabel('mu')
    plt.ylabel('Density')
    plt.grid(True)

    plt.subplot(2, 3, i + 4)
    plt.hist(results_nsamp[i]['sigma'], bins=30, density=True,
alpha=0.7, color='purple')
    plt.title(f'Posterior of sigma (nsamp={nsamp_val})')
    plt.xlabel('sigma')
    plt.ylabel('Density')
    plt.grid(True)

plt.tight_layout()
plt.show()

# Evaluate for different step values (Exercise 3.3)
step_values = [0.001, 0.005, 0.02]
results_step = evaluate_step_sensitivity(step_values)

# Plot posteriors for different step values (Exercise 3.3)
print(" ")
print("Part 3.3")
```

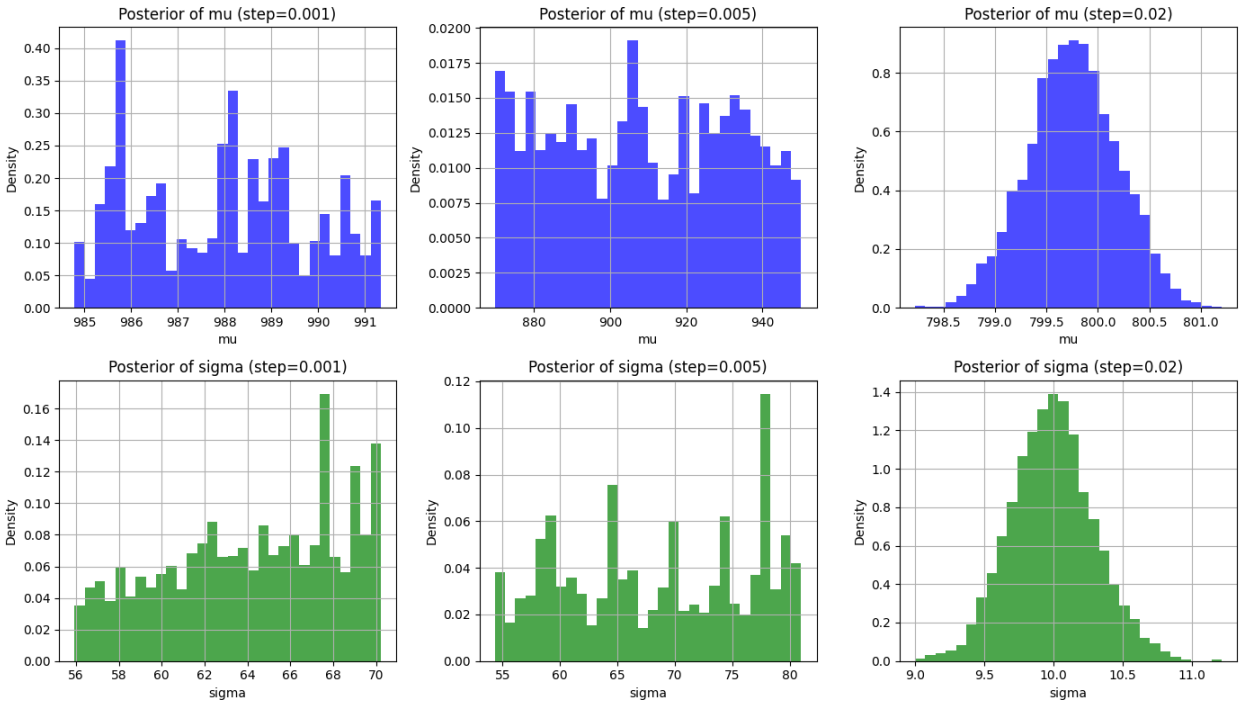
```
plt.figure(figsize=(14, 8))

for i, step_val in enumerate(step_values):
    plt.subplot(2, 3, i + 1)
    plt.hist(results_step[i]['mu'], bins=30, density=True, alpha=0.7,
color='blue')
    plt.title(f'Posterior of mu (step={step_val})')
    plt.xlabel('mu')
    plt.ylabel('Density')
    plt.grid(True)

    plt.subplot(2, 3, i + 4)
    plt.hist(results_step[i]['sigma'], bins=30, density=True,
alpha=0.7, color='green')
    plt.title(f'Posterior of sigma (step={step_val})')
    plt.xlabel('sigma')
    plt.ylabel('Density')
    plt.grid(True)

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

```
<ipython-input-41-5beaf81d3156>:54: RuntimeWarning: overflow
encountered in exp
  accept_prob = min(1, np.exp(current_V + current_T - proposed_V -
proposed_T))
```

Part 3.2



Part 3.3

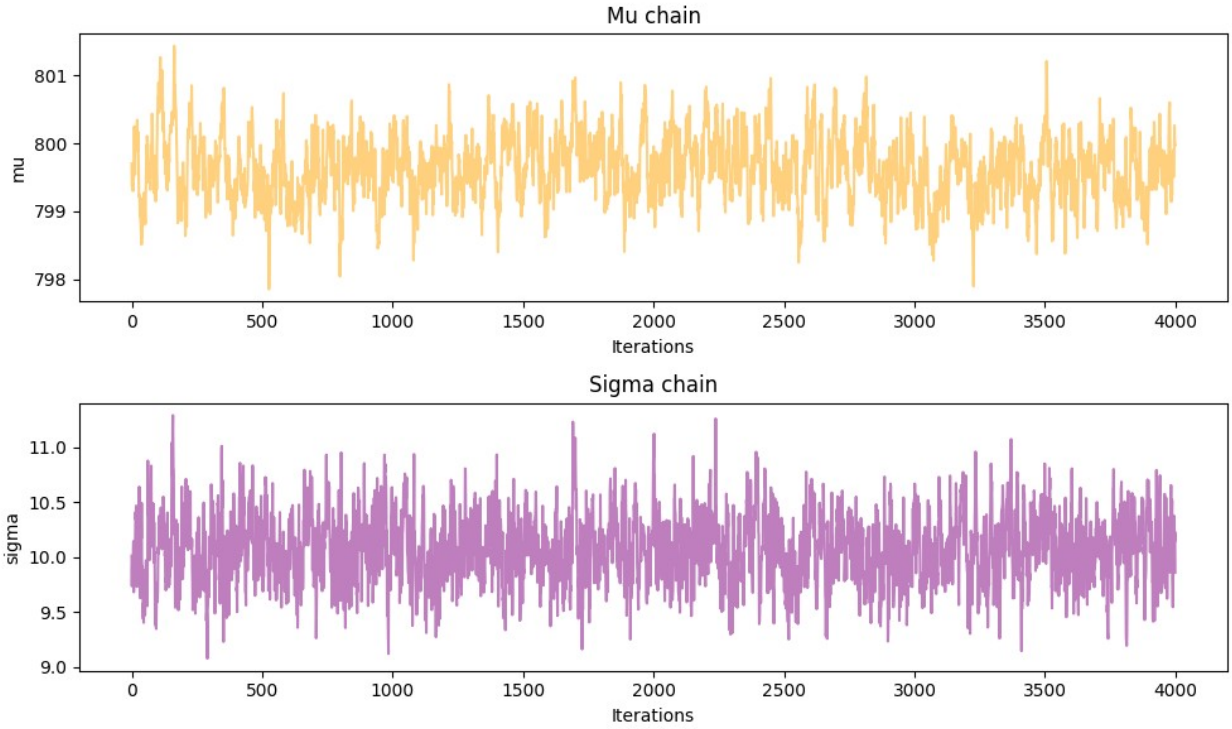



Part 3.4

```
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=(10, 6))

plt.subplot(2, 1, 1)
plt.plot(mu_chain, color='orange', alpha=0.5)
plt.title('Mu chain')
plt.xlabel('Iterations')
plt.ylabel('mu')

plt.subplot(2, 1, 2)
plt.plot(sigma_chain, color='purple', alpha=0.5)
plt.title('Sigma chain')
plt.xlabel('Iterations')
plt.ylabel('sigma')

plt.tight_layout()
plt.show()

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




```
Number of rejections: 3
```

The values of mu and sigma both keep oscillating between [798,801] and [9, 11] respectively and do not settle at some point and do not have any increasing or decreasing trend.

Part 3.5

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


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

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])

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

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]])
        p = np.random.normal(0, 1, size=len(q))

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

        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

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

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)

plt.figure(figsize=(12, 8))
colors = ['lightblue', 'red', 'lightgreen', 'purple', 'yellow']
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.grid(True)
plt.title('Posterior Distribution of μ for Different Priors')
plt.xlabel('μ')
plt.ylabel('Density')
plt.legend()
plt.tight_layout()
plt.show()
```