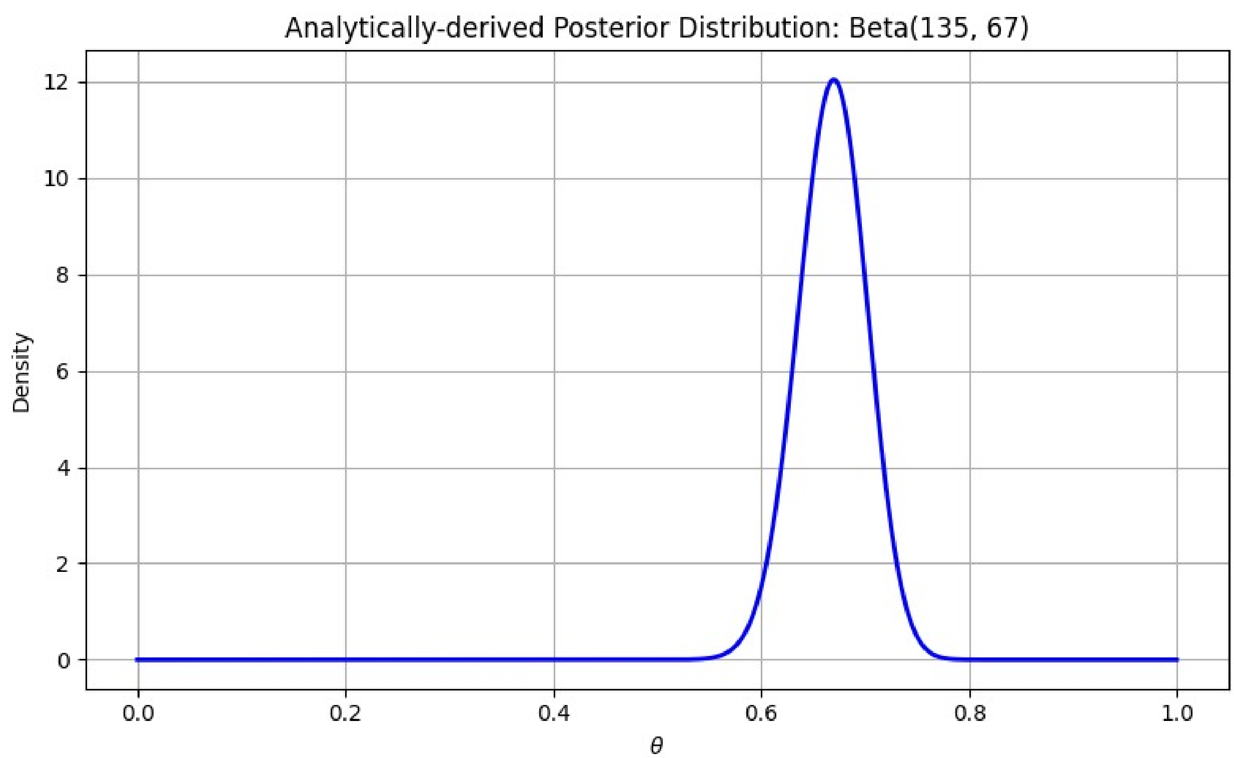
Part 1: Estimating the posterior distribution using different computational methods

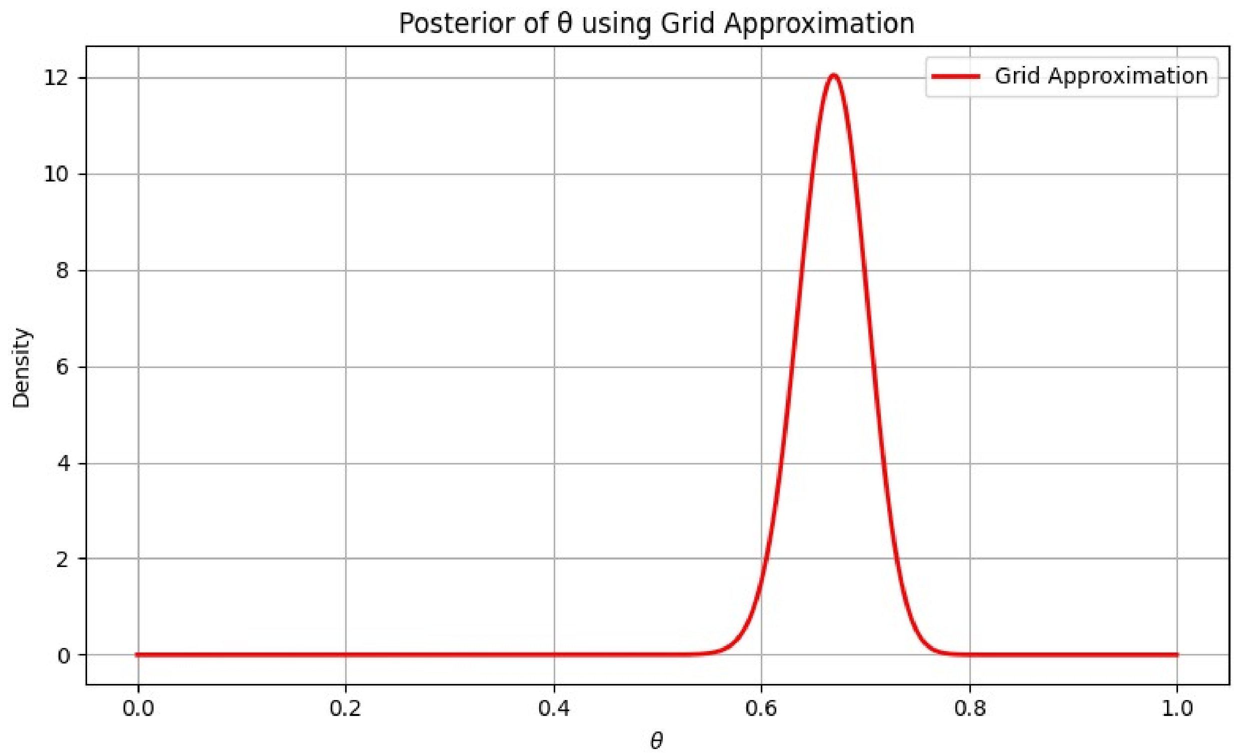
1. Analytically-derived posterior distribution of θ

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

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

posterior_pdf = beta.pdf(theta_vals, a=135, b=67)

plt.figure(figsize=(8, 5))
plt.plot(theta_vals, posterior_pdf, color='blue', lw=2)
plt.title('Analytically-derived Posterior Distribution: Beta(135,
67)')
plt.xlabel(r'$\theta$')
plt.ylabel('Density')
plt.grid(True)
plt.tight_layout()
plt.show()
```



2. Grid Approximation

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

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

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

# Prior
prior = np.ones_like(theta_vals)

# Likelihood
likelihood = np.ones_like(theta_vals)
for y in data:
    likelihood *= binom.pmf(y, n, theta_vals)

# Unnormalized posterior
unnormalized_posterior = likelihood * prior

# Normalize posterior
posterior = unnormalized_posterior / np.trapz(unnormalized_posterior,
theta_vals)

# Plot
plt.figure(figsize=(8, 5))
plt.plot(theta_vals, posterior, color='red', label='Grid
Approximation', lw=2)
plt.title('Posterior of θ using Grid Approximation')
plt.xlabel(r'$\theta$')
plt.ylabel('Density')
plt.grid(True)
plt.legend()
plt.tight_layout()
plt.show()
```

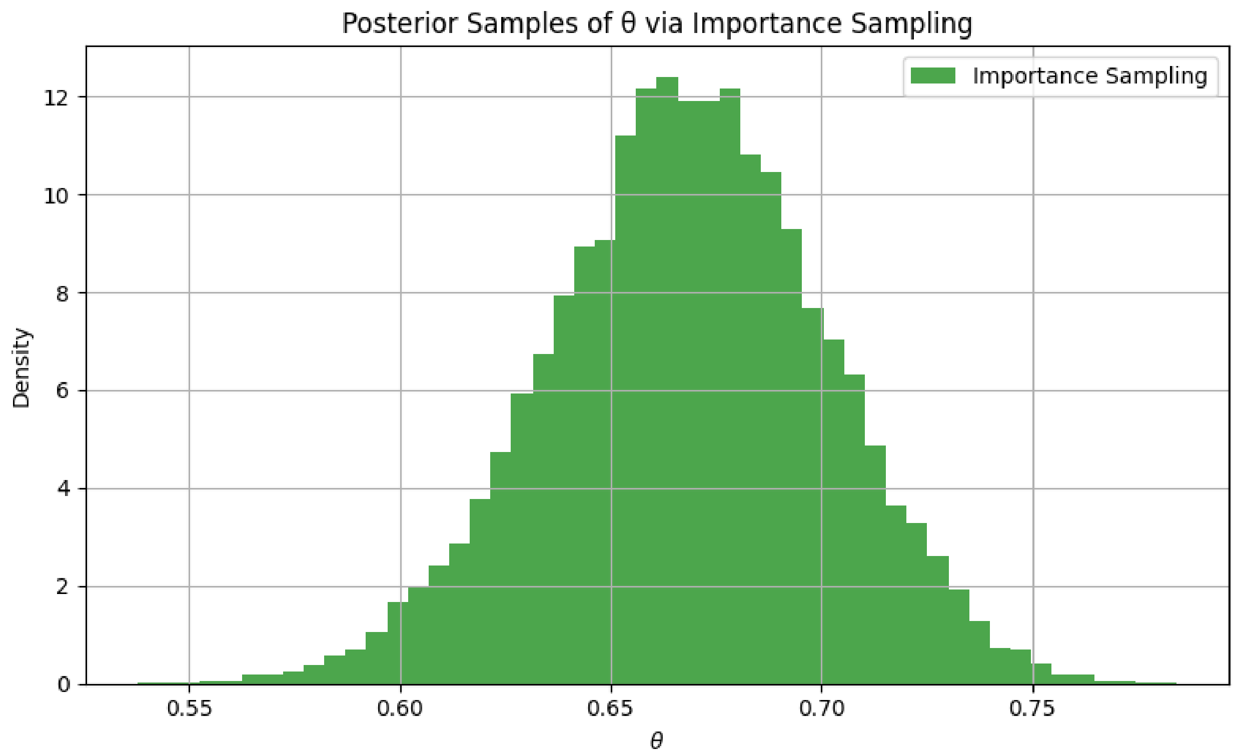


3. Monte Carlo Integration : Marginal Likelihood

```
import numpy as np
from scipy.stats import binom, beta

# Data
data = np.array([10, 15, 15, 14, 14, 14, 13, 11, 12, 16])
n = 20
N = 100000  # number of prior samples

# Drawn theta samples from prior Beta(1,1) (uniform)
theta_samples = np.random.beta(a=1, b=1, size=N)

# Computed likelihood for each theta
likelihoods = np.ones(N)
for y in data:
    likelihoods *= binom.pmf(y, n, theta_samples)

# Estimated marginal likelihood
marginal_likelihood = np.mean(likelihoods)

print("Estimated marginal likelihood (evidence):",
marginal_likelihood)

Estimated marginal likelihood (evidence): 1.3856194547650128e-10
```

4. Importance Sampling

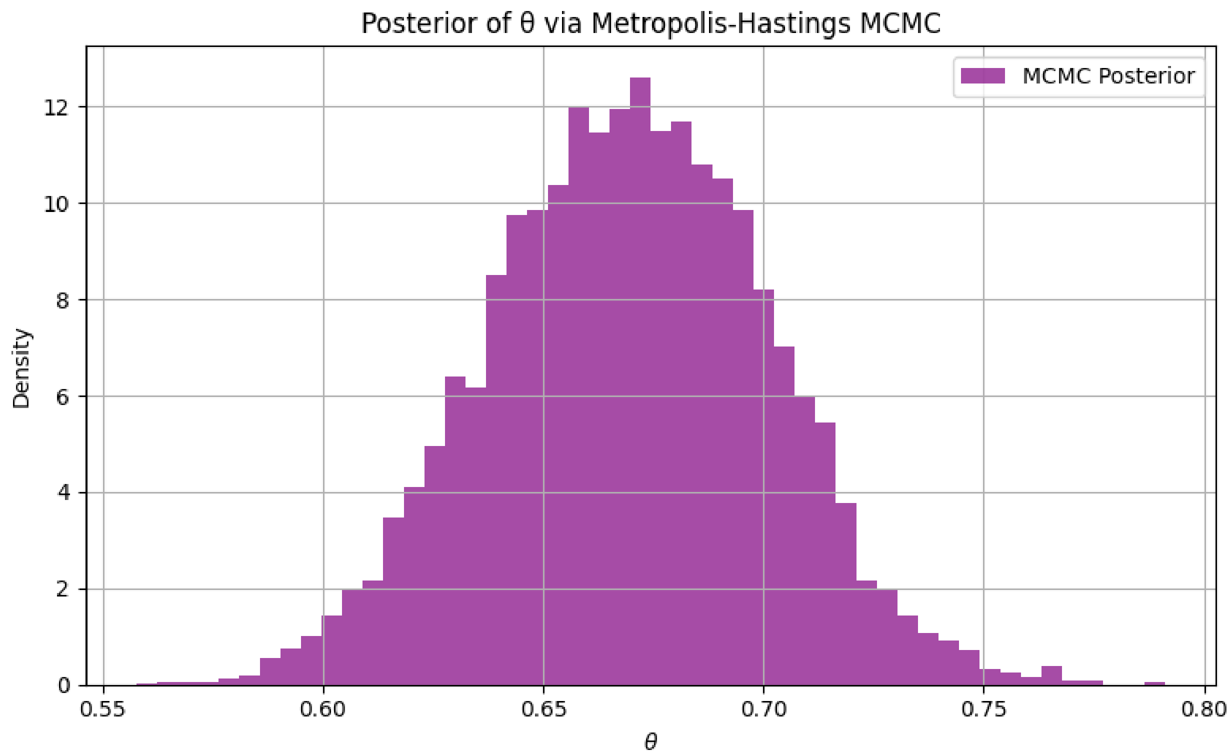
```
import numpy as np
import pandas as pd
from scipy.stats import binom, beta
import matplotlib.pyplot as plt


data = np.array([10, 15, 15, 14, 14, 14, 13, 11, 12, 16])
n = 20
N = 100000

# Step 1: Draw samples from proposal distribution q(θ)
proposal_theta = np.random.beta(a=5, b=5, size=N)

# Step 2: Compute prior p(θ), likelihood L(θ|y), proposal q(θ)
prior = beta.pdf(proposal_theta, a=1, b=1)
proposal_density = beta.pdf(proposal_theta, a=5, b=5)

# Likelihood of entire dataset for each θ
likelihood = np.ones(N)
for y in data:
    likelihood *= binom.pmf(y, n, proposal_theta)

# Step 3: Compute importance weights
weights = likelihood * prior / proposal_density
weights /= np.sum(weights)

# Step 4: Resample from θ using the weights
posterior_samples = np.random.choice(proposal_theta, size=N//4,
replace=True, p=weights)


plt.figure(figsize=(8, 5))
plt.hist(posterior_samples, bins=50, density=True, alpha=0.7,
color='green', label='Importance Sampling')
plt.title("Posterior Samples of θ via Importance Sampling")
plt.xlabel(r'$\theta$')
plt.ylabel("Density")
plt.grid(True)
plt.legend()
plt.tight_layout()
plt.show()
```



5. Markov Chain Monte Carlo method

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

# Data
data = np.array([10, 15, 15, 14, 14, 14, 13, 11, 12, 16])
n_trials = 20
N = 10000
burn_in = 2000
proposal_sd = 0.02

# Define log posterior
def log_posterior(theta, data):
    if theta <= 0 or theta >= 1:
        return -np.inf
    log_likelihood = np.sum(binom.logpmf(data, n_trials, theta))
    log_prior = 0
    return log_likelihood + log_prior

# MCMC using Metropolis-Hastings
samples = []
theta_current = 0.5

for i in range(N):
```

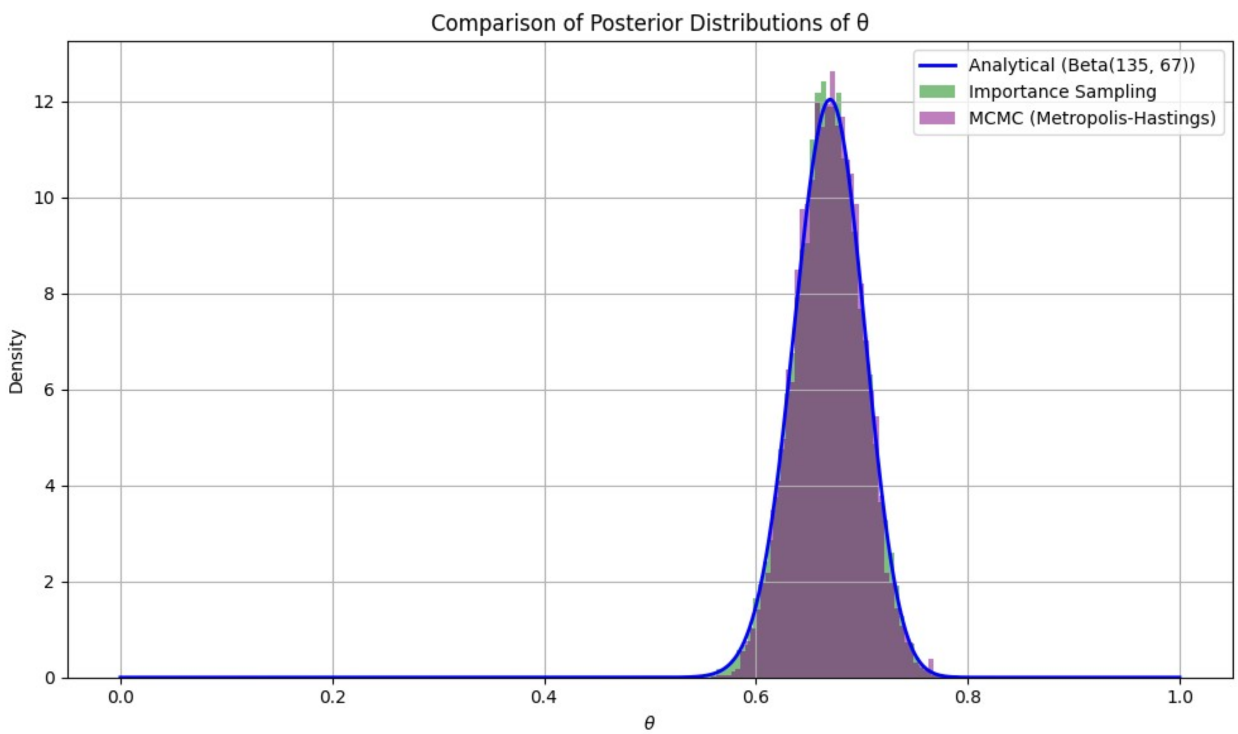
```
    theta_proposed = np.random.normal(theta_current, proposal_sd)


    if theta_proposed <= 0 or theta_proposed >= 1:
        samples.append(theta_current)
        continue

    # Compute acceptance probability
    log_p_current = log_posterior(theta_current, data)
    log_p_proposed = log_posterior(theta_proposed, data)
    acceptance_prob = np.exp(log_p_proposed - log_p_current)

    # Accept/reject
    if np.random.rand() < acceptance_prob:
        theta_current = theta_proposed

    samples.append(theta_current)

# Discard burn-in samples
posterior_samples_mcmc = np.array(samples[burn_in:])


plt.figure(figsize=(8, 5))
plt.hist(posterior_samples_mcmc, bins=50, density=True, alpha=0.7,
color='purple', label='MCMC Posterior')
plt.title("Posterior of θ via Metropolis-Hastings MCMC")
plt.xlabel(r'$\theta$')
plt.ylabel("Density")
plt.grid(True)
plt.legend()
plt.tight_layout()
plt.show()
```



6. Comparison

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

# grid for analytical posterior
theta_vals = np.linspace(0, 1, 1000)
analytical_density = beta.pdf(theta_vals, a=135, b=67)


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

# Analytical
plt.plot(theta_vals, analytical_density, color='blue', lw=2,
label='Analytical (Beta(135, 67))')

# Importance Sampling
plt.hist(posterior_samples, bins=50, density=True, alpha=0.5,
color='green', label='Importance Sampling')

# MCMC
plt.hist(posterior_samples_mcmc, bins=50, density=True, alpha=0.5,
color='purple', label='MCMC (Metropolis-Hastings)')

# Plot
plt.title("Comparison of Posterior Distributions of θ")
```

```
plt.xlabel(r"$\theta$")
plt.ylabel("Density")
plt.legend()
plt.grid(True)
plt.tight_layout()
plt.show()
```



Part 2: Writing your own sampler for Bayesian inference

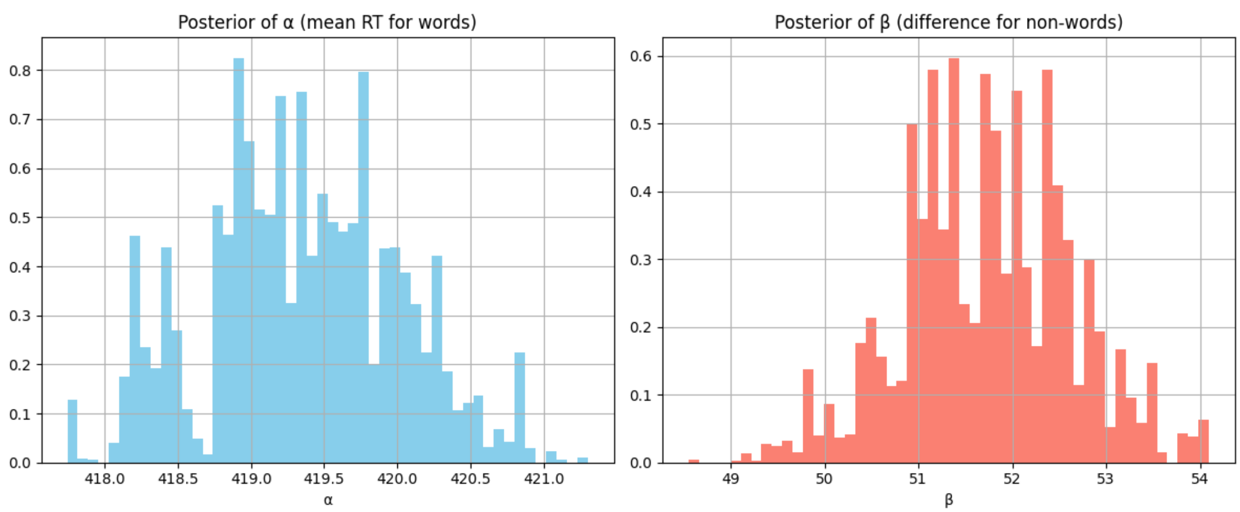
2.5.1

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

# Step 1: Load and prepare data
url =
"https://raw.githubusercontent.com/yadavhimanshu059/CGS698C/main/notes
/Data/word-recognition-times.csv"
df = pd.read_csv(url)
df['type'] = df['type'].map({'word': 0, 'non-word': 1})  # Encode:
word=0, non-word=1
```

```
x = df['type'].values
y = df['RT'].values
sigma = 30  # fixed

# Step 2: Define log posterior
def log_posterior(alpha, beta):
    if beta < 0:
        return -np.inf  # enforce truncation: beta > 0
    mu = alpha + beta * x
    log_likelihood = np.sum(norm.logpdf(y, loc=mu, scale=sigma))
    log_prior_alpha = norm.logpdf(alpha, loc=400, scale=50)
    log_prior_beta = norm.logpdf(beta, loc=0, scale=50) + np.log(2)  #
truncated normal correction
    return log_likelihood + log_prior_alpha + log_prior_beta

# Step 3: Initialize MCMC
n_samples = 10000
alpha_chain = np.zeros(n_samples)
beta_chain = np.zeros(n_samples)
alpha_chain[0] = 400
beta_chain[0] = 10
proposal_sd = 5

# Step 4: Run Metropolis sampler
for i in range(1, n_samples):
    # Propose new values
    alpha_prop = np.random.normal(alpha_chain[i-1], proposal_sd)
    beta_prop = np.random.normal(beta_chain[i-1], proposal_sd)

    # Calculate log posterior
    log_p_curr = log_posterior(alpha_chain[i-1], beta_chain[i-1])
    log_p_prop = log_posterior(alpha_prop, beta_prop)

    # Accept/reject
    accept_prob = np.exp(log_p_prop - log_p_curr)
    if np.random.rand() < accept_prob:
        alpha_chain[i] = alpha_prop
        beta_chain[i] = beta_prop
    else:
        alpha_chain[i] = alpha_chain[i-1]
        beta_chain[i] = beta_chain[i-1]
# Discard burn-in
burn = 2000
alpha_post = alpha_chain[burn:]
beta_post = beta_chain[burn:]

# Plot
plt.figure(figsize=(12, 5))

plt.subplot(1, 2, 1)
```

```
plt.hist(alpha_post, bins=50, density=True, color='skyblue')
plt.title("Posterior of α (mean RT for words)")
plt.xlabel("α")
plt.grid(True)

plt.subplot(1, 2, 2)
plt.hist(beta_post, bins=50, density=True, color='salmon')
plt.title("Posterior of β (difference for non-words)")
plt.xlabel("β")
plt.grid(True)

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

```
/var/folders/jj/nt7bpvds7tzdfs30sl0fh4kh0000gn/T/
ipykernel_41231/4287753762.py:43: RuntimeWarning: overflow encountered
in exp
  accept_prob = np.exp(log_p_prop - log_p_curr)
```



2.5.2

```
# Compute 95% credible intervals using quantiles
alpha_ci = np.quantile(alpha_post, [0.025, 0.975])
beta_ci = np.quantile(beta_post, [0.025, 0.975])

# Also compute posterior means
alpha_mean = np.mean(alpha_post)
beta_mean = np.mean(beta_post)

# Print results
print("Posterior mean of α:", round(alpha_mean, 2))
print("95% credible interval for α:", alpha_ci)
```

```
print("Posterior mean of β:", round(beta_mean, 2))
print("95% credible interval for β:", beta_ci)
```

```
Posterior mean of α: 419.35
95% credible interval for α: [418.17140585 420.70146415]
Posterior mean of β: 51.74
95% credible interval for β: [49.86630279 53.53983288]
```

Part 3: Hamiltonian Monte Carlo sampler

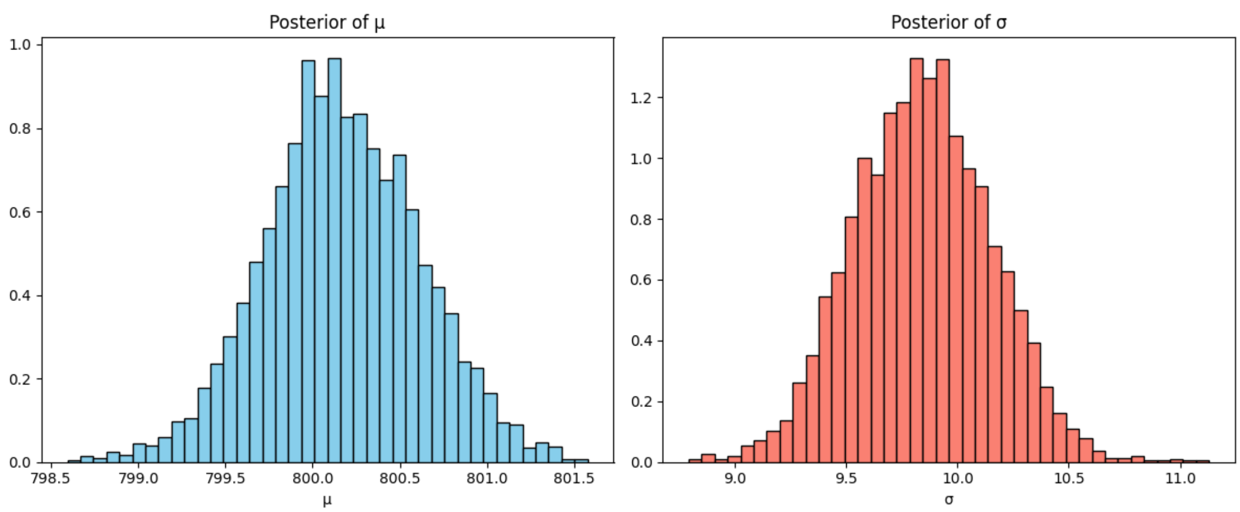
Exercise 3.1

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

# Step 1: Simulate the data
np.random.seed(42)
true_mu = 800
true_sigma = 10
y = np.random.normal(loc=true_mu, scale=true_sigma, size=500)

# Step 2: Define gradient and potential energy 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])

def potential_energy(mu, sigma, y, n, m, s, a, b):
    log_likelihood = np.sum(-0.5 * np.log(2 * np.pi * sigma**2) - ((y
- mu)**2) / (2 * sigma**2))
    log_prior_mu = -0.5 * ((mu - m)**2) / (s**2) - np.log(s *
np.sqrt(2 * np.pi))
    log_prior_sigma = -0.5 * ((sigma - a)**2) / (b**2) - np.log(b *
np.sqrt(2 * np.pi))
    return -1 * (log_likelihood + log_prior_mu + log_prior_sigma)

# Step 3: Define 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)

    mu_chain[0], sigma_chain[0] = initial_q

    i = 0
    while i < nsamp - 1:
        q = np.array([mu_chain[i], sigma_chain[i]])
        p = np.random.normal(0, 1, size=2)
```

```
        current_q = q.copy()
        current_p = p.copy()

        current_U = potential_energy(*current_q, y, n, m, s, a, b)
        current_K = np.sum(current_p**2) / 2

        # Leapfrog steps
        p = p - 0.5 * step * gradient(*q, y, n, m, s, a, b)
        for _ in range(L):
            q = q + step * p
            p = p - step * gradient(*q, y, n, m, s, a, b)
        p = p + 0.5 * step * gradient(*q, y, n, m, s, a, b)
        p = -p  # Reverse momentum for symmetry

        proposed_U = potential_energy(*q, y, n, m, s, a, b)
        proposed_K = np.sum(p**2) / 2

        acceptance_prob = np.exp(current_U + current_K - proposed_U -
proposed_K)
        if np.random.rand() < acceptance_prob and q[1] > 0:
            mu_chain[i + 1], sigma_chain[i + 1] = q
        else:
            mu_chain[i + 1], sigma_chain[i + 1] = mu_chain[i],
sigma_chain[i]
        i += 1

    # Drop burn-in samples
    return mu_chain[nburn:], sigma_chain[nburn:]

# Step 4: Run HMC with specified parameters
posterior_mu, posterior_sigma = HMC(
    y=y,
    n=len(y),
    m=1000,
    s=20,
    a=10,
    b=2,
    step=0.02,
    L=12,
    initial_q=[1000, 11],
    nsamp=6000,
    nburn=2000
)

# Step 5: Plot the posterior distributions
plt.figure(figsize=(12, 5))

plt.subplot(1, 2, 1)
plt.hist(posterior_mu, bins=40, color='skyblue', edgecolor='black',
```

```
density=True)
plt.title("Posterior of μ")
plt.xlabel("μ")

plt.subplot(1, 2, 2)
plt.hist(posterior_sigma, bins=40, color='salmon', edgecolor='black',
density=True)
plt.title("Posterior of σ")
plt.xlabel("σ")

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

```
/var/folders/jj/nt7bpvds7tzdfs30sl0fh4kh0000gn/T/
ipykernel_41231/3426198734.py:51: RuntimeWarning: overflow encountered
in exp
  acceptance_prob = np.exp(current_U + current_K - proposed_U -
proposed_K)
```



Exercise 3.2

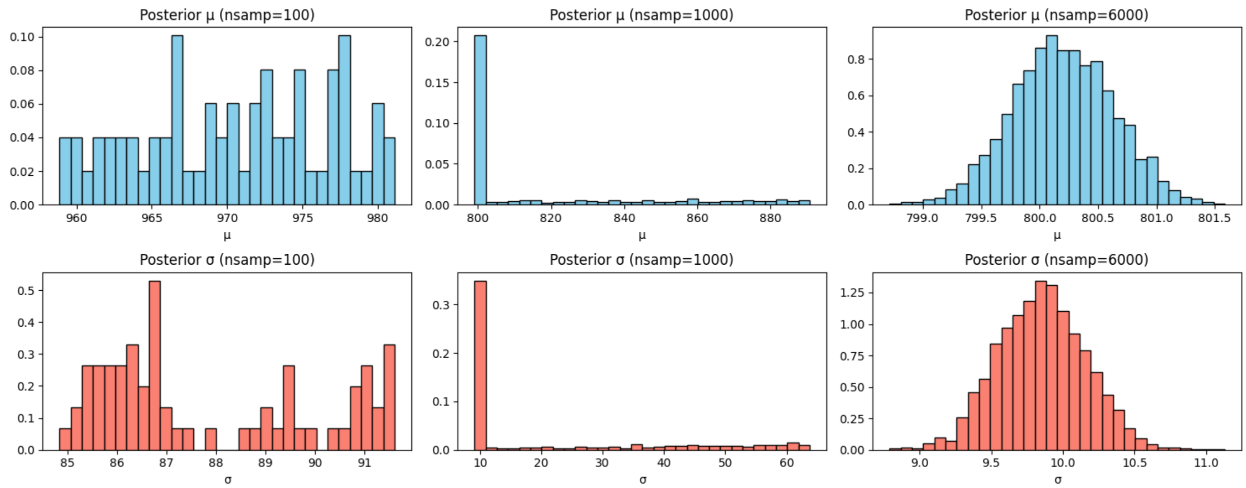
```
import numpy as np
import matplotlib.pyplot as plt

# Step 1: Simulate the data
np.random.seed(42)
true_mu = 800
true_sigma = 10
y = np.random.normal(loc=true_mu, scale=true_sigma, size=500)

# Step 2: Define gradient and potential energy 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])

def potential_energy(mu, sigma, y, n, m, s, a, b):
    log_likelihood = np.sum(-0.5 * np.log(2 * np.pi * sigma**2) - ((y
- mu)**2) / (2 * sigma**2))
    log_prior_mu = -0.5 * ((mu - m)**2) / (s**2) - np.log(s *
np.sqrt(2 * np.pi))
    log_prior_sigma = -0.5 * ((sigma - a)**2) / (b**2) - np.log(b *
np.sqrt(2 * np.pi))
    return -1 * (log_likelihood + log_prior_mu + log_prior_sigma)

# Step 3: 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)

    mu_chain[0], sigma_chain[0] = initial_q

    i = 0
    while i < nsamp - 1:
        q = np.array([mu_chain[i], sigma_chain[i]])
        p = np.random.normal(0, 1, size=2)

        current_q = q.copy()
        current_p = p.copy()

        current_U = potential_energy(*current_q, y, n, m, s, a, b)
        current_K = np.sum(current_p**2) / 2

        # Leapfrog steps
        p = p - 0.5 * step * gradient(*q, y, n, m, s, a, b)
        for _ in range(L):
            q = q + step * p
            p = p - step * gradient(*q, y, n, m, s, a, b)
        p = p + 0.5 * step * gradient(*q, y, n, m, s, a, b)
        p = -p

        proposed_U = potential_energy(*q, y, n, m, s, a, b)
        proposed_K = np.sum(p**2) / 2

        acceptance_prob = np.exp(current_U + current_K - proposed_U -
proposed_K)
        if np.random.rand() < acceptance_prob and q[1] > 0:
            mu_chain[i + 1], sigma_chain[i + 1] = q
        else:
            mu_chain[i + 1], sigma_chain[i + 1] = mu_chain[i],
sigma_chain[i]
        i += 1
```

```python
    return mu_chain[nburn:], sigma_chain[nburn:]

# Step 4: Run HMC for different nsamp values
nsamp_values = [100, 1000, 6000]
results = {}

for nsamp in nsamp_values:
    print(f"Running HMC with nsamp = {nsamp}")
    nburn = nsamp // 3
    post_mu, post_sigma = HMC(
        y=y, n=len(y), m=1000, s=20, a=10, b=2,
        step=0.02, L=12, initial_q=[1000, 11],
        nsamp=nsamp, nburn=nburn
    )
    results[nsamp] = (post_mu, post_sigma)

# Step 5: Plot comparison
fig, axes = plt.subplots(2, 3, figsize=(15, 6))
for i, nsamp in enumerate(nsamp_values):
    post_mu, post_sigma = results[nsamp]

    # μ plots
    axes[0, i].hist(post_mu, bins=30, color='skyblue',
edgecolor='black', density=True)
    axes[0, i].set_title(f"Posterior μ (nsamp={nsamp})")
    axes[0, i].set_xlabel("μ")

    # σ plots
    axes[1, i].hist(post_sigma, bins=30, color='salmon',
edgecolor='black', density=True)
    axes[1, i].set_title(f"Posterior σ (nsamp={nsamp})")
    axes[1, i].set_xlabel("σ")

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

```
Running HMC with nsamp = 100
Running HMC with nsamp = 1000
Running HMC with nsamp = 6000
```

```
/var/folders/jj/nt7bpvds7tzdfs30sl0fh4kh0000gn/T/
ipykernel_41231/3612915364.py:51: RuntimeWarning: overflow encountered
in exp
  acceptance_prob = np.exp(current_U + current_K - proposed_U -
proposed_K)
```



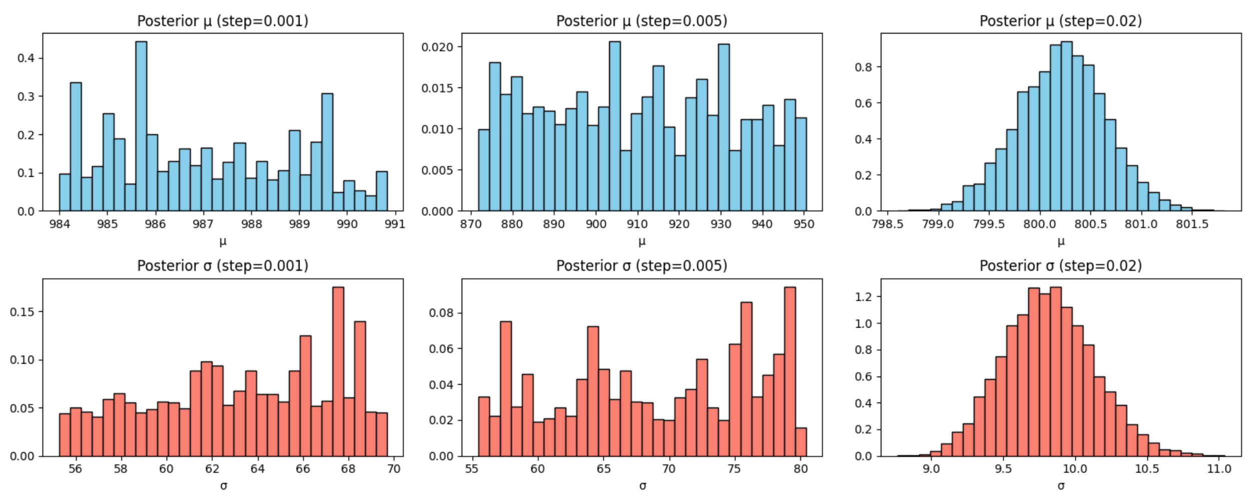
Exercise 3.3

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

# Step 1: Simulate the data
np.random.seed(42)
true_mu = 800
true_sigma = 10
y = np.random.normal(loc=true_mu, scale=true_sigma, size=500)

# Step 2: Define gradient and potential energy
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 potential_energy(mu, sigma, y, n, m, s, a, b):
    log_likelihood = np.sum(-0.5 * np.log(2 * np.pi * sigma**2) - ((y
- mu)**2) / (2 * sigma**2))
    log_prior_mu = -0.5 * ((mu - m)**2) / (s**2) - np.log(s *
np.sqrt(2 * np.pi))
    log_prior_sigma = -0.5 * ((sigma - a)**2) / (b**2) - np.log(b *
np.sqrt(2 * np.pi))
    return -1 * (log_likelihood + log_prior_mu + log_prior_sigma)

# Step 3: 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)
    mu_chain[0], sigma_chain[0] = initial_q

    i = 0
    while i < nsamp - 1:
```

```
        q = np.array([mu_chain[i], sigma_chain[i]])
        p = np.random.normal(0, 1, size=2)

        current_U = potential_energy(*q, y, n, m, s, a, b)
        current_K = np.sum(p**2) / 2

        # Leapfrog
        p = p - 0.5 * step * gradient(*q, y, n, m, s, a, b)
        for _ in range(L):
            q = q + step * p
            p = p - step * gradient(*q, y, n, m, s, a, b)
        p = p + 0.5 * step * gradient(*q, y, n, m, s, a, b)
        p = -p

        proposed_U = potential_energy(*q, y, n, m, s, a, b)
        proposed_K = np.sum(p**2) / 2

        accept_prob = np.exp(current_U + current_K - proposed_U -
proposed_K)
        if np.random.rand() < accept_prob and q[1] > 0:
            mu_chain[i + 1], sigma_chain[i + 1] = q
        else:
            mu_chain[i + 1], sigma_chain[i + 1] = mu_chain[i],
sigma_chain[i]
        i += 1

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

# Step 4: Run HMC for different step sizes
step_sizes = [0.001, 0.005, 0.02]
results = {}

for step in step_sizes:
    print(f"Running HMC with step = {step}")
    mu_post, sigma_post = HMC(
        y=y, n=len(y), m=1000, s=20, a=10, b=2,
        step=step, L=12, initial_q=[1000, 11],
        nsamp=6000, nburn=2000
    )
    results[step] = (mu_post, sigma_post)

# Step 5: Plot
fig, axes = plt.subplots(2, 3, figsize=(15, 6))

for i, step in enumerate(step_sizes):
    mu_post, sigma_post = results[step]

    axes[0, i].hist(mu_post, bins=30, color='skyblue',
edgecolor='black', density=True)
    axes[0, i].set_title(f"Posterior μ (step={step})")
```

```
    axes[0, i].set_xlabel("μ")

    axes[1, i].hist(sigma_post, bins=30, color='salmon', 
edgecolor='black', density=True)
    axes[1, i].set_title(f"Posterior σ (step={step})")
    axes[1, i].set_xlabel("σ")

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

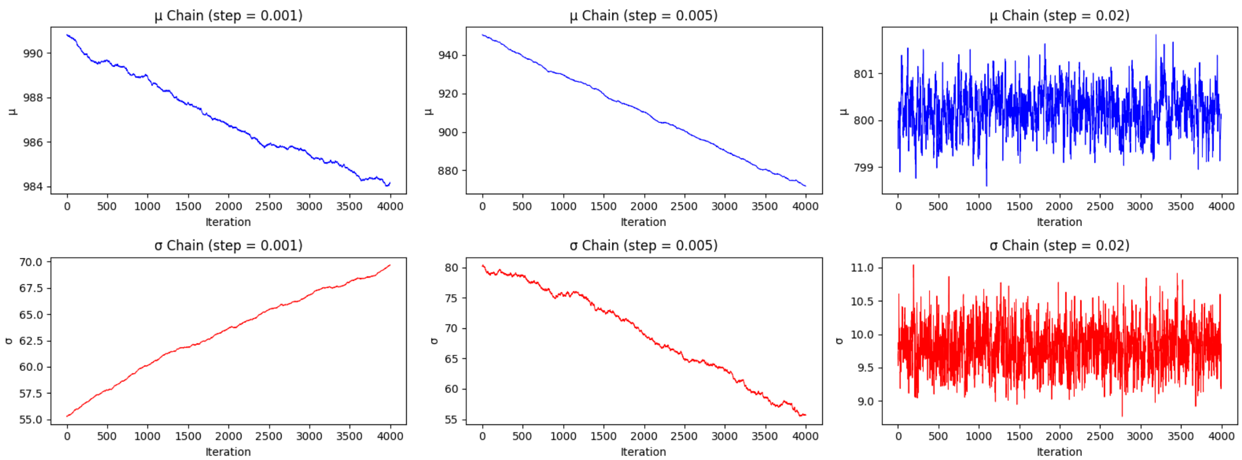
```
Running HMC with step = 0.001
Running HMC with step = 0.005
Running HMC with step = 0.02

/var/folders/jj/nt7bpvds7tzdfs30sl0fh4kh0000gn/T/
ipykernel_41231/3364791785.py:47: RuntimeWarning: overflow encountered 
in exp
  accept_prob = np.exp(current_U + current_K - proposed_U - 
proposed_K)
```



Exercise 3.4

```
import matplotlib.pyplot as plt

step_sizes = [0.001, 0.005, 0.02]

fig, axes = plt.subplots(2, 3, figsize=(16, 6))

for i, step in enumerate(step_sizes):
    mu_chain, sigma_chain = results[step]

    # Plot μ trace
    axes[0, i].plot(mu_chain, color='blue', linewidth=0.7)
    axes[0, i].set_title(f"μ Chain (step = {step})")
    axes[0, i].set_xlabel("Iteration")
```

```
    axes[0, i].set_ylabel("μ")

    # Plot σ trace
    axes[1, i].plot(sigma_chain, color='red', linewidth=0.7)
    axes[1, i].set_title(f"σ Chain (step = {step})")
    axes[1, i].set_xlabel("Iteration")
    axes[1, i].set_ylabel("σ")

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



Exercise 3.5

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

# Step 1: Simulate data
np.random.seed(42)
true_mu = 800
true_sigma = 10
y = np.random.normal(loc=true_mu, scale=true_sigma, size=500)

# Step 2: Gradient and Potential Energy
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 potential_energy(mu, sigma, y, n, m, s, a, b):
    log_likelihood = np.sum(-0.5 * np.log(2 * np.pi * sigma**2) - ((y
- mu)**2) / (2 * sigma**2))
    log_prior_mu = -0.5 * ((mu - m)**2) / (s**2) - np.log(s *
np.sqrt(2 * np.pi))
    log_prior_sigma = -0.5 * ((sigma - a)**2) / (b**2) - np.log(b *
```

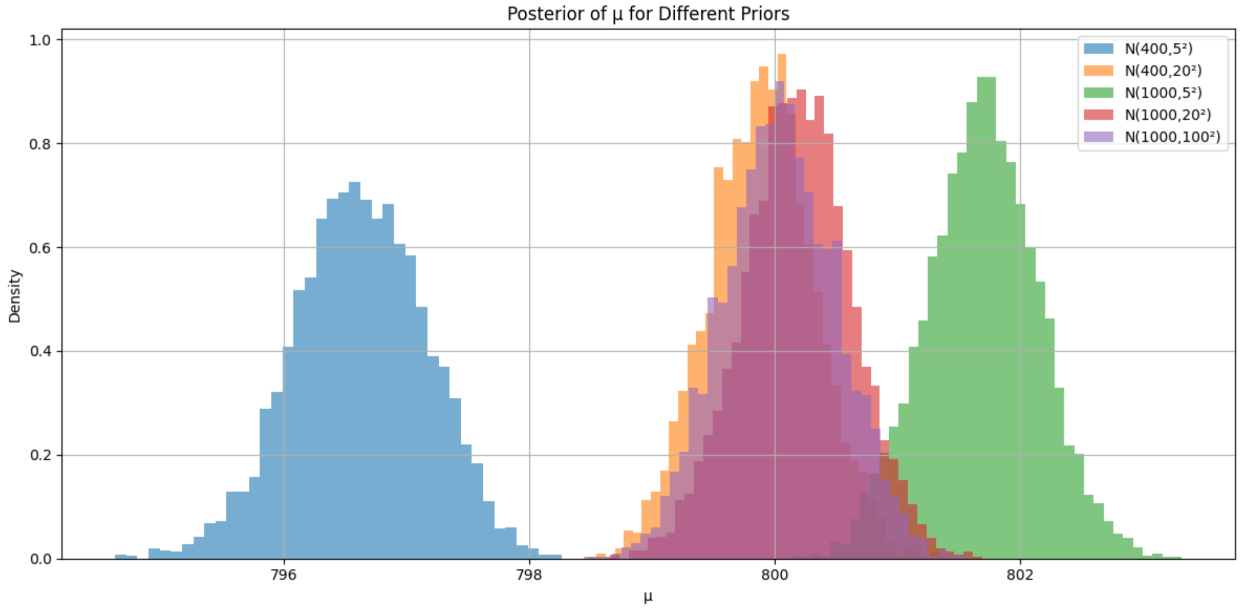
```
np.sqrt(2 * np.pi))
    return -1 * (log_likelihood + log_prior_mu + log_prior_sigma)

# Step 3: HMC function
def HMC(y, n, m, s, a, b, step, L, initial_q, nsamp, nburn):
    mu_chain = np.zeros(nsamp)
    sigma_chain = np.zeros(nsamp)
    mu_chain[0], sigma_chain[0] = initial_q

    i = 0
    while i < nsamp - 1:
        q = np.array([mu_chain[i], sigma_chain[i]])
        p = np.random.normal(0, 1, size=2)

        current_U = potential_energy(*q, y, n, m, s, a, b)
        current_K = np.sum(p**2) / 2

        # Leapfrog
        p = p - 0.5 * step * gradient(*q, y, n, m, s, a, b)
        for _ in range(L):
            q = q + step * p
            p = p - step * gradient(*q, y, n, m, s, a, b)
        p = p + 0.5 * step * gradient(*q, y, n, m, s, a, b)
        p = -p

        proposed_U = potential_energy(*q, y, n, m, s, a, b)
        proposed_K = np.sum(p**2) / 2

        accept_prob = np.exp(current_U + current_K - proposed_U -
proposed_K)
        if np.random.rand() < accept_prob and q[1] > 0:
            mu_chain[i + 1], sigma_chain[i + 1] = q
        else:
            mu_chain[i + 1], sigma_chain[i + 1] = mu_chain[i],
sigma_chain[i]
        i += 1

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

# Step 4: Try different priors on μ
priors_mu = [
    (400, 5),
    (400, 20),
    (1000, 5),
    (1000, 20),
    (1000, 100)
]

results = {}
```

```python
for m, s in priors_mu:
    print(f"Running HMC with μ ~ N({m}, {s}²)")
    mu_post, _ = HMC(
        y=y, n=len(y),
        m=m, s=s, a=10, b=2,
        step=0.02, L=12,
        initial_q=[1000, 11],
        nsamp=6000, nburn=2000
    )
    results[f"N({m},{s}²)"] = mu_post

# Step 5: Plot
plt.figure(figsize=(12, 6))
for label, chain in results.items():
    plt.hist(chain, bins=40, density=True, alpha=0.6, label=label)

plt.title("Posterior of μ for Different Priors")
plt.xlabel("μ")
plt.ylabel("Density")
plt.legend()
plt.grid(True)
plt.tight_layout()
plt.show()
```

```
Running HMC with μ ~ N(400, 5²)
```

```
/var/folders/jj/nt7bpvds7tzdfs30sl0fh4kh0000gn/T/
ipykernel_41231/1601325459.py:47: RuntimeWarning: overflow encountered
in exp
  accept_prob = np.exp(current_U + current_K - proposed_U -
proposed_K)
```

```
Running HMC with μ ~ N(400, 20²)
Running HMC with μ ~ N(1000, 5²)
Running HMC with μ ~ N(1000, 20²)
Running HMC with μ ~ N(1000, 100²)
```


Posterior of μ for Different Priors
N(400,5²)
N(400,20²)
N(1000,5²)
N(1000,20²)
N(1000,100²)
Density
μ
0.0
0.2
0.4
0.6
0.8
1.0
796
798
800
802