In [8]:
```
import numpy as np
from scipy.integrate import fixed_quad, quad
import matplotlib.pyplot as plt
import numpy as np
import matplotlib.pyplot as plt
k = 1.38064852e-23
h = 6.626e-34
pi = np.pi
c = 3e8
hbar = h / (2 * pi)
```

Task 1

In [4]:
```
prefactor = k**4 / (c**2 * hbar**3 * 4 * pi**2)

# ----------------------------
# Part A: Transform the integral
# ----------------------------

def integrand_z(z):
    x = z / (1 - z)
    dx_dz = 1 / (1 - z)**2
    return (x**3 / (np.exp(x) - 1)) * dx_dz

I_fixed, _ = fixed_quad(integrand_z, 0, 1, n=5000)

# ----------------------------
# Part B: Compute σ using fixed_quad result
# ----------------------------

sigma_fixed = prefactor * I_fixed
print("Stefan-Boltzmann constant (fixed_quad):", sigma_fixed)

# ----------------------------
# Part C: Compute the integral directly over [0, ∞) using quad
# ----------------------------

def integrand_x(x):
    return x**3 / (np.exp(x) - 1)

I_quad, err = quad(integrand_x, 0, np.inf)
sigma_quad = prefactor * I_quad
print("Stefan-Boltzmann constant (quad with infinite limit):", sigma_quad)
```

```
Stefan-Boltzmann constant (fixed_quad): 5.662703503453973e-08
Stefan-Boltzmann constant (quad with infinite limit): 5.662703503454045e-08
```

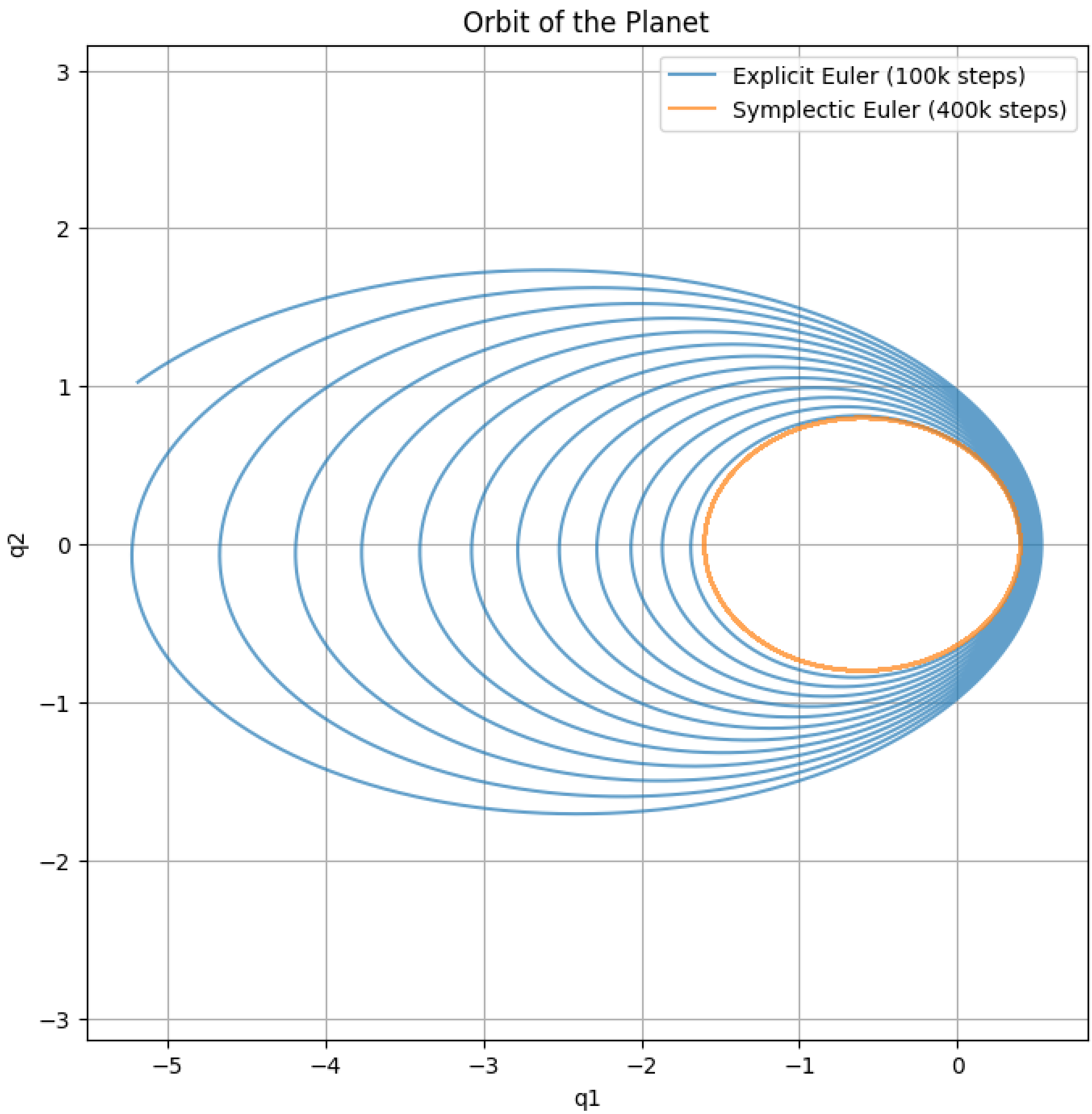
```
C:\Users\Eric\AppData\Local\Temp\ipykernel_14380\3453470642.py:10: RuntimeWarning: overf
low encountered in exp
  return (x**3 / (np.exp(x) - 1)) * dx_dz
C:\Users\Eric\AppData\Local\Temp\ipykernel_14380\3453470642.py:26: RuntimeWarning: overf
low encountered in exp
  return x**3 / (np.exp(x) - 1)
```

Task 2

In [7]:
```
# Parameters and initial conditions
e = 0.6
Tf = 200.0
```

```python
# Explicit Euler parameters
N_euler = 100000
dt_euler = Tf / N_euler

# Symplectic Euler parameters
N_symp = 400000
dt_symp = Tf / N_symp

# Initial conditions
q1_0 = 1 - e
q2_0 = 0.0
v1_0 = 0.0
v2_0 = np.sqrt((1+e)/(1-e))

def acceleration(q1, q2):
    r = np.sqrt(q1**2 + q2**2)
    # Avoid division by zero
    r3 = r**3 if r != 0 else np.inf
    return -q1 / r3, -q2 / r3

# ---------------------------
# Explicit Euler method
# ---------------------------
q1_euler = np.zeros(N_euler+1)
q2_euler = np.zeros(N_euler+1)
v1_euler = np.zeros(N_euler+1)
v2_euler = np.zeros(N_euler+1)

q1_euler[0], q2_euler[0] = q1_0, q2_0
v1_euler[0], v2_euler[0] = v1_0, v2_0

for n in range(N_euler):
    a1, a2 = acceleration(q1_euler[n], q2_euler[n])
    # Update positions
    q1_euler[n+1] = q1_euler[n] + dt_euler * v1_euler[n]
    q2_euler[n+1] = q2_euler[n] + dt_euler * v2_euler[n]
    # Update velocities
    v1_euler[n+1] = v1_euler[n] + dt_euler * a1
    v2_euler[n+1] = v2_euler[n] + dt_euler * a2

# ---------------------------
# Symplectic Euler method
# ---------------------------
q1_symp = np.zeros(N_symp+1)
q2_symp = np.zeros(N_symp+1)
v1_symp = np.zeros(N_symp+1)
v2_symp = np.zeros(N_symp+1)

q1_symp[0], q2_symp[0] = q1_0, q2_0
v1_symp[0], v2_symp[0] = v1_0, v2_0

for n in range(N_symp):
    # Update momentum (velocity) using the acceleration at current position
    a1, a2 = acceleration(q1_symp[n], q2_symp[n])
    v1_symp[n+1] = v1_symp[n] + dt_symp * a1
    v2_symp[n+1] = v2_symp[n] + dt_symp * a2
    # Update positions using the new velocities
    q1_symp[n+1] = q1_symp[n] + dt_symp * v1_symp[n+1]
    q2_symp[n+1] = q2_symp[n] + dt_symp * v2_symp[n+1]

# ---------------------------
# Plotting both orbits
# ---------------------------
plt.figure(figsize=(8,8))
plt.plot(q1_euler, q2_euler, label='Explicit Euler (100k steps)', alpha=0.7)
plt.plot(q1_symp, q2_symp, label='Symplectic Euler (400k steps)', alpha=0.7)
```

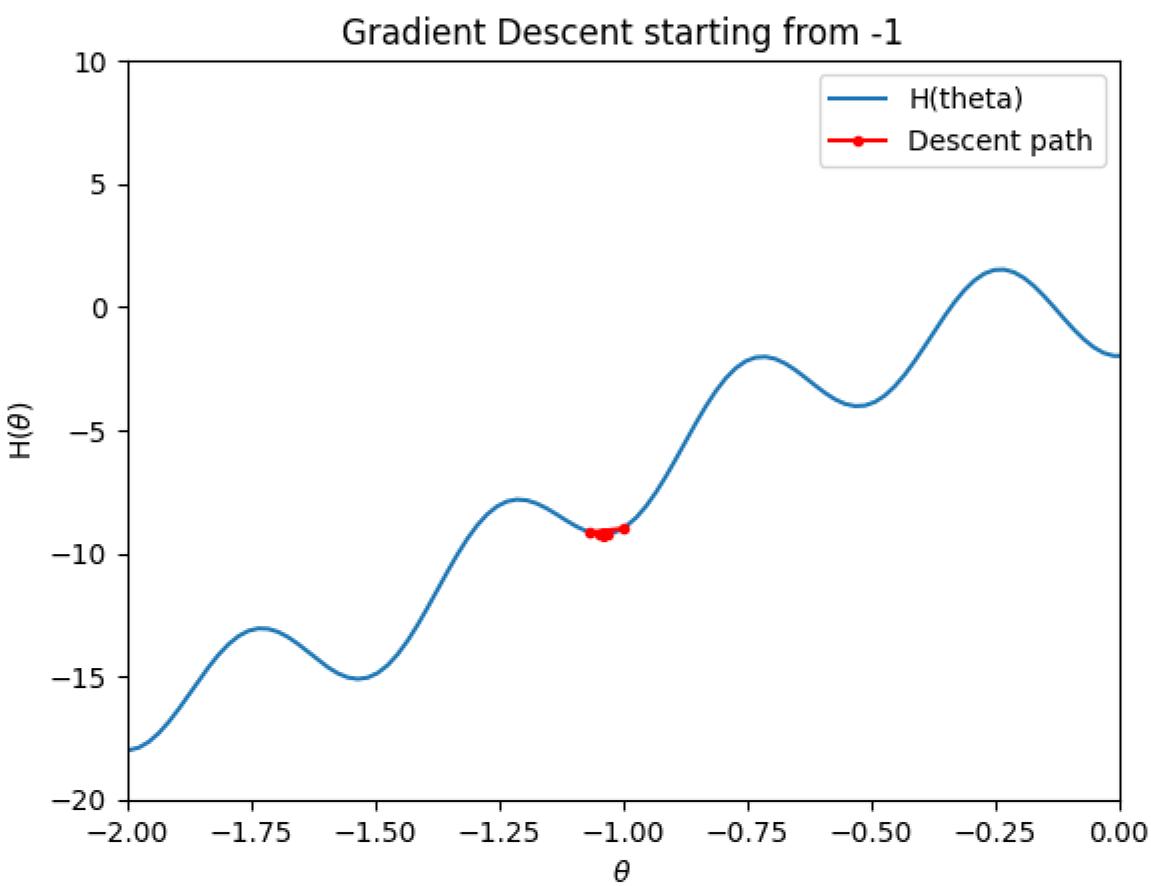
```python
plt.xlabel('q1')
plt.ylabel('q2')
plt.title('Orbit of the Planet')
plt.legend()
plt.axis('equal')
plt.grid(True)
plt.show()
```

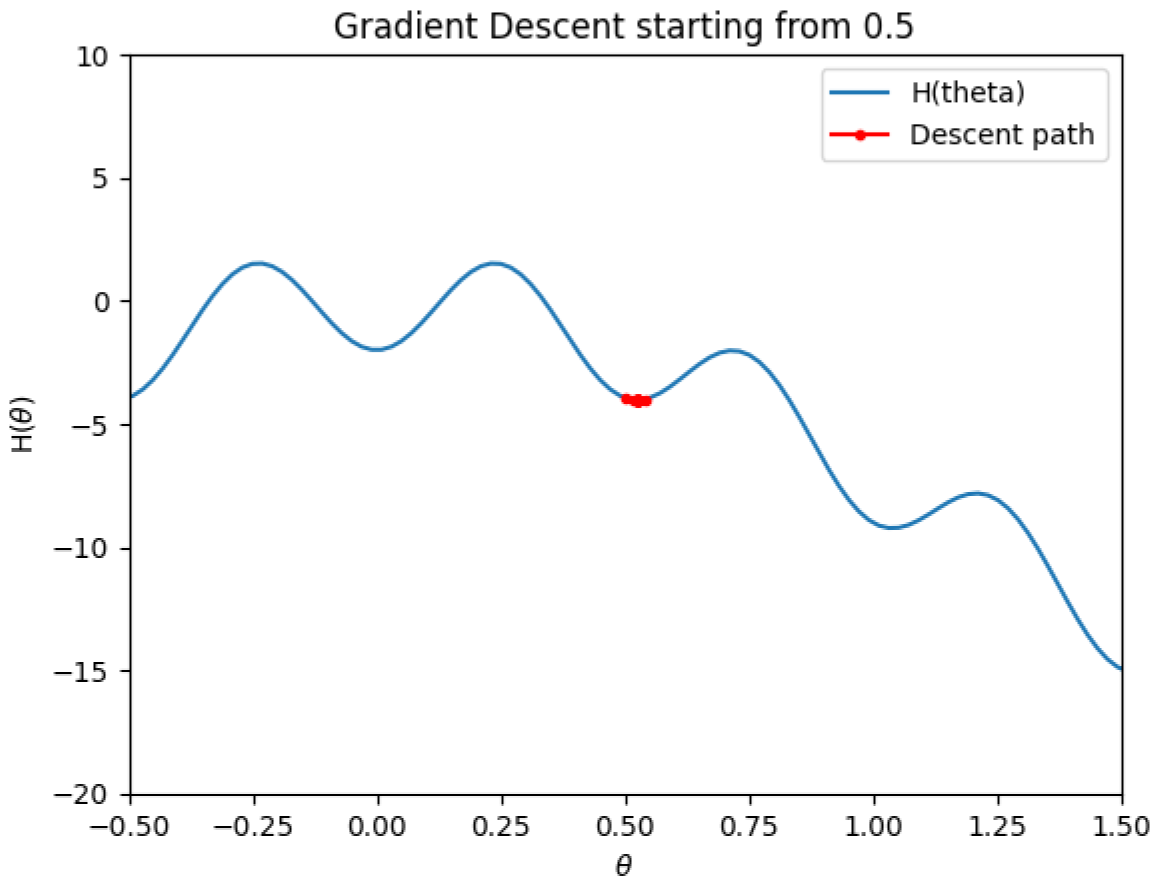


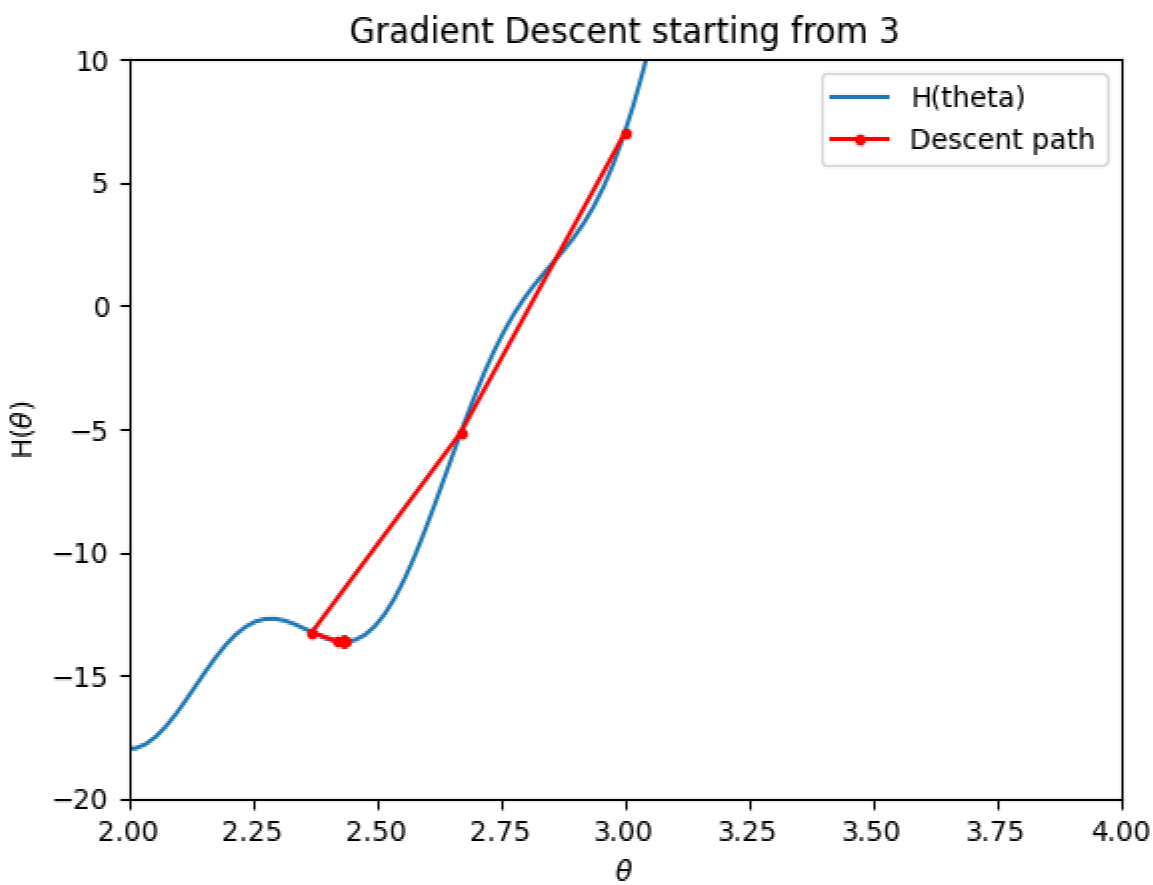
Task 3

In [ ]:
```python
# Part a

def H(theta):
    return theta**4 - 8*theta**2 - 2*np.cos(4*np.pi*theta)

def dH(theta):
    return 4*theta**3 - 16*theta + 8*np.pi*np.sin(4*np.pi*theta)
```

```python
initial_thetas = [-1, 0.5, 3]
learning_rate = 0.0055 # This may need tuning
max_iters = 10000
tolerance = 1e-6

theta_vals = np.linspace(-4, 4, 400)
H_vals = H(theta_vals)

for theta0 in initial_thetas:
    theta = theta0
    history = [theta]
    for i in range(max_iters):
        grad = dH(theta)
        theta_new = theta - learning_rate * grad
        history.append(theta_new)
        if abs(theta_new - theta) < tolerance:
            break
        theta = theta_new

    plt.figure()
    plt.plot(theta_vals, H_vals, label='H(theta)')
    plt.plot(history, H(np.array(history)), 'ro-', markersize=3, label='Descent path')
    plt.xlabel(r'$\theta$')
    plt.xlim(theta0-1,theta0+1)
    plt.ylim(-20,10)
    plt.ylabel('H($\\theta$)')
    plt.title(f'Gradient Descent starting from {theta0}')
    plt.legend()
    plt.show()
```



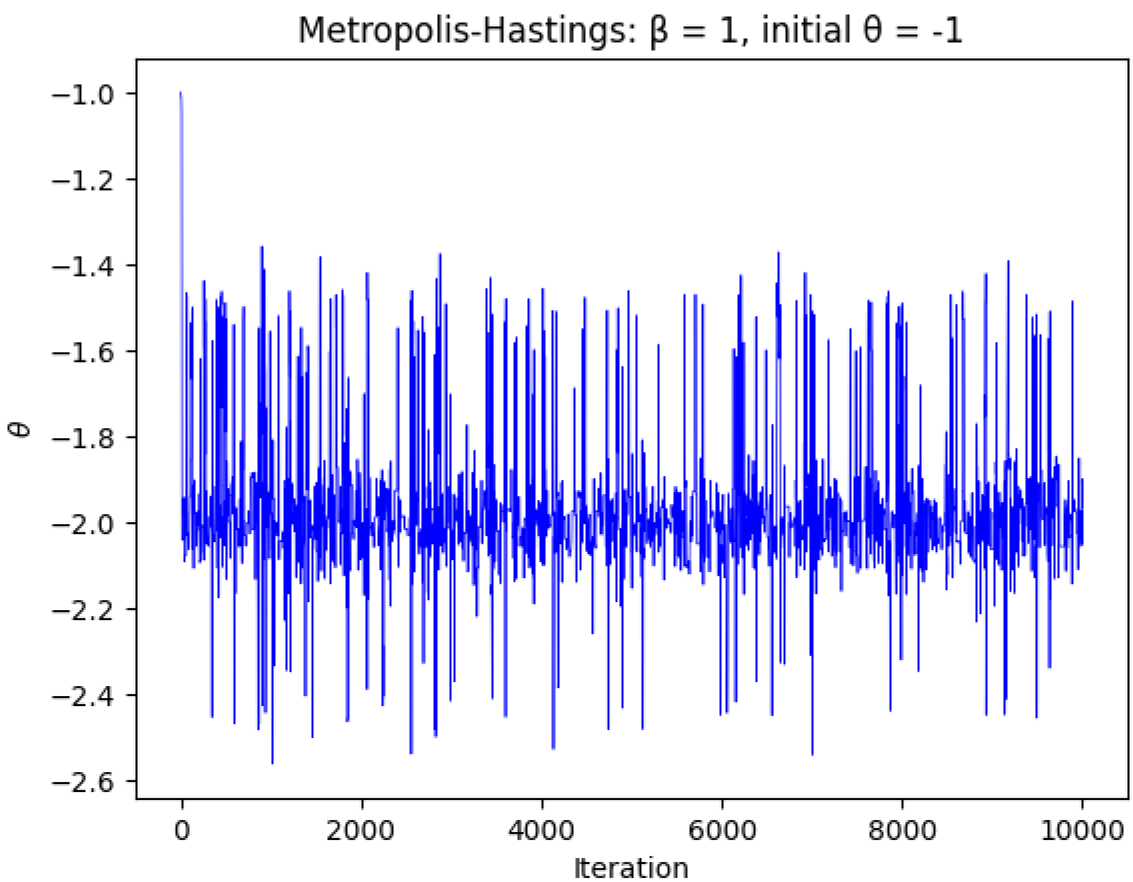


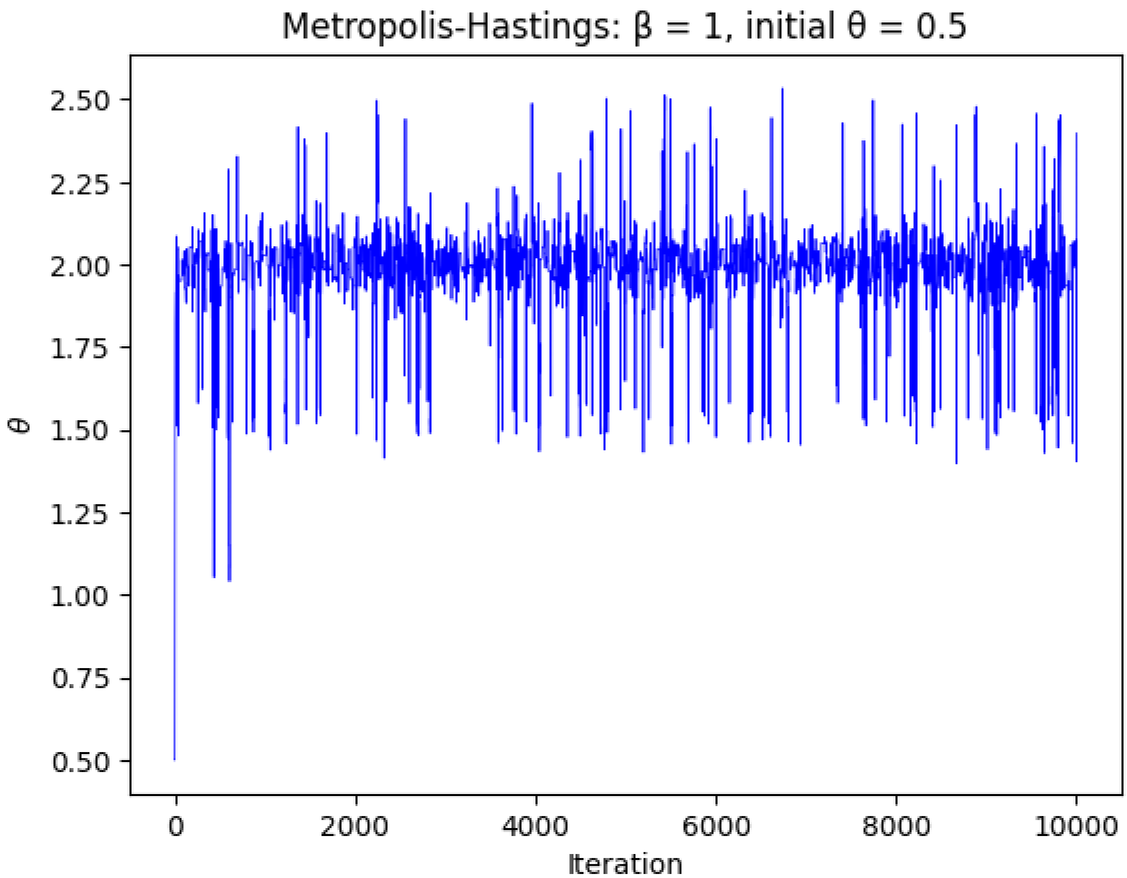


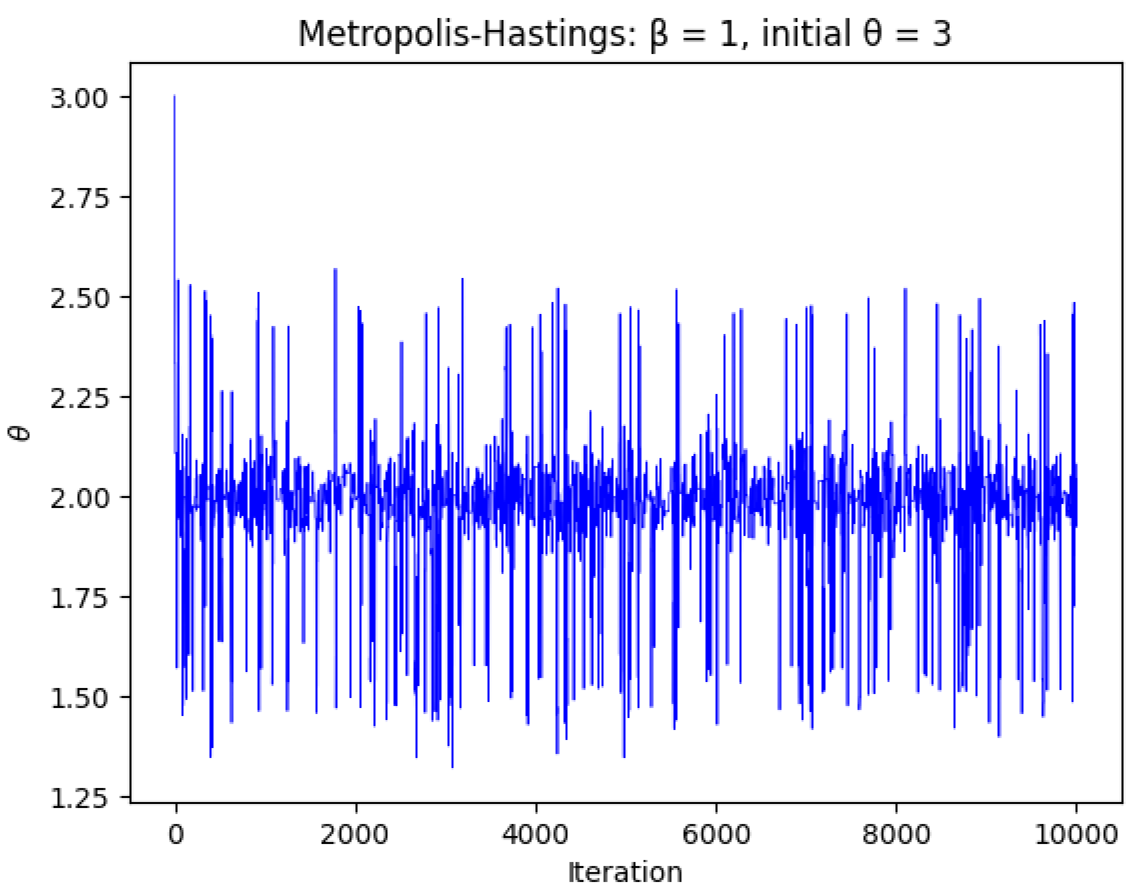
In [22]:

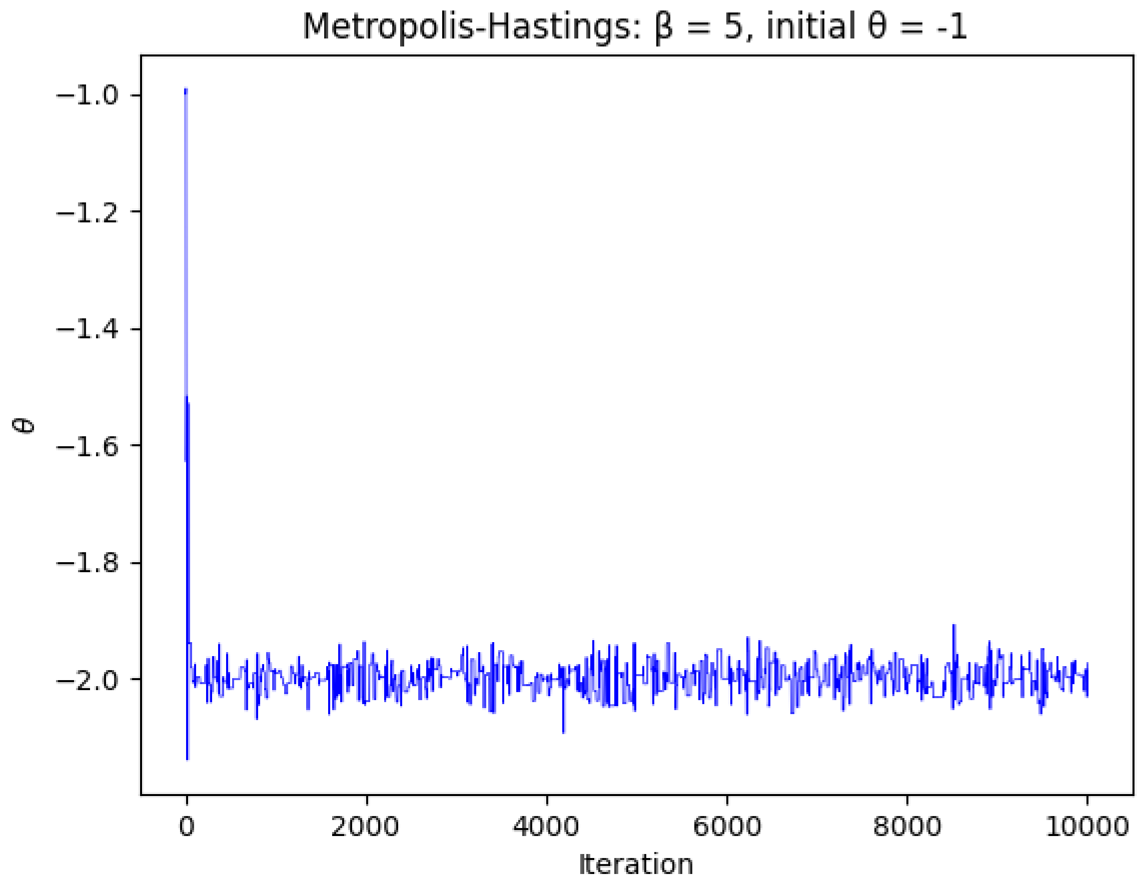
```
# Part b
def metropolis_hastings(theta0, beta, sigma, iterations):
    theta = theta0
    chain = [theta]
    for i in range(iterations):
        theta_proposal = theta + np.random.normal(0, sigma)
        dH_val = H(theta_proposal) - H(theta)
```

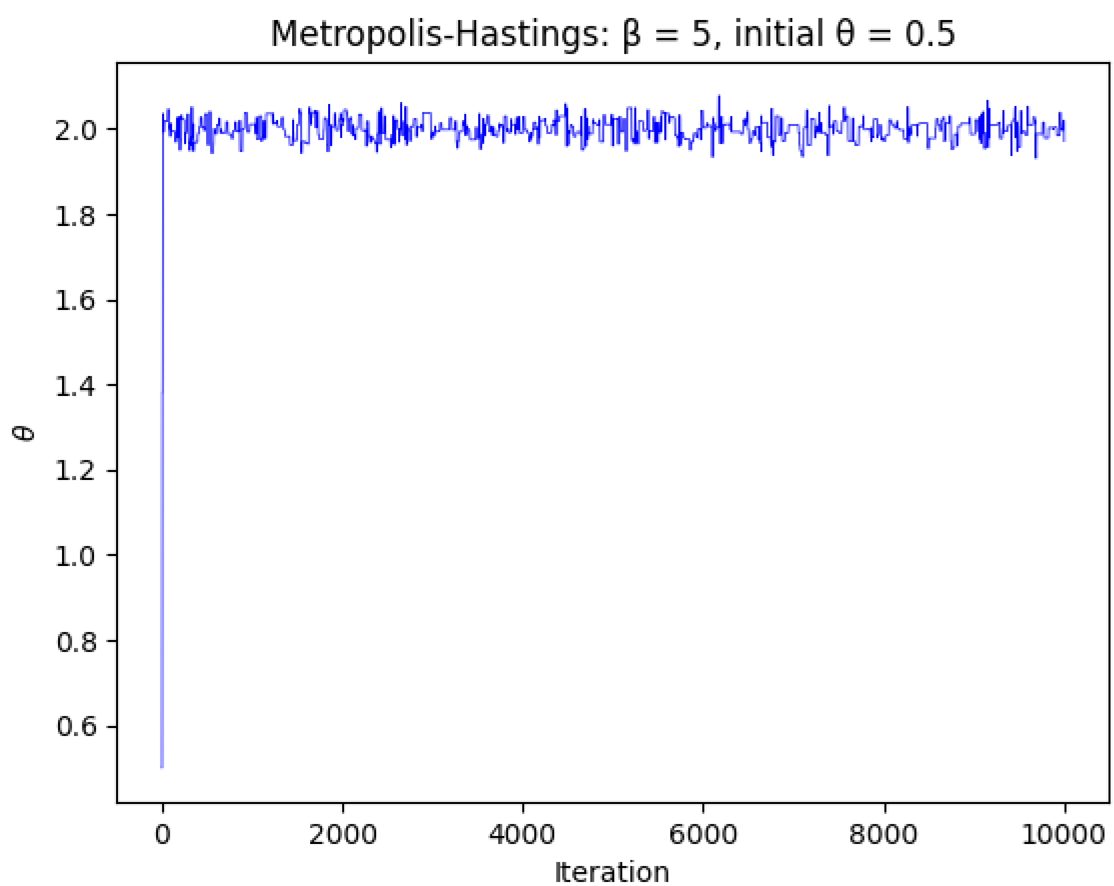
```python
        r = np.exp(-beta * dH_val)
        if r >= 1 or np.random.rand() < r:
            theta = theta_proposal
        chain.append(theta)
    return chain

# Parameters
iterations = 10000
sigma = 0.5
beta_values = [1, 5, 10]
initial_thetas = [-1, 0.5, 3]

for beta in beta_values:
    for theta0 in initial_thetas:
        chain = metropolis_hastings(theta0, beta, sigma, iterations)
        plt.figure()
        plt.plot(chain, 'b-', lw=0.5)
        plt.xlabel('Iteration')
        plt.ylabel(r'$\theta$')
        plt.title(f'Metropolis-Hastings: β = {beta}, initial θ = {theta0}')
        plt.show()
```

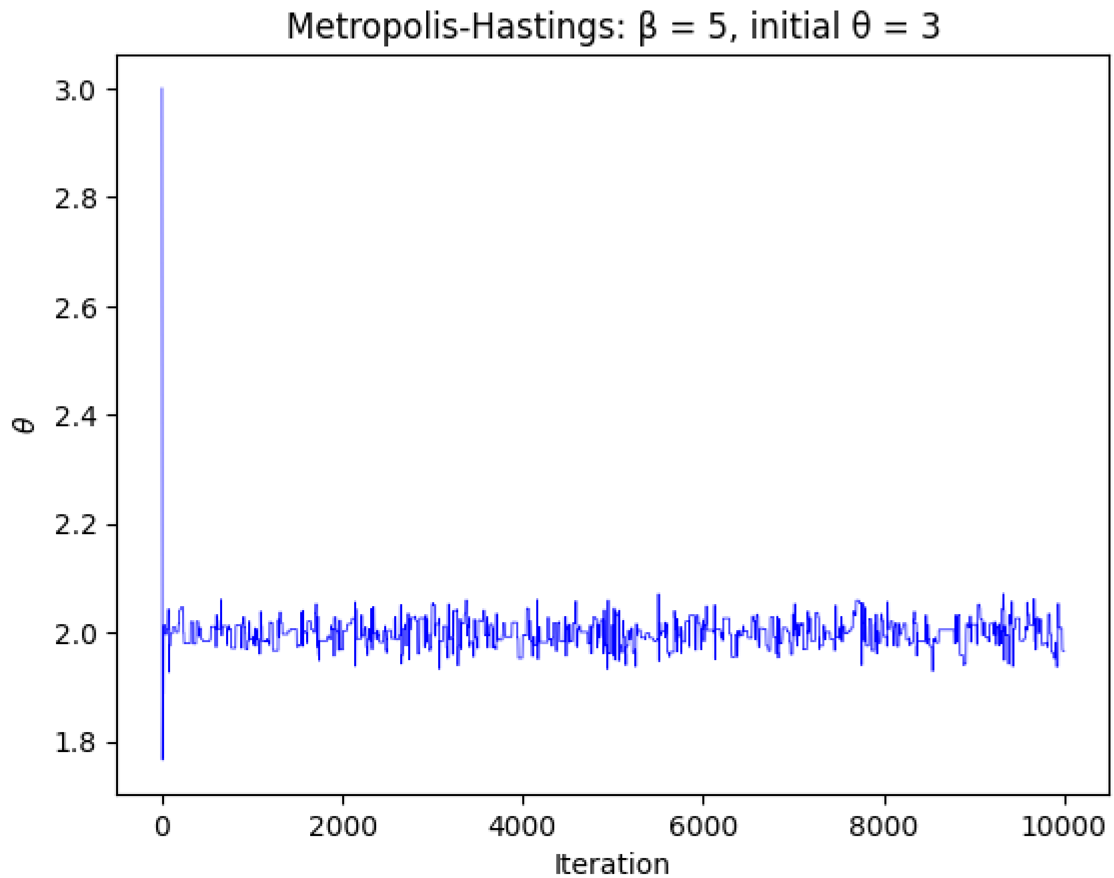


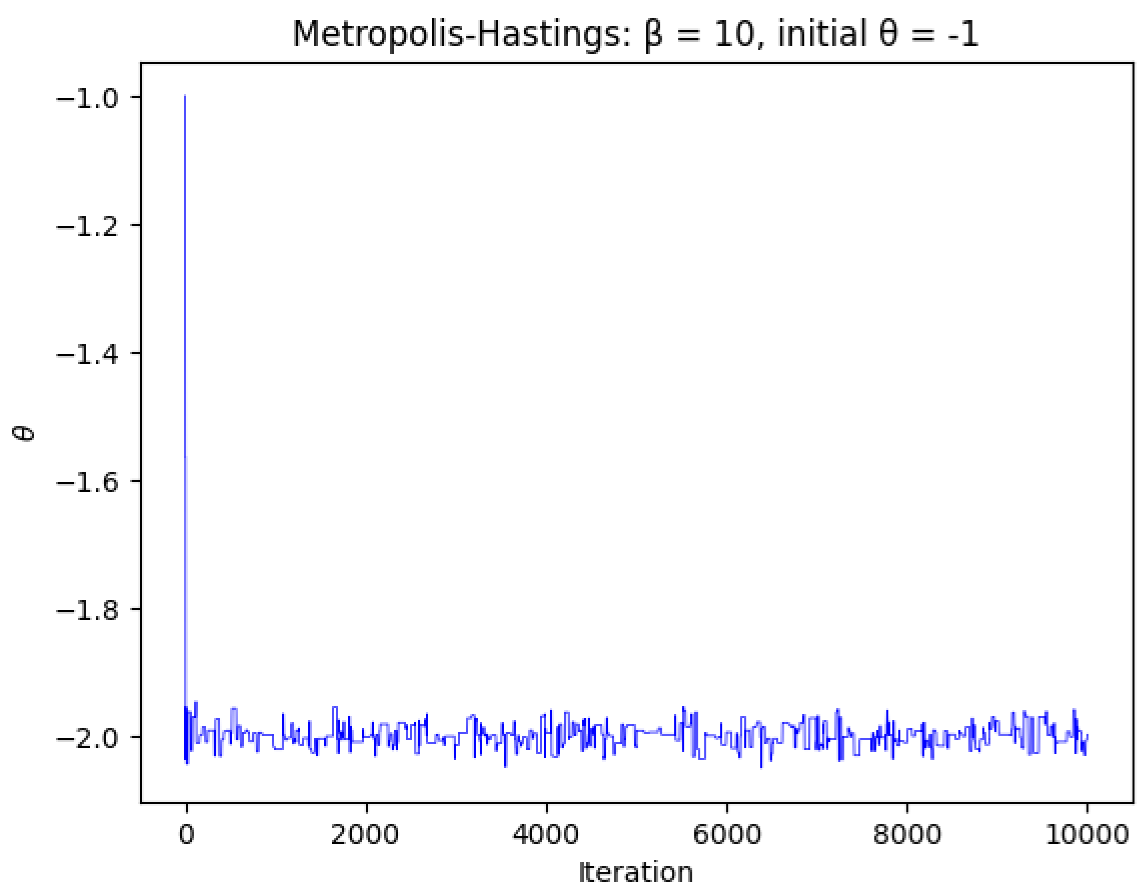

Metropolis-Hastings: β = 1, initial θ = 0.5
θ
Iteration
2.50
2.25
2.00
1.75
1.50
1.25
1.00
0.75
0.50
0
2000
4000
6000
8000
10000

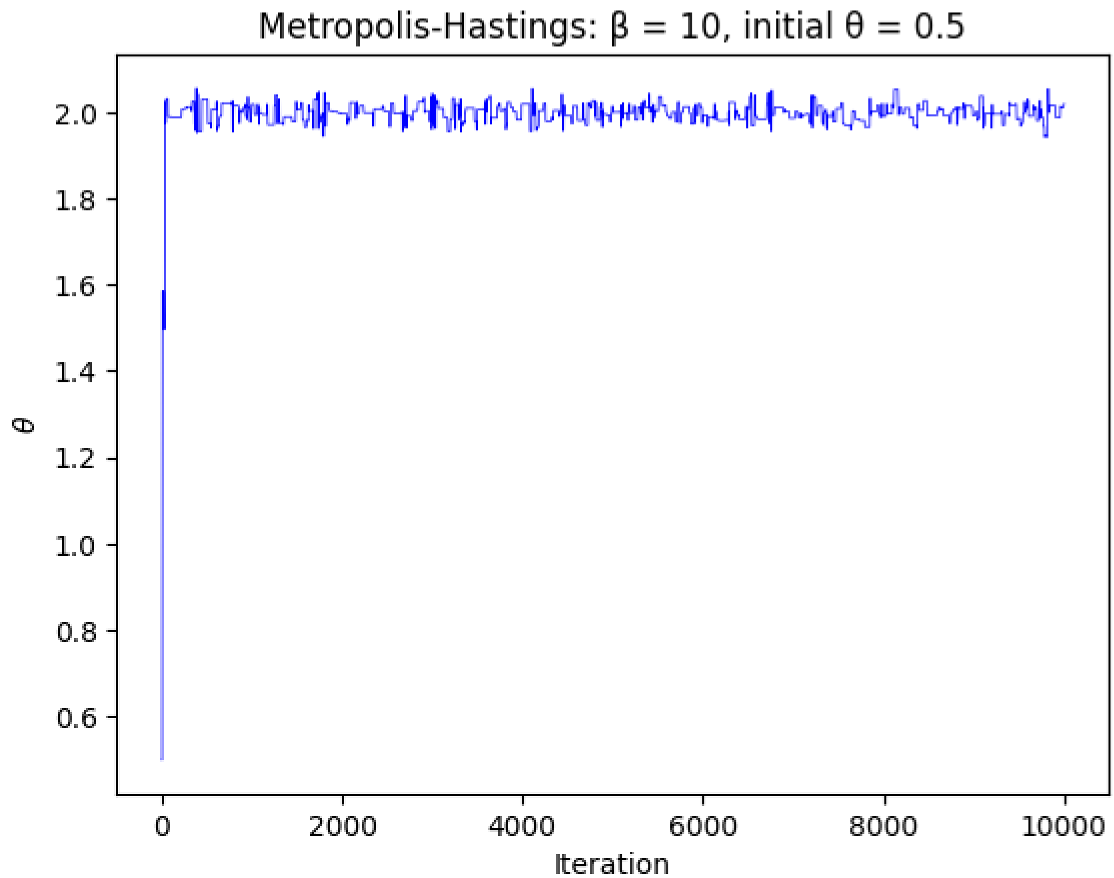

Metropolis-Hastings: β = 1, initial θ = 3
θ
Iteration
3.00
2.75
2.50
2.25
2.00
1.75
1.50
1.25
0
2000
4000
6000
8000
10000

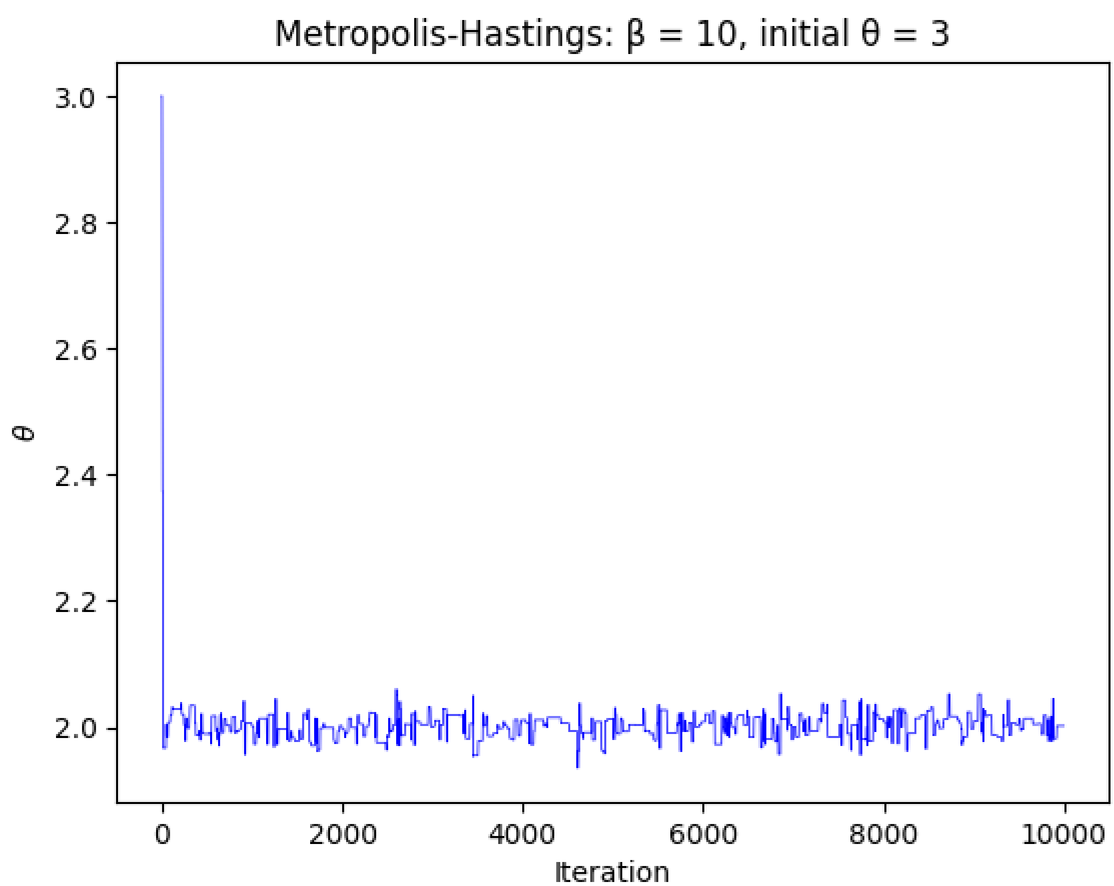

Metropolis-Hastings: β = 5, initial θ = -1
θ
Iteration


Metropolis-Hastings: β = 5, initial θ = 0.5
θ
Iteration


Metropolis-Hastings: β = 5, initial θ = 3
θ
3.0
2.8
2.6
2.4
2.2
2.0
1.8
0
2000
4000
6000
8000
10000
Iteration

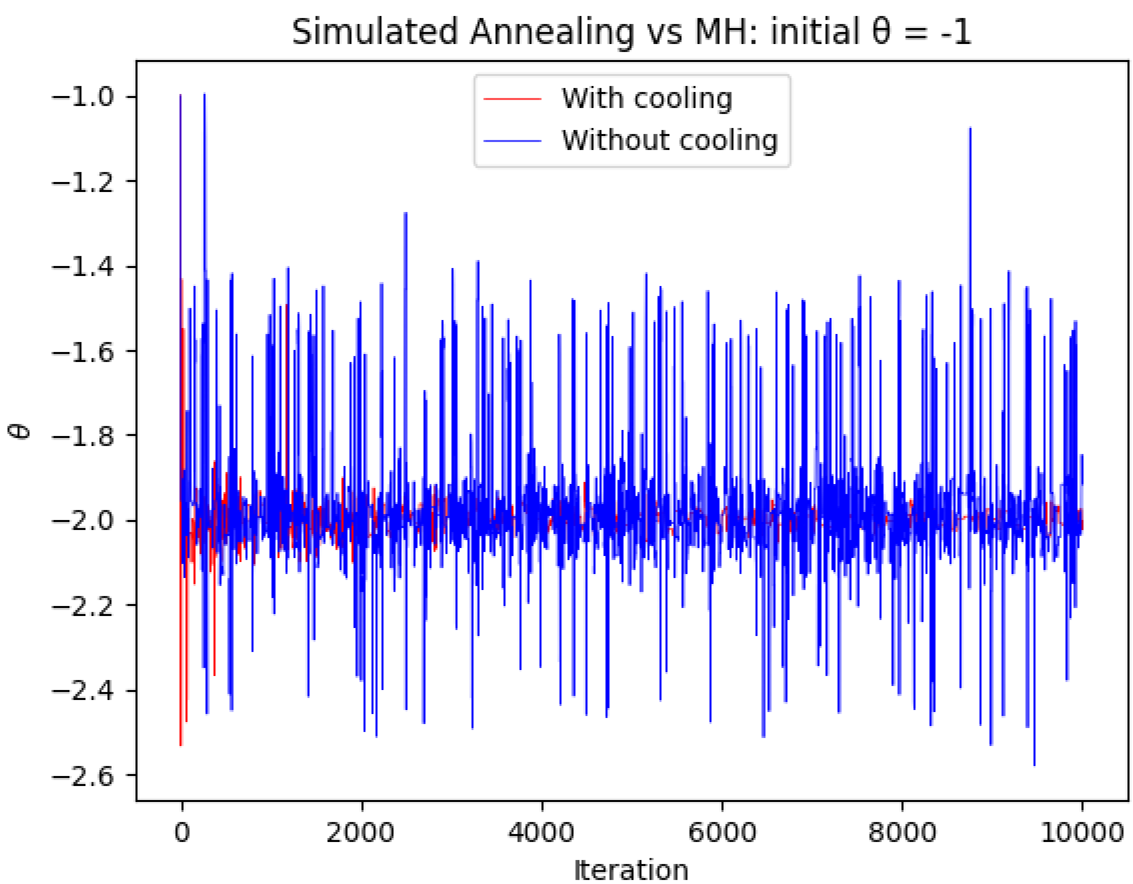

Metropolis-Hastings: β = 10, initial θ = -1
θ
−1.0
−1.2
−1.4
−1.6
−1.8
−2.0
0
2000
4000
6000
8000
10000
Iteration

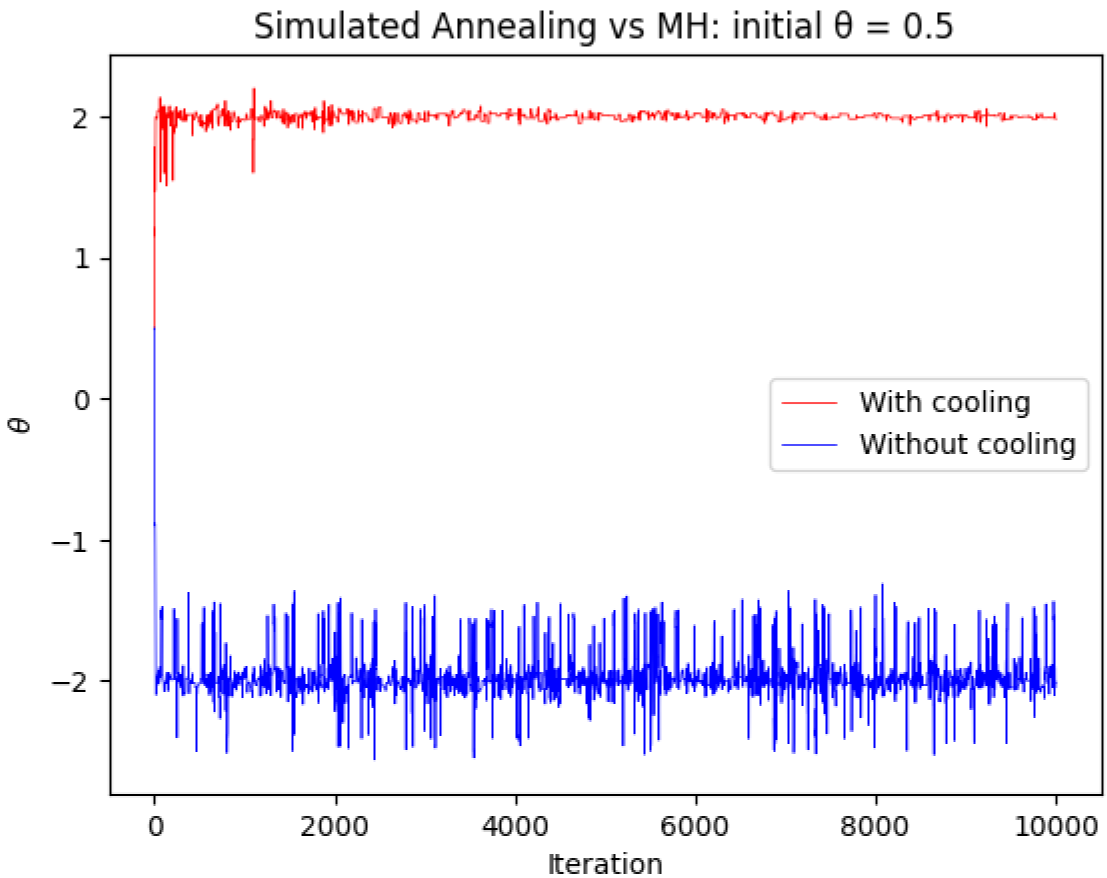


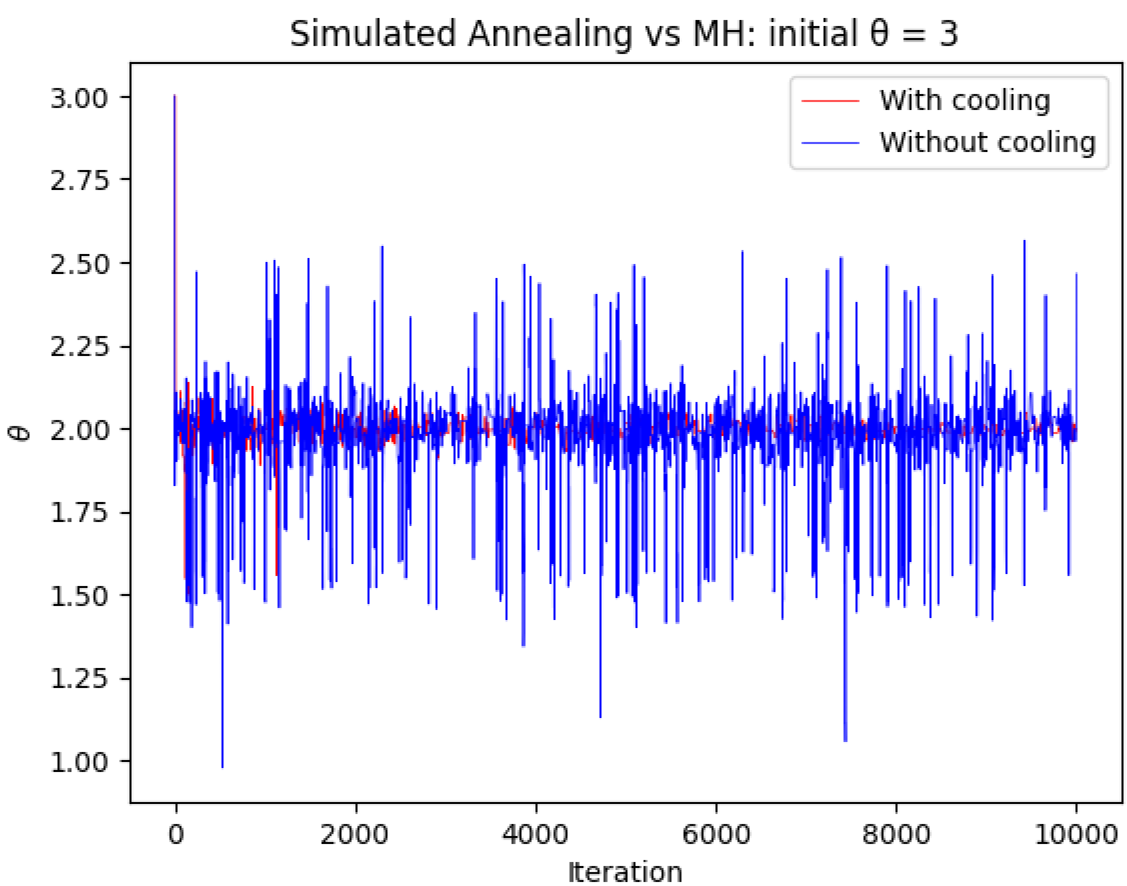


In [23]:
```
# Part c
def simulated_annealing(theta0, beta0, delta, sigma, iterations):
    theta = theta0
    beta = beta0
    chain = [theta]
    beta_chain = [beta]
    for i in range(iterations):
```

```
        theta_proposal = theta + np.random.normal(0, sigma)
        dH_val = H(theta_proposal) - H(theta)
        r = np.exp(-beta * dH_val)
        if r >= 1 or np.random.rand() < r:
            theta = theta_proposal
        chain.append(theta)
        beta += delta
        beta_chain.append(beta)
    return chain, beta_chain

beta0 = 1
delta = 0.001  # Cooling rate; tune as needed
iterations = 10000

for theta0 in initial_thetas:
    chain_sa, beta_chain = simulated_annealing(theta0, beta0, delta, sigma, iterations)
    plt.figure()
    plt.plot(chain_sa, 'r-', lw=0.5, label='With cooling')
    # For comparison, run the standard MH with constant beta (e.g., beta = beta0)
    chain_mh = metropolis_hastings(theta0, beta0, sigma, iterations)
    plt.plot(chain_mh, 'b-', lw=0.5, label='Without cooling')
    plt.xlabel('Iteration')
    plt.ylabel(r'$\theta$')
    plt.title(f'Simulated Annealing vs MH: initial θ = {theta0}')
    plt.legend()
    plt.show()
```




Simulated Annealing vs MH: initial θ = 0.5
2
1
0
−1
−2
θ
0
2000
4000
6000
8000
10000
Iteration
With cooling
Without cooling


Simulated Annealing vs MH: initial θ = 3
3.00
2.75
2.50
2.25
2.00
1.75
1.50
1.25
1.00
θ
0
2000
4000
6000
8000
10000
Iteration
With cooling
Without cooling

In [ ]: