untitled0

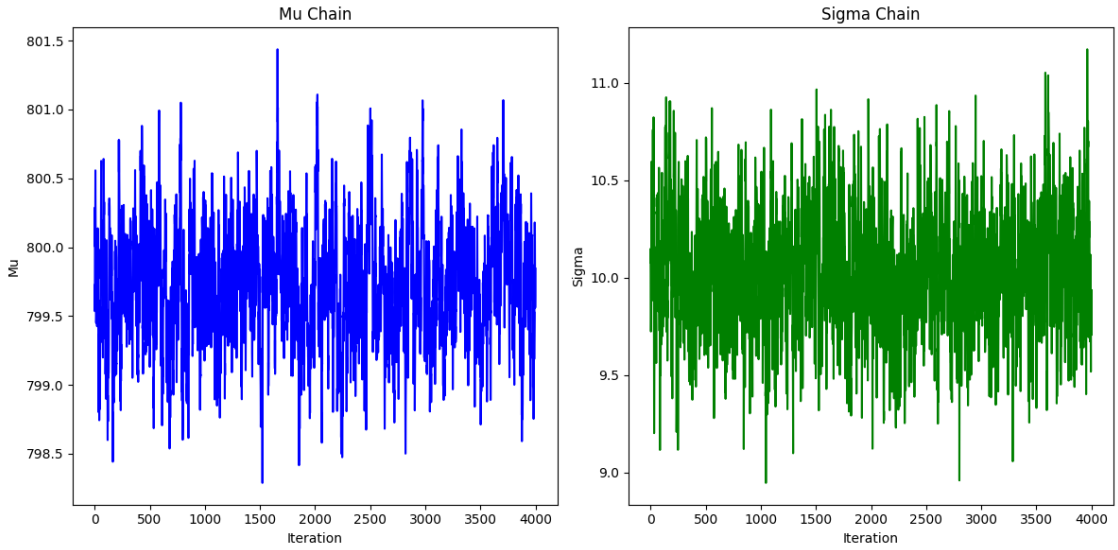
June 21, 2024

[15]:
```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from scipy.stats import norm

# 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=(12, 6))
    plt.subplot(1, 2, 1)
    plt.plot(mu_chain, color='blue')
    plt.title('Mu Chain')
    plt.xlabel('Iteration')
    plt.ylabel('Mu')

    plt.subplot(1, 2, 2)
    plt.plot(sigma_chain, color='green')
    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 synthetic data
np.random.seed(0)
true_mu = 800
true_var = 100
y = np.random.normal(true_mu, np.sqrt(true_var), 500)

# Set sampler 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}")
```

```
<ipython-input-15-ae81e16b233c>:46: RuntimeWarning: overflow encountered in exp
  accept_prob = min(1, np.exp(current_V + current_T - proposed_V - proposed_T))
```



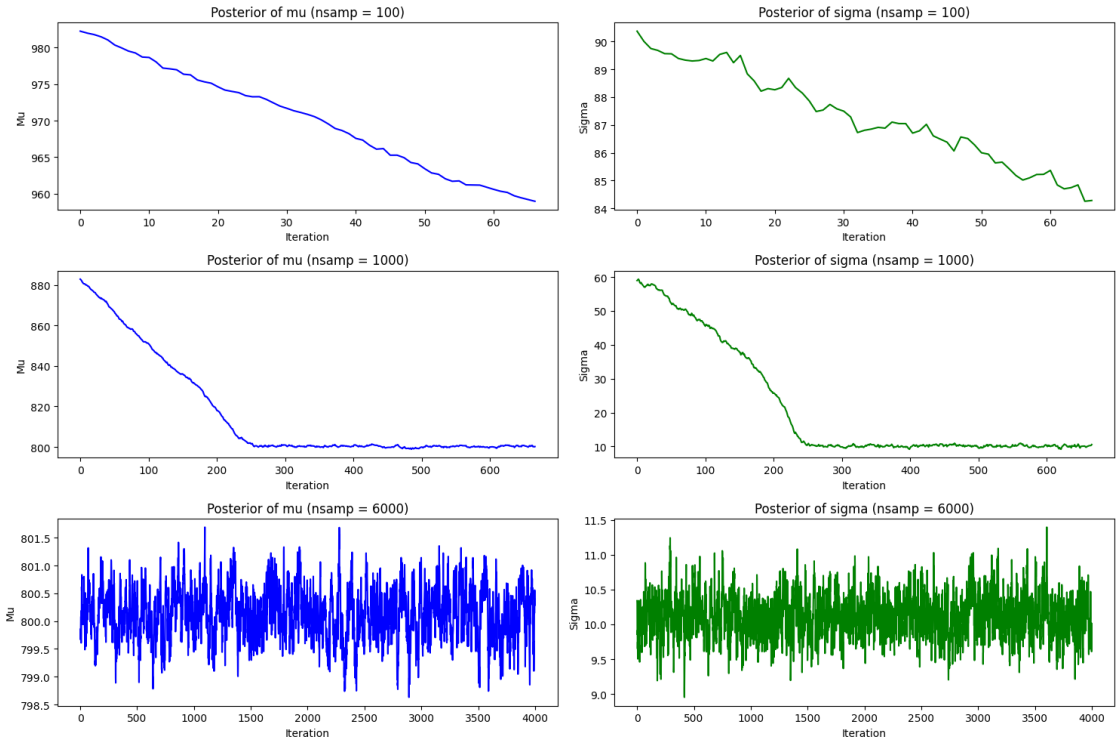
```
95% credible interval for mu: [798.88388025 800.5938084 ]
95% credible interval for sigma: [ 9.42517064 10.65641535]
```

[16]:
```python
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
from scipy.stats import norm

# Generate synthetic 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 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 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

# Settings for different sample sizes and burn-in ratios
sample_settings = [100, 1000, 6000]
burnin_ratios = [1/3, 1/3, 1/3]
step = 0.02
L = 12
initial_q = [1000, 11]

# Run HMC sampler for each setting
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'], color='blue')
    plt.title(f'Posterior of mu (nsamp = {nsamp})')
    plt.xlabel('Iteration')
    plt.ylabel('Mu')

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

plt.tight_layout()
plt.show()

# Calculate and print the 95% credible intervals
for nsamp, df_posterior in results:
    mu_credible_interval = np.percentile(df_posterior['mu_chain'], [2.5, 97.5])
    sigma_credible_interval = np.percentile(df_posterior['sigma_chain'], [2.5,␣
 ↪97.5])
    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")
```

```
<ipython-input-16-622c300dd696>: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): [959.36237994 981.86935002]
95% credible interval for sigma (nsamp = 100): [84.5581077  89.83552927]

95% credible interval for mu (nsamp = 1000): [799.3346759  877.53153936]
95% credible interval for sigma (nsamp = 1000): [ 9.64289526 57.60094111]

95% credible interval for mu (nsamp = 6000): [799.21236702 801.02514051]
95% credible interval for sigma (nsamp = 6000): [ 9.53766473 10.75581162]
```

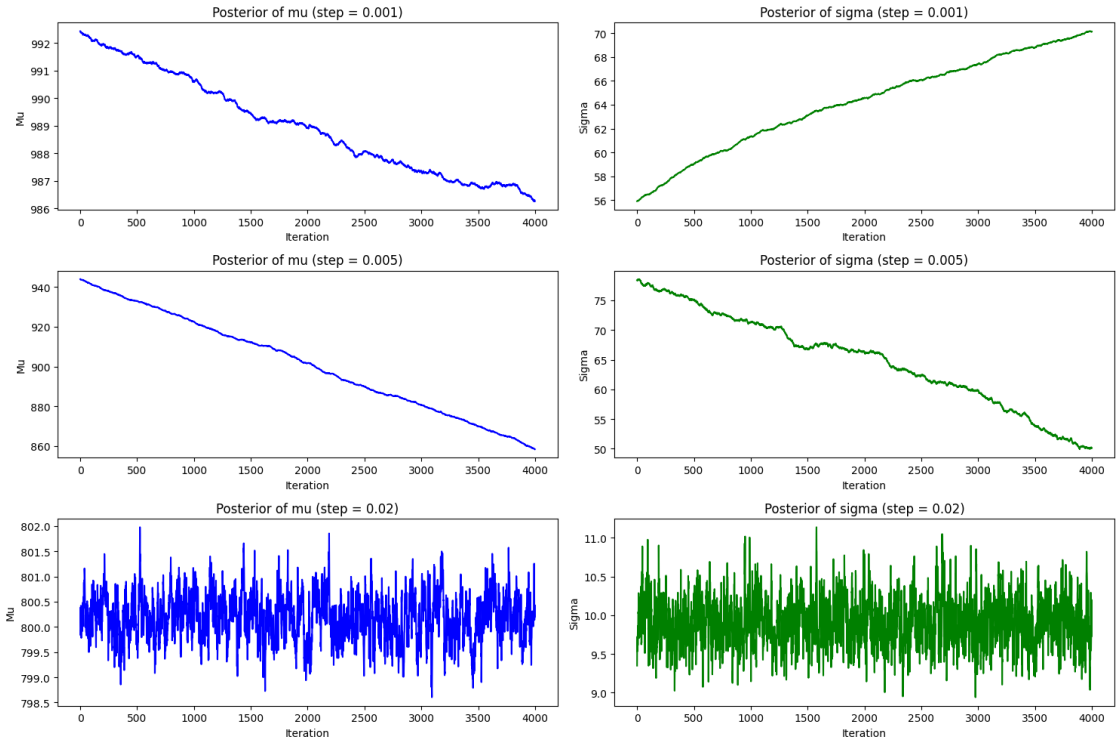
[17]:
```python
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
from scipy.stats import norm

# Generate synthetic 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 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 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
    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(size=q.shape)  # Generate random momentum at the␣
 ↪current position

        # Leapfrog integration
        current_q = q.copy()
        current_p = p.copy()
        for _ in range(L):
            p -= (step / 2) * gradient(q[0], q[1], y, n, m, s, a, b)  #␣
 ↪Half-step in momentum
            q += step * p  # Full-step in position
            p -= (step / 2) * gradient(q[0], q[1], y, n, m, s, a, b)  #␣
 ↪Half-step in momentum

        # Compute current and proposed energies
        current_V = V(current_q[0], current_q[1], y, n, m, s, a, b)
        current_T = np.sum(current_p**2) / 2
        proposed_V = V(q[0], q[1], y, n, m, s, a, b)
        proposed_T = np.sum(p**2) / 2

        # Accept/reject step
        accept_prob = min(1, np.exp(current_V + current_T - proposed_V -␣
 ↪proposed_T))
```

```
        if accept_prob > np.random.rand():
            mu_chain[i] = q[0]
            sigma_chain[i] = q[1]
        else:
            mu_chain[i] = mu_chain[i-1]
            sigma_chain[i] = sigma_chain[i-1]
            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 = []

# Run HMC sampler for each step size setting
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'], color='blue')
    plt.title(f'Posterior of mu (step = {step})')
    plt.xlabel('Iteration')
    plt.ylabel('Mu')

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

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

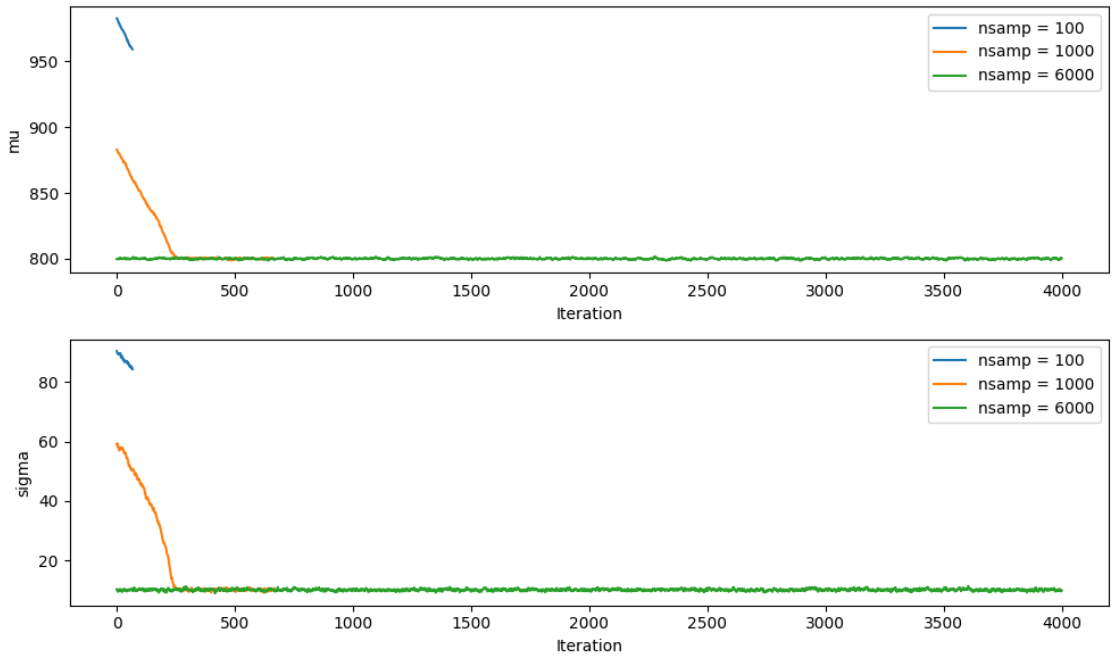
```
# Calculate and print the 95% credible intervals
for step, df_posterior in results_step:
    mu_credible_interval = np.percentile(df_posterior['mu_chain'], [2.5, 97.5])
    sigma_credible_interval = np.percentile(df_posterior['sigma_chain'], [2.5,␣
 ↪97.5])
    print(f"95% credible interval for mu (step = {step}):␣
 ↪{mu_credible_interval}")
    print(f"95% credible interval for sigma (step = {step}):␣
 ↪{sigma_credible_interval}\n")
```

```
<ipython-input-17-0d879112465d>:52: RuntimeWarning: overflow encountered in exp
  accept_prob = min(1, np.exp(current_V + current_T - proposed_V - proposed_T))
```



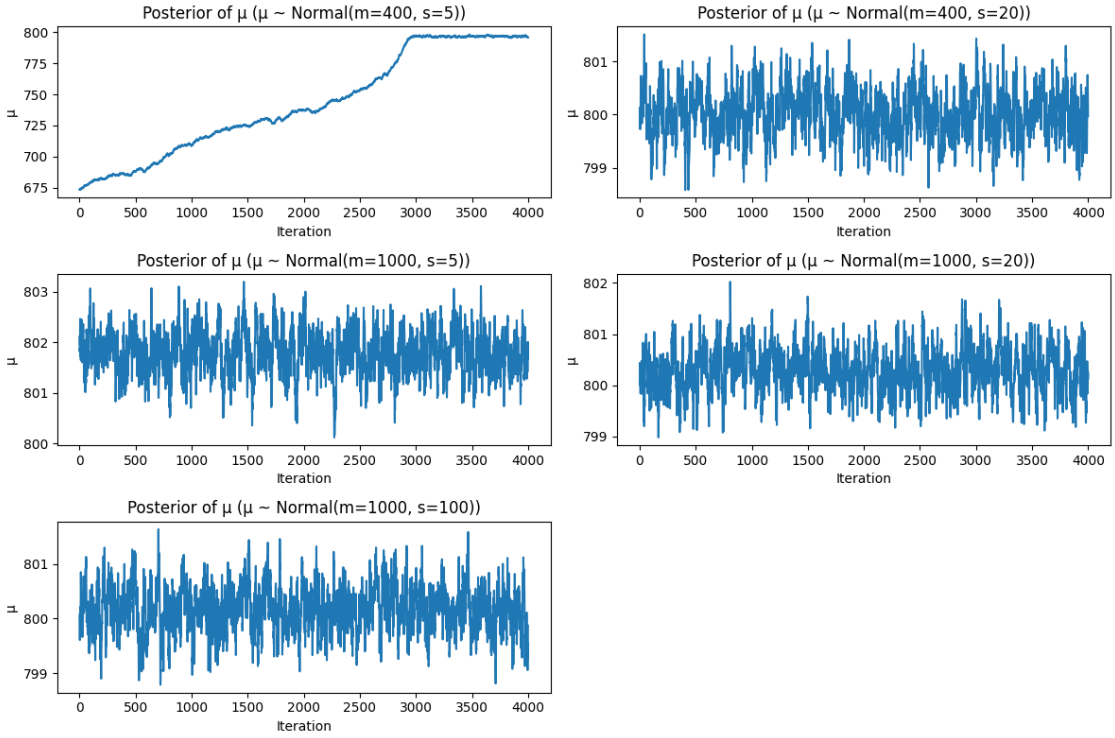

```
95% credible interval for mu (step = 0.001): [986.53058438 992.13066719]
95% credible interval for sigma (step = 0.001): [56.48609209 69.84539208]

95% credible interval for mu (step = 0.005): [861.17656769 941.05660798]
95% credible interval for sigma (step = 0.005): [50.36189655 77.51440887]

95% credible interval for mu (step = 0.02): [799.28179534 801.0748466 ]
95% credible interval for sigma (step = 0.02): [ 9.33136191 10.55693004]
```

[19]:
```
import matplotlib.pyplot as plt

# Plot the mu and sigma chains
plt.figure(figsize=(10, 6))

plt.subplot(2, 1, 1)
plt.plot(results[0][1]['mu_chain'], label='nsamp = 100')
plt.plot(results[1][1]['mu_chain'], label='nsamp = 1000')
plt.plot(results[2][1]['mu_chain'], label='nsamp = 6000')
plt.xlabel('Iteration')
plt.ylabel('mu')
plt.legend()

plt.subplot(2, 1, 2)
plt.plot(results[0][1]['sigma_chain'], label='nsamp = 100')
plt.plot(results[1][1]['sigma_chain'], label='nsamp = 1000')
plt.plot(results[2][1]['sigma_chain'], label='nsamp = 6000')
plt.xlabel('Iteration')
plt.ylabel('sigma')
plt.legend()

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



[20]:
```
import matplotlib.pyplot as plt

def plot_posteriors(results):
    plt.figure(figsize=(12, 8))

    for i, (prior_desc, df_posterior) in enumerate(results):
        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()

# 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 different prior settings
plot_posteriors(results_prior_sensitivity)
```

```
<ipython-input-17-0d879112465d>:52: 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

[ ]: