Hamiltonian Monte Carlo (HMC)

🗓️Date: 03-29-2023
🕐Time: 21:50

Bipin Koirala

Table of Contents

Introduction

#Hamiltonian-Monte-Carlo or #Hybrid-Monte-Carlo (HMC) is a sampling technique that improves upon traditional MCMC methods such as the #Metropolis-Hasting algorithm by incorporating ideas from Hamiltonian dynamics to explore the target distribution more efficiently.

Hamiltonian Dynamics

#Hamiltonian-Dynamics is a formalism in classical mechanics that provides a powerful framework for understanding the time evolution of a physical system. #Hamiltonian-Dynamics is an alternative to the more familiar #Newtonian mechanics and #Lagrangian mechanics. In #Hamiltonian-Dynamics , a physical system is described in terms of generalized coordinates (q) and their conjugate momenta (p). The **Hamiltonian** (H) is a scalar function of these coordinates and momenta, which represents the total energy of the system.

Hamiltonian dynamics operates on a $p, q \in \mathbb{R}^d$. The time evolution of the system is governed by Hamilton's equations of motion, which are a set of first-order differential equations:

$$\begin{aligned} \frac{\partial q_i}{\partial t} &= \frac{\partial H}{\partial p_i} \\ \frac{\partial p_i}{\partial t} &= -\frac{\partial H}{\partial q_i} \end{aligned} \tag{1}$$

$\forall\, i = 1, 2, \ldots d$. For any time interval of s, these equations define a mapping, T_s, from the state at any time, t, to the state at time $t + s$. H, T_s are assumed to not depend on t. These equations describe how the coordinates and momenta change over time, based on the partial derivatives of the Hamiltonian with respect to the corresponding variables.

> **Practical Interpretation**
>
> This above system can be visualized the dynamics as that of a frictionless ball rolling over a surface of varying height. In 2D, the state of this system consists of the *position* of the ball, given by a 2D vector q, and its *momentum* given by a 2D vector p. The potential energy $U(q)$ of the ball is proportional to its height on the surface at current position, and its kinetic energy $K(p)$ is given by $\frac{|p|^2}{2m}$ where m is the mass of the ball.

When p, q are combined into a single vector $z = \begin{bmatrix} q \\ p \end{bmatrix} \in \mathbb{R}^{2d}$; we can re-write equation (1) as

$$\frac{dz}{dt} = J\, \nabla H(z) \tag{2}$$

where, $J = \begin{bmatrix} 0_{d\times d} & I_{d\times d} \\ -I_{d\times d} & 0_{d\times d} \end{bmatrix}$ and $\nabla H(z) = \begin{bmatrix} \vdots \\ \partial H/\partial z_k \\ \vdots \end{bmatrix}$

Properties of Hamiltonian Dynamics

- **Hamiltonian Dynamics is reversible**. The mapping T_s from any state at time t i.e $(q(t), p(t))$ to the state at time $t + s$ i.e. $(q(t + s), p(t + s))$ has one-to-one corresponding and hence has an inverse T_s^{-1}.
- **Conservation of Hamiltonian**. Hamiltonian dynamics conserves the Hamiltonian H i.e. $\frac{dH}{dt} = 0$. Hamiltonian is invariant.
- **Volume Preservation**. If we apply the mapping T_s to the points in some region $R \in \mathbb{R}^{q\times p}$ with volume V then the image of R under T_s will also have the same volume V.

- **Symplectic**. Suppose B_s is the Jacobian matrix of the mapping T_s then the following condition is satisfied

$$B_s^T J^{-1} B_s = J^{-1}$$

  This implied volume conservation and sympletic condition is stronger than volume preservation.

Reversibility, volume preservation, and sympleticness can be maintained exactly even when the Hamiltonian dynamics is approximated.

Hamiltonian Dynamics in MCMC

From a #Bayesian-Inference perspective, we are looking to sample from the posterior distribution $\mathbb{P}(\theta \mid y, X)$ and the position vector (q) in this setting corresponds to the model parameter (θ) and the momenta term (p) are auxiliary variables associated with each parameter so as to facilitate the efficient exploration of the parameter space.

For #Hamiltonian-Monte-Carlo, we usually use Hamiltonian functions that can be written as follows:

$$H(\theta, p) = U(\theta) + K(p) \tag{3}$$

where, $U(\theta)$ is called the potential energy of the system and is defined to be the negative log probability density of the distribution for θ (that we wish to sample from) plus some constant that is convenient. $K(p)$ is called the kinetic energy of the system.

$$\begin{aligned} U(\theta) &= -\log \mathbb{P}(\theta \mid y, X) \\ K(p) &= \frac{1}{2} p^T M^{-1} p \end{aligned} \tag{4}$$

Here, M is a symmetric, positive-definite #mass-matrix which is typically diagonal and is often a scalar multiple of the identity matrix.

Recall

Notice that the form of $K(p)$ corresponds to the negative log p.d.f of a zero-mean Gaussian distribution with covariance matrix M.

Similar to log-marginal-likelihood for a Gaussian Process.

$$\log \mathbb{P}(y \mid X, \theta) = -\frac{1}{2} y^T K_y^{-1} y - \frac{1}{2} \log \mid K_y \mid - \frac{n}{2} \log 2\pi$$

With these forms, the Hamiltonian equations in (1) can be re-written as;

$$\begin{aligned} \frac{\partial \theta_i}{\partial t} &= [M^{-1} p]_i \\ \frac{\partial p_i}{\partial t} &= \frac{\partial U}{\partial \theta_i} \end{aligned} \tag{5}$$

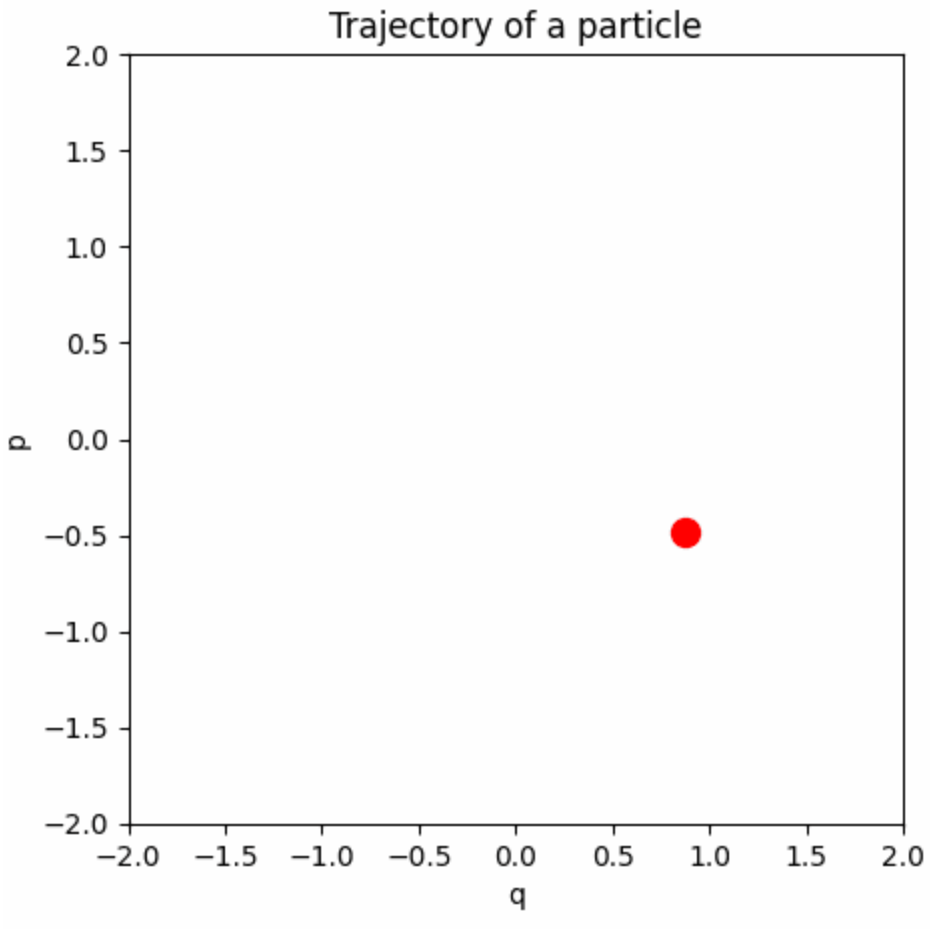
Hamiltonian Dynamics in 1D

Let, $U(q) = q^2/2$ and $K(p) = p^2/2$ then the solution to the Hamiltonian equations are:

$$q(t) = r \cos(a + t), \quad p(t) = -r \sin(a + t)$$

Visualize the Dynamics

For a fixed r and phase a, the above dynamics looks something like this (in PDF only the first frame of GIF is rendered as a static image):



Discretization of Hamilton's Equation - The Leapfrog

When implementing MCMC using Hamiltonian dynamics, there is an inherent need to discretize the time such that Hamilton's equations in (1) can be approximated. We take some small step-size ϵ and then iteratively compute the states at times $\epsilon, 2\epsilon, 3\epsilon, \ldots$ starting with the state at time zero.

Assume that $K(p) = p^T M^{-1} p$ and m_i is the i^{th} diagonal element of matrix M. Let's look at three types of methods. 👀

Euler's Method

This is easiest way to approximate the solution to D.E. In our case, we perform the following operations for each index of position and momentum vector.

$$p_i(t+\epsilon) = p_i(t) + \epsilon \frac{dp_i}{dt} = p_i(t) - \epsilon \frac{\partial U}{\partial q_i}(q(t))$$
$$q_i(t+\epsilon) = q_i(t) + \epsilon \frac{dq_i}{dt} = q_i(t) + \epsilon \frac{p_i(t)}{m_i}$$

Modification to Euler's Method

We first compute i^{th} momentum variable and simply use this when computing the corresponding position variable.

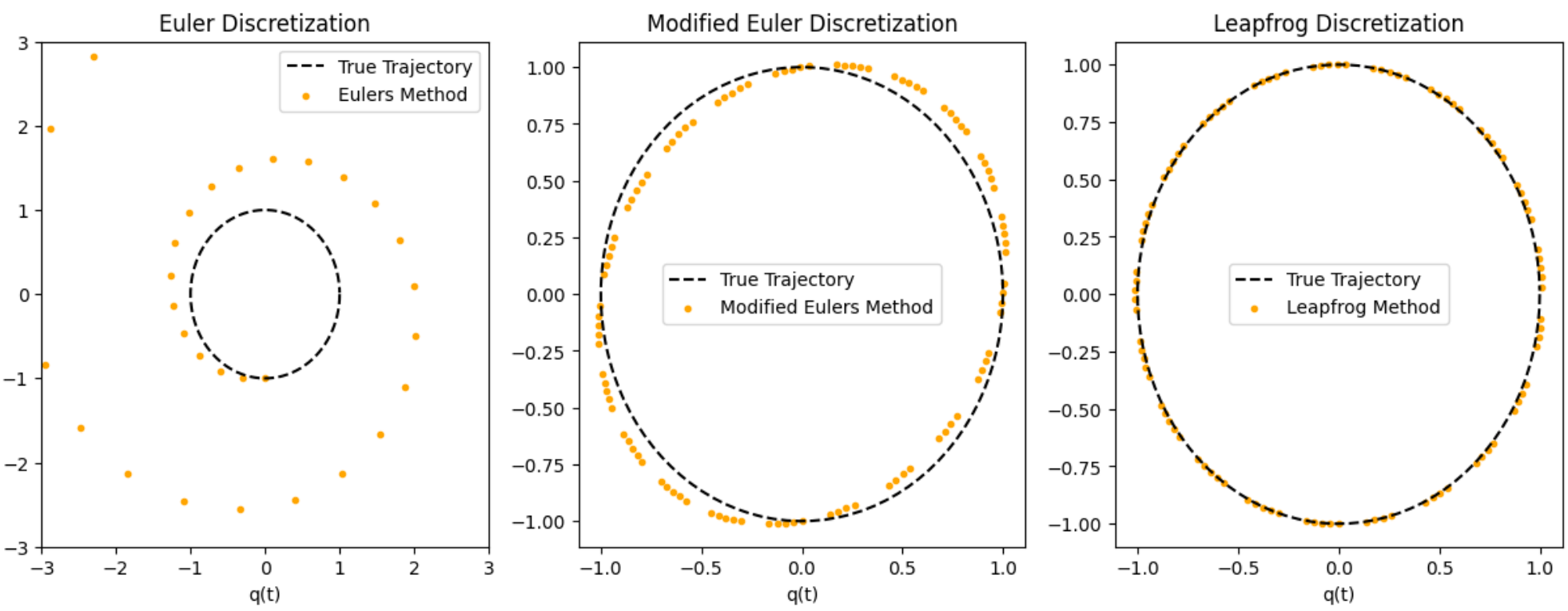
$$p_i(t+\epsilon) = p_i(t) - \epsilon \frac{\partial U}{\partial q_i}(q(t))$$
$$q_i(t+\epsilon) = q_i(t) + \epsilon \frac{p_i(t+\epsilon)}{m_i}$$

Leapfrog

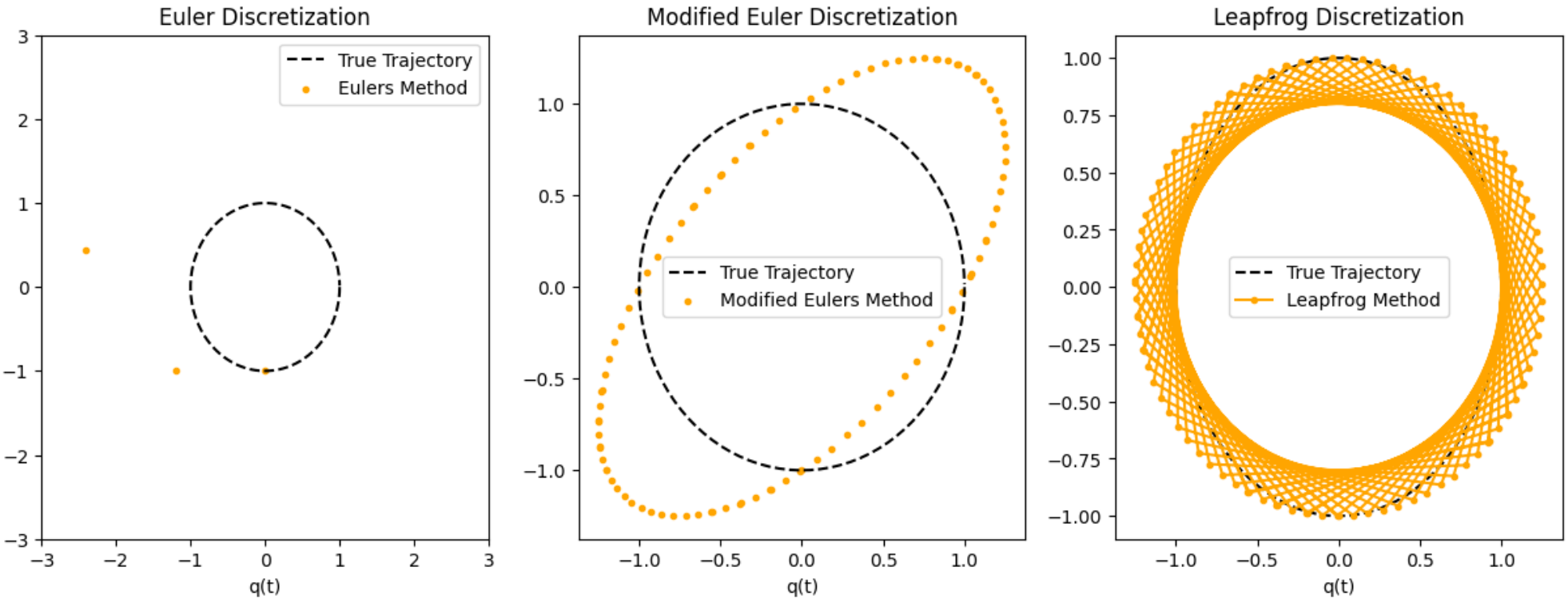
Better result is obtained with this method as compared to the results from above two methods. The idea 💡 is to use half step when calculating i^{th} momentum variable, then use it to update corresponding position variable at full step following by updating the same variable using latest position variable.

$$p_i(t+\epsilon/2) = p_i(t) - (\epsilon/2) \frac{\partial U}{\partial q_i}(q(t))$$
$$q_i(t+\epsilon) = q_i(t) + \epsilon \frac{p_i(t+\epsilon/2)}{m_i}$$
$$p_i(t+\epsilon) = p_i(t+\epsilon/2) - (\epsilon/2) \frac{\partial U}{\partial q_i}(q(t+\epsilon))$$

Below is a visualization for state space representation of the trajectories obtained via different time discretization methods when solving Hamiltonian dynamics in 1-D as discussed earlier. Here $\epsilon = 0.3$



When step-size is increased, these trajectories are less accurate. However, even when ϵ is large, say 1.2, Leapfrog method achieves better performance as compared to others methods. See the figure below.




```
import matplotlib.pyplot as plt

import numpy as np


r = 1
a = 0



# Plot Trajectory
t = np.linspace(0,6.268,100)
x = np.zeros(t.shape[0])
y = np.zeros(t.shape[0])

for i in range(len(t)):
        x[i], y[i] = r*np.cos(a + t[i]), -r*np.sin(a + t[i])

fig = plt.figure(figsize=(15,5))

ax1 = fig.add_subplot(1,3,1)
ax1.plot(x,y, label = 'True Trajectory', linestyle = '--', color = 'black')
ax1.set_xlabel('q(t)')
ax1.set_xlim(-3, 3), ax1.set_ylim(-3, 3)
ax1.set_title('Euler Discretization')


# Euler's Method:


def U(q):
        return 0.5*q**2

def dUdq(q):
        return q

m = 1
```

```
ϵ = 1.2
num_steps = 1000
p_e, q_e = np.zeros(num_steps), np.zeros(num_steps)
p_e[0] = -1
q_e[0] = 0

for t in range(1, num_steps):
        p_e[t] = p_e[t-1] - ϵ * dUdq(q_e[t-1])
        q_e[t] = q_e[t-1] + ϵ * p_e[t-1]/m

ax1.scatter(q_e,p_e, alpha = 1, label = 'Euler''s Method', linestyle='-', marker = '.', color = 'orange')
plt.legend()




# Modified Euler:


num_steps = 100
p_m, q_m = np.zeros(num_steps), np.zeros(num_steps)
p_m[0] = -1
q_m[0] = 0

for t in range(1, num_steps):
        p_m[t] = p_m[t-1] - ϵ * dUdq(q_m[t-1])
        q_m[t] = q_m[t-1] + ϵ * p_m[t]/m

ax2 = fig.add_subplot(1,3,2)
ax2.plot(x,y, label = 'True Trajectory', linestyle = '--', color = 'black')
ax2.set_xlabel('q(t)')
ax2.set_title('Modified Euler Discretization')
ax2.scatter(q_m,p_m, alpha = 1, label = 'Modified Euler''s Method', linestyle='-', marker = '.', color = 'orange')
plt.legend()



# Leap-frog


num_steps = 100
p_l, q_l = np.zeros(num_steps), np.zeros(num_steps)
p_l[0] = -1
q_l[0] = 0


for t in range(1, num_steps):
        p_l[t] = p_l[t-1] - (ϵ/2) * dUdq(q_l[t-1])
        q_l[t] = q_l[t-1] + ϵ * p_l[t]/m
        p_l[t] = p_l[t] - (ϵ/2) * dUdq(q_l[t])


ax3 = fig.add_subplot(1,3,3)
ax3.set_title('Leapfrog Discretization')
ax3.set_xlabel('q(t)')
ax3.plot(x,y, label = 'True Trajectory', linestyle = '--', color = 'black')
ax3.plot(q_l, p_l, alpha = 1, label = 'Leapfrog Method', marker = '.', color = 'orange')
plt.legend()
plt.show()
```

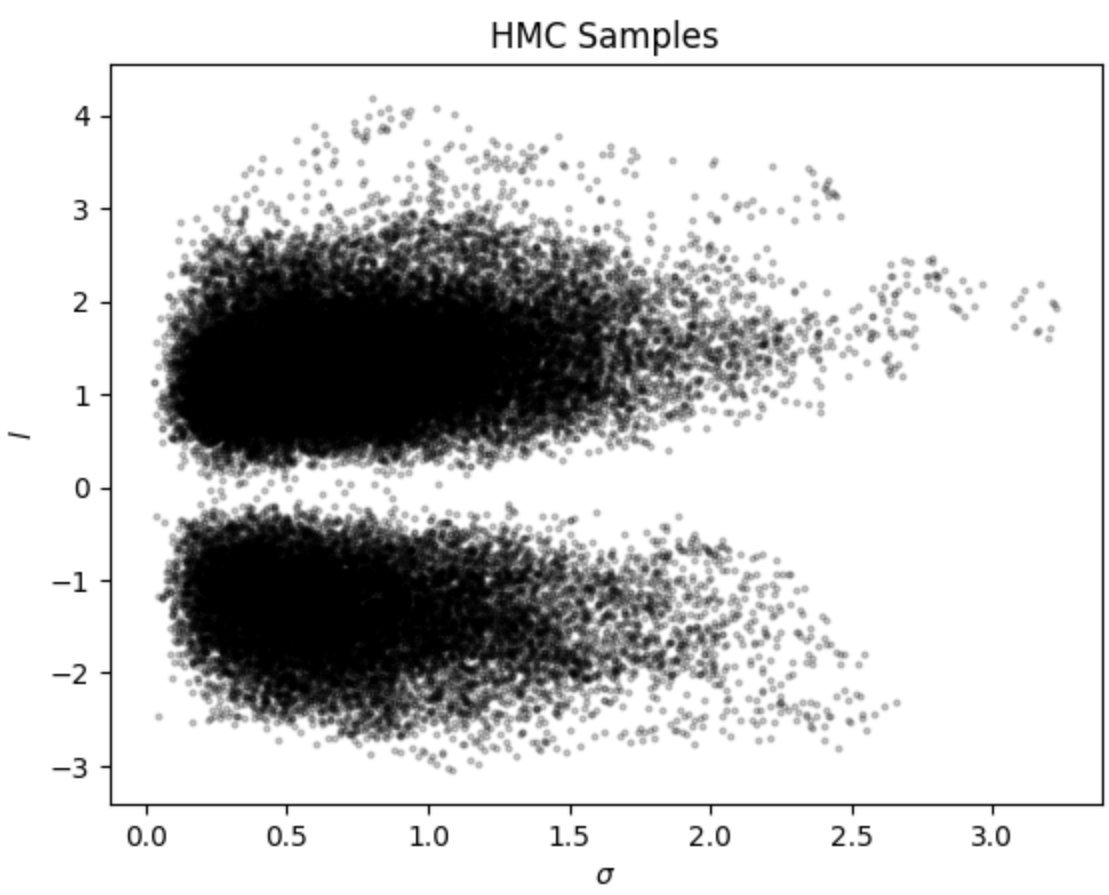
HMC Algorithm

HMC Algorithm

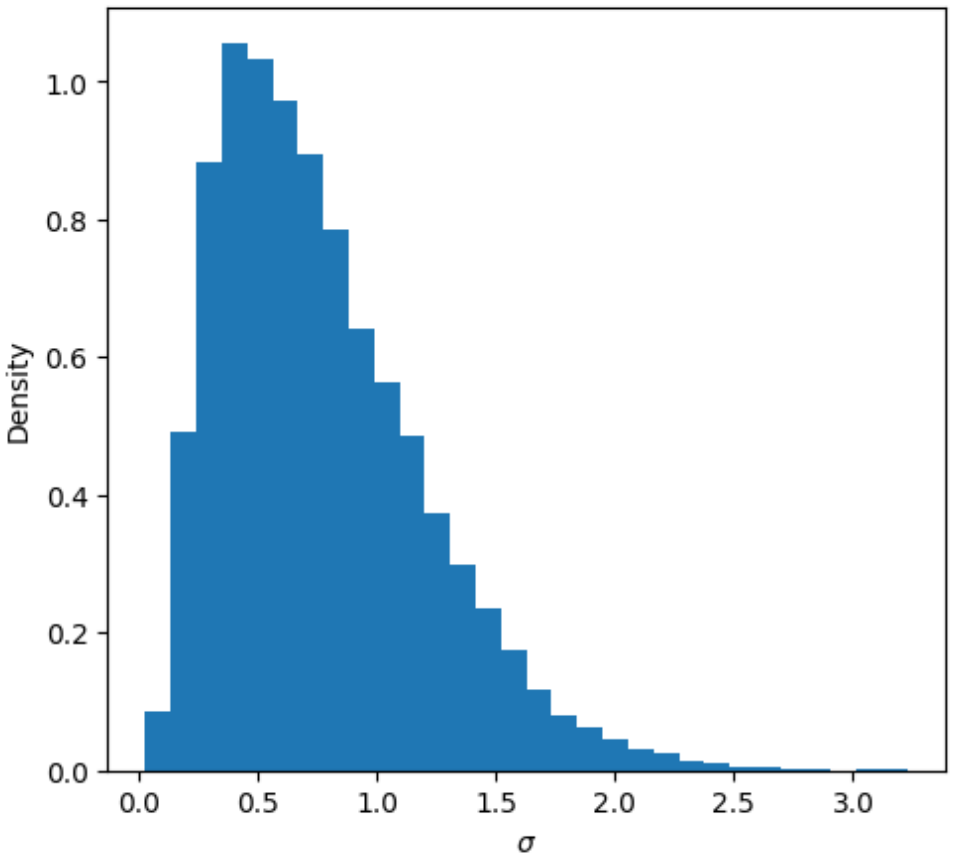
- Draw new values for each momentum variable from $\mathcal{N}(0,1)$. $P \in \mathbb{R}^d$
- Perform Metropolis update using Hamiltonian dynamics to propose a new state
  - start with the current state (q, p)
  - simulate Hamiltonian Dynamics for L steps using Leapfrog Technique
  - momentum variables at the end of L-step trajectory is negated, which gives the proposed state (q^*, p^*)
- Acceptance Probability: $\min(1, \exp(U(q) - U(q^*) + K(p) - K(p^*)))$

For a test dataset (X, y) where $X \sim \text{Unif}(-4, 4)$ and $y = \sin(X)$, performing HMC sampling technique gives the following results on the hyper-parameter for R.B.F kernel.

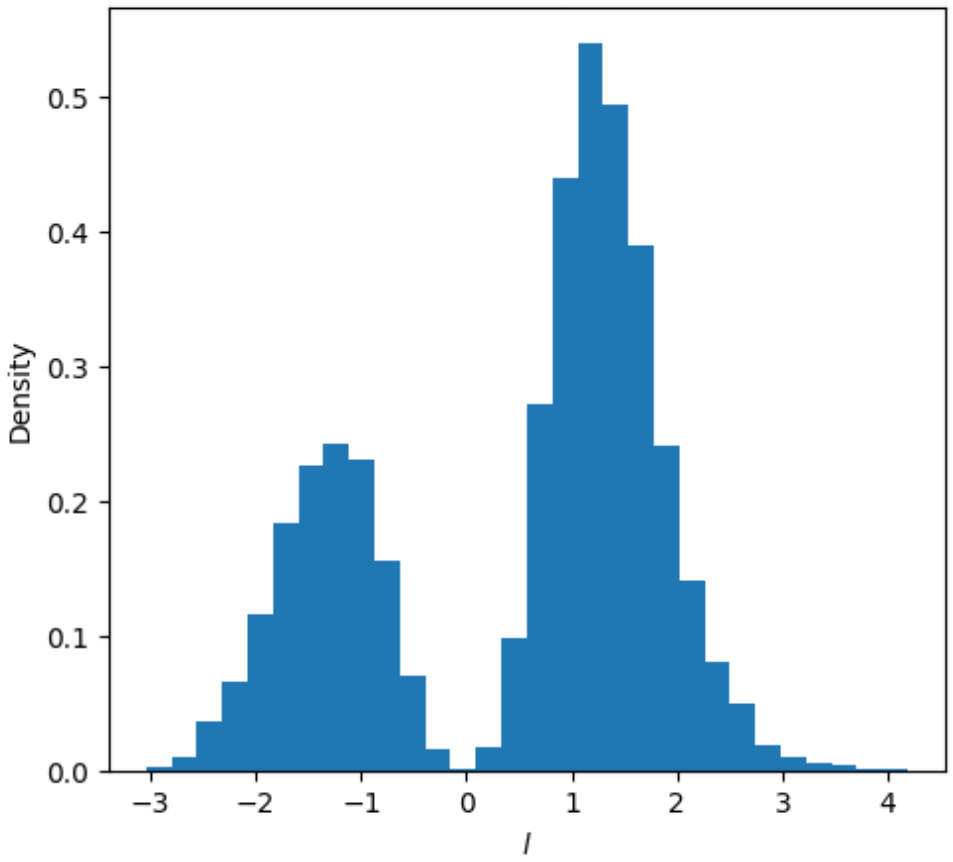
The figure below is based on the samples obtained via HMC on the above dataset.



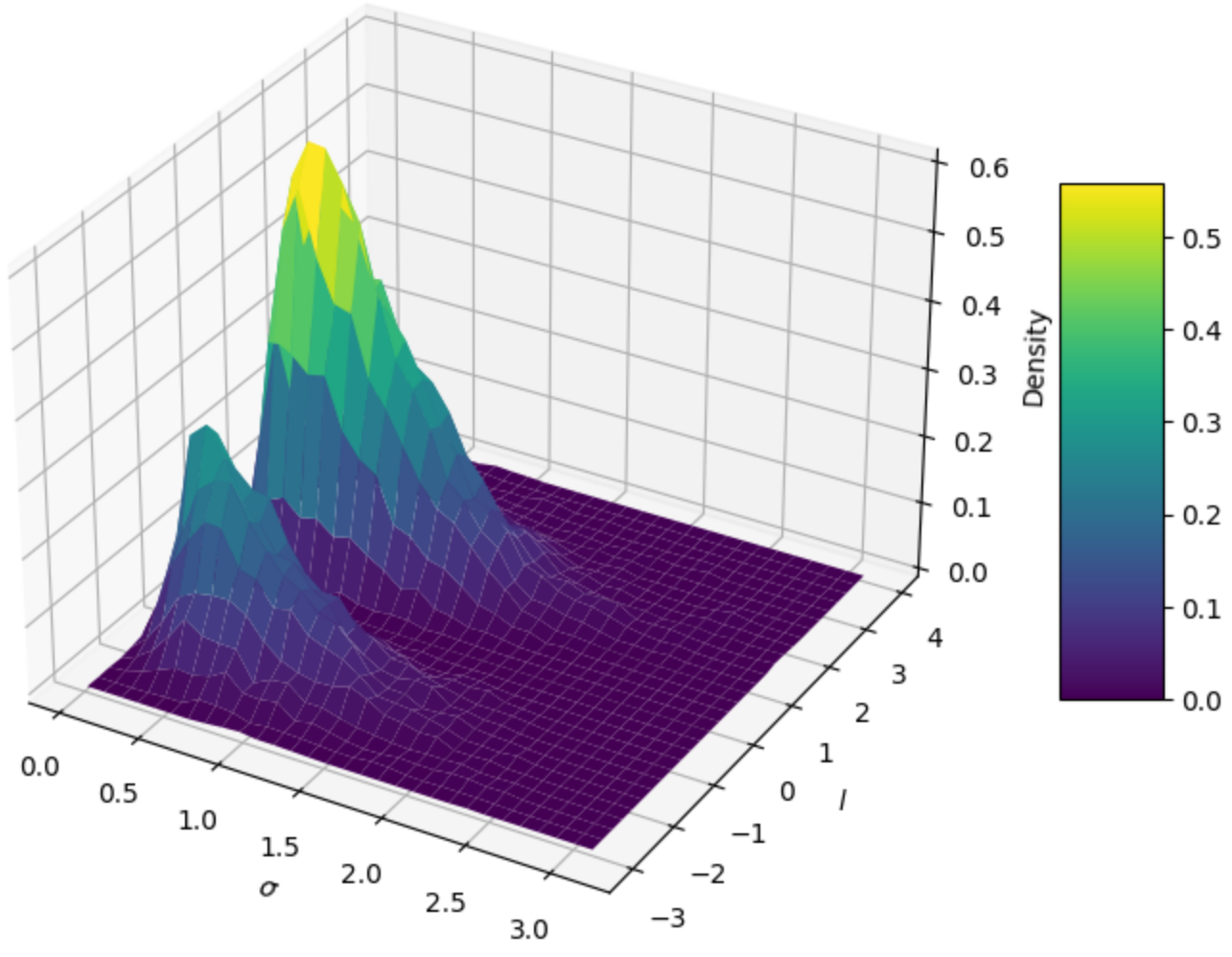
Sample Standard Deviation



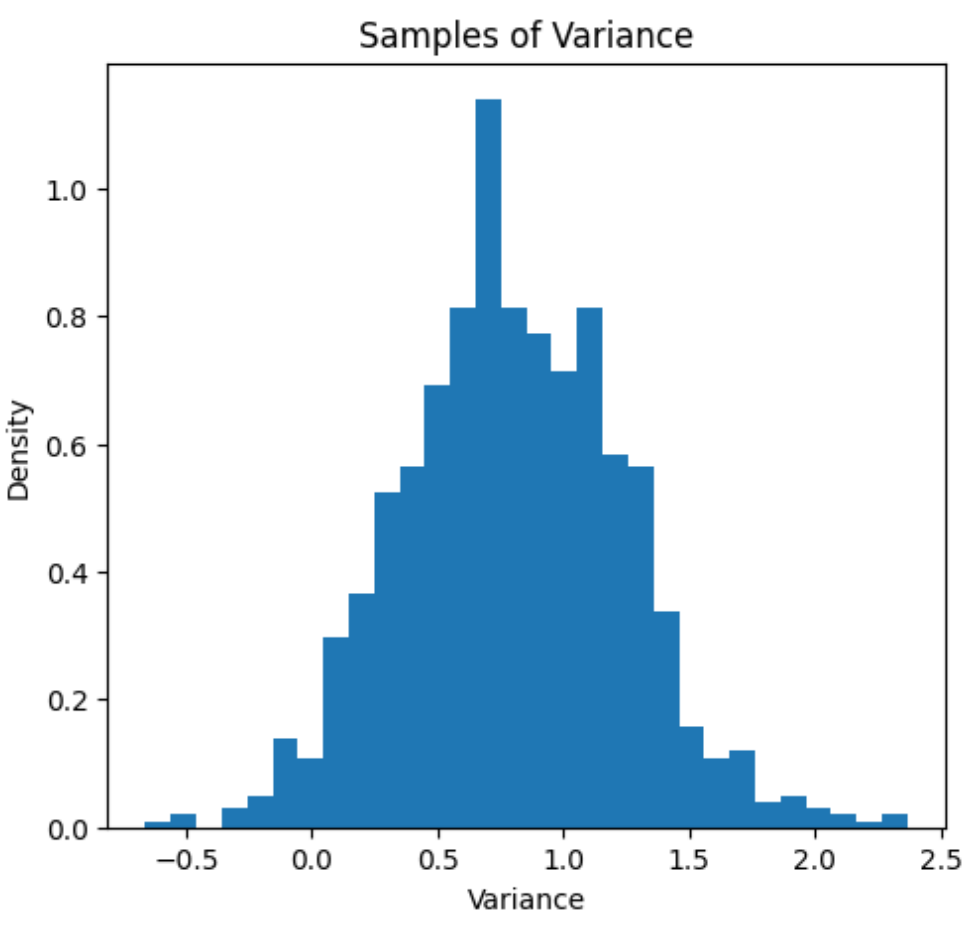
Sample Lengthscale

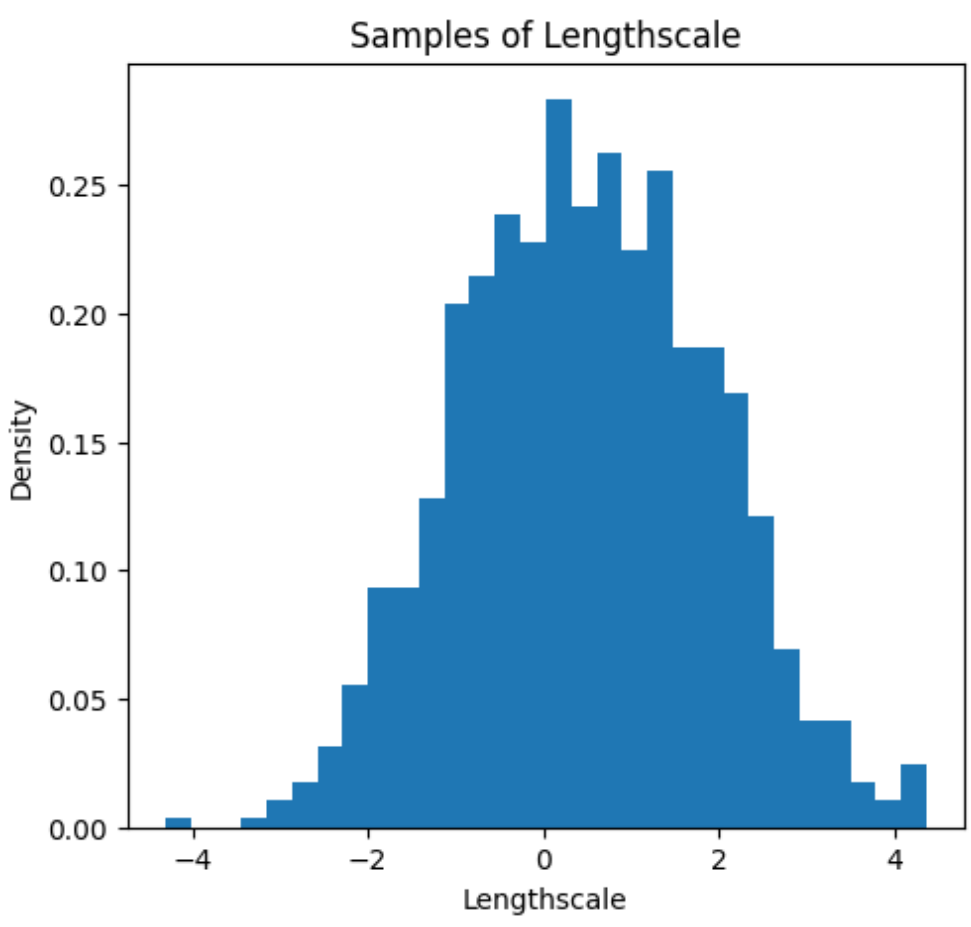


Posterior Distribution of Sampled Kernel Parameters

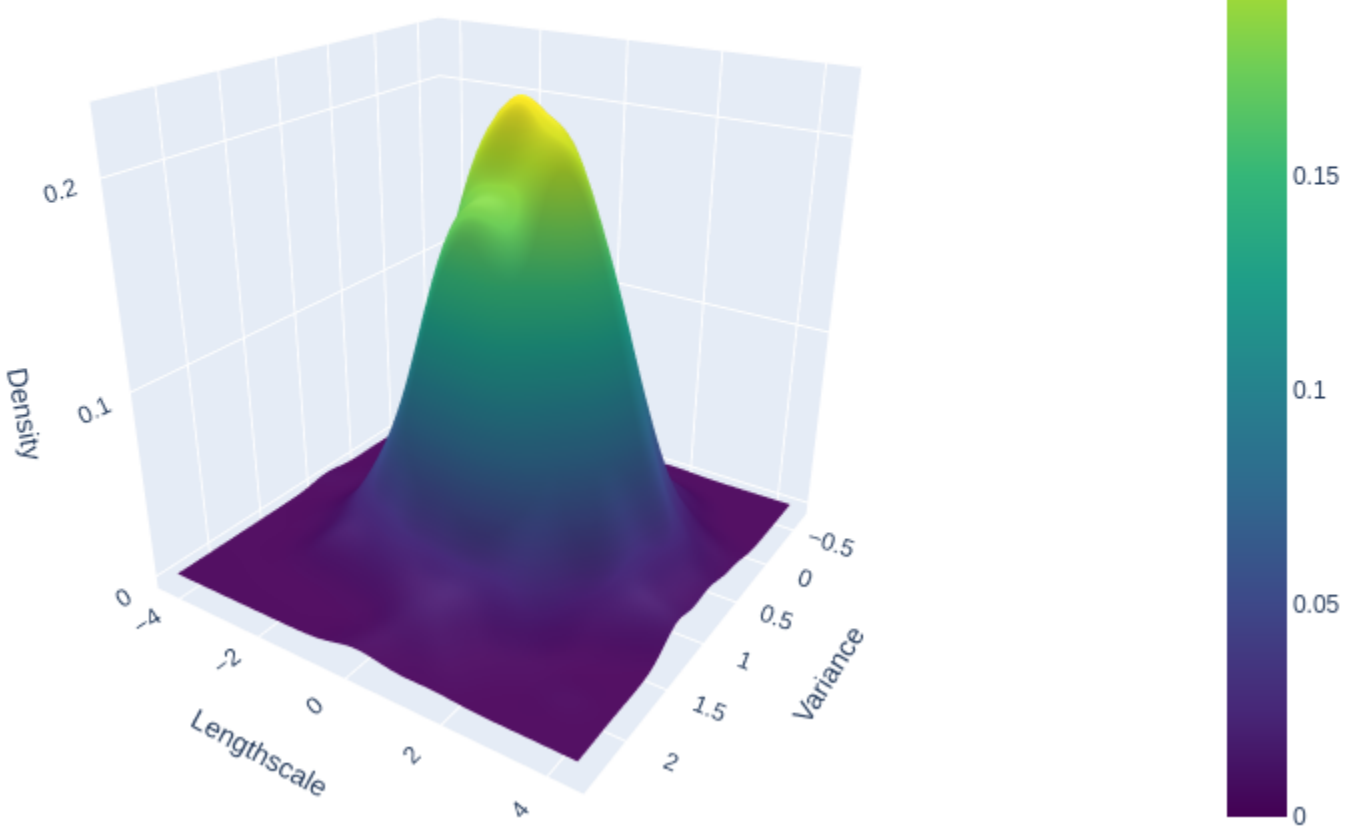


Now, new samples can be obtained from these estimates of mean and variance for each kernel parameters i.e. sampled from posterior distribution.





Joint pdf of Kernel Parameters



Convergence Diagnostic

#Gelman-Rubin convergence diagnostic, also known as the potential scale reduction factor (PSRF), is a method for accessing the convergence of multiple MCMC chains. If we were to start multiple parallel chains in many different starting values, the theory claims that they should all eventually converge to the stationary distribution. So after some amount of time, it should be impossible to distinguish between the multiple chains. They should all “look” like the stationary distribution. One way to assess this is to compare the variation between chains to the variation within the chains. If all the chains are “the same”, then the between chain variation should be close to zero.

Algorithm

1. Run multiple HMC chains (e.g. 3-4 chains) with different initial values for the parameters θ
2. Discard certain number of initial samples as burn-in and there is now L samples
3. Compute the within-chain variance (W) and between-chain variance (B) for each parameter
4. Compute PSRF value:

$$R = \frac{\frac{L-1}{L}W + \frac{1}{L}B}{W}$$

Here, step 3 can be computed as follows; Suppose there are a total of J number of chains

$$
\begin{aligned}
\bar{x}_j &= \frac{1}{L}\sum_{t=1}^{L} x_t^{(j)} \quad \text{(chain mean)} \\
\bar{x} &= \frac{1}{J}\sum_{j=1}^{J} \bar{x}_j \quad \text{(global mean)} \\
B &= \frac{L}{J-1}\sum_{j=1}^{J}(\bar{x}_j - \bar{x})^2 \quad \text{(between-chain variance)} \\
s_j^2 &= \frac{1}{L-1}\sum_{t=1}^{L}(x^{(j)} - \bar{x}_j)^2 \quad \text{(within-chain variance)} \\
W &= \frac{1}{J}\sum_{j=1}^{J} s_j^2
\end{aligned}
$$

Code: 💻

```
import jax.numpy as jnp
import numpy as np

num_samples = 5000
num_warmup = 1000

def gelman_rubin(chains):

        num_chains, num_samples, num_params = chains.shape
        mean_per_chain = jnp.mean(chains, axis=1)
        global_mean = jnp.mean(mean_per_chain, axis=0)

        B = num_samples / (num_chains - 1) * jnp.sum((mean_per_chain - global_mean) ** 2, axis=0)
        W = jnp.mean(jnp.var(chains, axis=1, ddof=1), axis=0)

        # Calculate variance estimate
        var = (1 - 1 / num_samples) * W + (1 / num_samples) * B

        # Compute the potential scale reduction factor (R-hat)
        R_hat = jnp.sqrt(var / W)

        return R_hat


num_chains = 4
chains = []


for i in range(num_chains):
        # Run HMC with different initial values for the parameters
        σ = np.absolute(np.random.normal())*posterior_std[0] + posterior_mean[0]
        l = np.absolute(np.random.normal())*posterior_std[1] + posterior_mean[1]
        init_hyperparams = jnp.array([σ, l])

        samples = hmc_sampling(init_hyperparams, noise_variance, X, y, m, epsilon, L, num_samples, num_warmup)
        chains.append(samples)

chains = jnp.stack(chains)
R_hat = gelman_rubin(chains)

print("R-hat values:", R_hat)
```

Output:

```
R-hat values: [1.0000557 1.0030391]
```

Code for HMC in MCMC

```
from jax.scipy.stats import multivariate_normal
import jax.numpy as jnp
from jax import grad, jit, vmap
import jax.random as random


@jit
def kernel(X1, X2, theta):
        """
        Computes the RBF kernel matrix between two sets of input vectors X1 and X2.
        Args:
        X1: A numpy array of shape (n1, d), representing the first set of input vectors.
        X2: A numpy array of shape (n2, d), representing the second set of input vectors.
```

```
        lengthscale: The lengthscale parameter of the RBF kernel.
        variance: The variance parameter of the RBF kernel.

        Returns:
        A numpy array of shape (n1, n2), representing the RBF kernel matrix.
        """
        variance, lengthscale = theta
        dist_sq = jnp.sum(X1**2, axis=1).reshape(-1, 1) + jnp.sum(X2**2, axis=1) - 2*jnp.dot(X1, X2.T)
        K = variance * jnp.exp(-0.5 * dist_sq / (lengthscale ** 2))
        return K


@jit
def unnormalized_posterior(hyperparameters, noise_variance, X, y):
        """
        Computation of the unnormalized prior.
        :param Measurementplane self:
        An instance of the measurement plane.
        :param numpy array xo:
        A numpy array of priors.
        """
        Kxx = kernel(X, X, hyperparameters)
        Z = Kxx + noise_variance * jnp.eye(y.shape[0])
        try:
                likelihood = multivariate_normal.logpdf(y, mean=jnp.zeros((len(y))), cov=Z)
        except FloatingPointError as e:
                print('got here!')
                return -jnp.inf

        # Sample from the prior distribution!
        prior_alpha = multivariate_normal.logpdf(hyperparameters, mean=jnp.zeros((2,)), \

cov=jnp.eye(2)*1.0)
        unnormalized_posterior = likelihood + prior_alpha
        return unnormalized_posterior


@jit
def Uθ(hyperparameters, noise_variance, X, y):
        return -unnormalized_posterior(hyperparameters, noise_variance, X, y)


@jit
def dUdθ(hyperparameters, noise_variance, X, y):
        return grad(Uθ)(hyperparameters, noise_variance, X, y)


@jit
def kinetic_energy(p,m):
        return 0.5*jnp.sum( (p**2) /m)


@jit
def leapfrog_step(q, p, m, ϵ, noise_variance, X, y):
        p_half = p - 0.5 * ϵ * dUdθ(q, noise_variance, X, y)
        q_new = q + (ϵ/m) * p_half
        p_new = p_half - 0.5 * ϵ * dUdθ(q_new, noise_variance, X, y)
        return q_new, p_new


def hmc_iteration(key, q, L, m, ϵ, noise_variance, X, y):
        key, subkey = random.split(key)
        # draw new vector for momentum variables
        p = random.normal(key, shape = q.shape)

        # start with current state
        q_new, p_new = q, p

        # simulate Hamiltonian Dyanmics for L steps using Leap
        for _ in range(L):
                q_new, p_new = leapfrog_step(q_new, p_new, m, ϵ, noise_variance, X, y)


        # take the negative of momentum vector at the end of L-step trajectory (Radford Neil MCMC for H.D)
        p_new = -p_new

        K_new, K_old = kinetic_energy(p_new,m), kinetic_energy(p, m)
        U_new, U_old = Uθ(q_new, noise_variance, X, y), Uθ(q, noise_variance, X, y)


        # NOTE: There is a subtle difference between min and minimum.
        accept_reject_prob = jnp.minimum(1, jnp.exp(K_old + U_old - K_new - U_new))
```

```
        if random.uniform(subkey) <= accept_reject_prob:
                q = q_new

        return q, key


def hmc_sampling(init_hyperparams, noise_variance, X, y, m, ϵ, L, num_samples, num_warmup):
        samples = []
        key = random.PRNGKey(0)
        q = init_hyperparams


        for i in range(num_samples + num_warmup):
                q, key = hmc_iteration(key, q, L, m, ϵ, noise_variance, X, y)

        if i >= num_warmup:
                samples.append(q)

        return jnp.array(samples)
```

Test the sampling method

```
init_hyperparams = jnp.array([1.0, 1.0]) # initial values for the hyperparameters
m = 1.0 # mass
epsilon = 0.01 # step size
L = 10 # number of Leapfrog steps
num_samples = 50000
num_warmup = 10000
noise_variance = 0.2

# Test data
rng_key = random.PRNGKey(0)
f_sin = lambda x: (jnp.sin(x)).flatten()
X = random.uniform(rng_key, minval= -4, maxval= 4, shape=(10,1))
y = f_sin(X)

samples = hmc_sampling(init_hyperparams, noise_variance, X, y, m, epsilon, L, num_samples, num_warmup)

posterior_mean = jnp.mean(samples, axis=0)
posterior_std = jnp.std(samples, axis=0)

print ('Mean of \u03B8:', posterior_mean)
print ('Std of \u03B8:', posterior_std)
```

Figure Plots

```
import matplotlib.pyplot as plt

plt.plot(samples[2000:,0], samples[2000:,1], 'ko', markersize=2, markerfacecolor='k', alpha=0.2)
plt.xlabel('$\sigma$')
plt.ylabel('$l$')
plt.title('HMC Samples')
plt.show()

# ------------------------------------------------

fig, axes = plt.subplots(1, 2, figsize=(12, 5))
axes[0].hist(samples[:, 0], bins=30, density=True)
axes[0].set_title('Sample Standard Deviation')
axes[0].set_xlabel('$\sigma$')
axes[0].set_ylabel('Density')

axes[1].hist(samples[:, 1], bins=30, density=True)
axes[1].set_title('Sample Lengthscale')
axes[1].set_xlabel('$l$')
axes[1].set_ylabel('Density')

plt.show()

# ------------------------------------------------

from mpl_toolkits.mplot3d import Axes3D

# Compute the 2D histogram
H, xedges, yedges = np.histogram2d(samples[:, 0], samples[:, 1], bins=30, density=True)

# Create meshgrid for the plot
X, Y = np.meshgrid(xedges[:-1], yedges[:-1])

fig = plt.figure(figsize=(10, 7))
```

```python
ax = fig.add_subplot(111, projection='3d')

# Plot the surface
surf = ax.plot_surface(X, Y, H.T, cmap='viridis', linewidth=0, antialiased=True)

ax.set_title('Posterior Distribution of Sampled Kernel Parameters')
ax.set_xlabel('$\sigma$')
ax.set_ylabel('$l$')
ax.set_zlabel('Density')


# Add a color bar to show the colormap scale
fig.colorbar(surf, ax=ax, shrink=0.5, aspect=5)

plt.show()

# ---------------------------------------------

# Sample from posterior

# Set a random seed for reproducibility
key = random.PRNGKey(0)

# Generate random samples for each parameter using the posterior mean and standard deviation
num_samples = 1000 # Number of samples to generate

# For variance (sigma)
key, subkey = random.split(key)
variance_samples = random.normal(subkey, shape=(num_samples,)) * posterior_std[0] + posterior_mean[0]

# For lengthscale (l)
key, subkey = random.split(key)
lengthscale_samples = random.normal(subkey, shape=(num_samples,)) * posterior_std[1] + posterior_mean[1]

# ---------------------------------------------

fig, axes = plt.subplots(1, 2, figsize=(12, 5))

axes[0].hist(variance_samples, bins=30, density=True)
axes[0].set_title('Samples of Variance')
axes[0].set_xlabel('')
axes[0].set_ylabel('Density')


axes[1].hist(lengthscale_samples, bins=30, density=True)
axes[1].set_title('Samples of Lengthscale')
axes[1].set_xlabel('Lengthscale')
axes[1].set_ylabel('Density')

plt.show()

# ---------------------------------------------


from mpl_toolkits.mplot3d import Axes3D
from scipy.stats import gaussian_kde


# Perform Gaussian kernel density estimation
kde = gaussian_kde(np.vstack([variance_samples, lengthscale_samples]))


# Create a grid for plotting
x = np.linspace(variance_samples.min(), variance_samples.max(), 100)
y = np.linspace(lengthscale_samples.min(), lengthscale_samples.max(), 100)
X, Y = np.meshgrid(x, y)

# Evaluate the KDE on the grid
Z = kde(np.vstack([X.ravel(), Y.ravel()])).reshape(X.shape)


fig = plt.figure(figsize=(10, 7))
ax = fig.add_subplot(111, projection='3d')


# Plot the surface
surf = ax.plot_surface(X, Y, Z, cmap='viridis', linewidth=0, antialiased=True)


ax.set_title('Smooth Distributions of Variance and Lengthscale')
ax.set_xlabel('Variance')
ax.set_ylabel('Lengthscale')
```

```
ax.set_zlabel('Density')



# Add a color bar to show the colormap scale
fig.colorbar(surf, ax=ax, shrink=0.5, aspect=5)

plt.show()

# ------------------------------------------------

import plotly.graph_objects as go


# Create a surface plot with the estimated probability density function
fig = go.Figure(data=[go.Surface(x=X, y=Y, z=Z, colorscale='Viridis')])
fig.update_layout(
        title='Joint pdf of Kernel Parameters',
        scene=dict(
                xaxis_title='Variance',
                yaxis_title='Lengthscale',
                zaxis_title='Density',
        ),
        autosize=False,
        width=800,
        height=600,
        margin=dict(l=20, r=20, b=20, t=80),
)

# Show the interactive plot
fig.show()
```