```
import numpy as np
import arviz as az
import pymc as pm
import math
import pickle

from learning_function_library import *
```

Definition of Functions for Analytical Solutions

```
def deg_to_rad(phi_deg):
    '''Converts angles from degrees to radians'''
    return phi_deg*np.pi/180

def rad_to_deg(phi_rad):
    '''Converts angles from radians to degrees'''
    return phi_rad*180/np.pi

def true_function_sbc(X, gamma = 15, B = 3, D = 2):
    '''Solves for the bearing capacity of a strip footing using
Meyerhof's equation.
    Returns vector/matrix of regression function values.

    Parameters
    ----------
    X : tuple of cohesion (kPa) and friction angle (degrees)
        element type: tuple; float

    gamma : unit weight (kN/m^3)
        element type : float

    B : width of strip footing (m)
        element type : float

    D : depth of embedment of strip footing (m)
        element type : float

    Returns
    -------
    q_u : bearing capacity (kPa)
        array of regression function values
    '''
    phi = X[:,0]
    c = X[:,1]

    phi_rad = deg_to_rad(phi)

    N_q =
np.exp(np.pi*np.tan(phi_rad))*np.tan(deg_to_rad(45+phi/2))**2
    N_c = (N_q-1)*(1/np.tan(phi_rad))
```

```
    N_g = (N_q-1)*np.tan(1.4*phi_rad)

    q = gamma*D
    q_u = c*N_c + q*N_q + (1/2)*B*gamma*N_g

    return q_u

def true_function_retaining_wall(X, gamma = 15):
    # Definition of wall dimension (m) and unit weight (kN/m^3)
    H = 5
    B = 3
    t = 0.6
    l_toe = 1
    l_heel = B - t - l_toe
    l_stem = H - t
    gamma_c = 23.5

    phi = X[:,0]
    c = X[:,1]

    c_interface = c/2
    phi_interface = phi - 5

    phi_rad = deg_to_rad(phi)
    phi_interface_rad = deg_to_rad(phi_interface)

    K_a = (1-np.sin(phi_rad))/(1+np.sin(phi_rad))
    P_a = (1/2) * gamma * K_a * H**2

    P_v1 = gamma*l_stem*l_heel
    P_v2 = gamma_c*t*l_stem
    P_v3 = gamma_c*t*B
    P_v = P_v1 + P_v2 + P_v3

    # FS for sliding
    P_R = P_v*np.tan(phi_interface_rad) + B*c_interface
    P_D = P_a
    FS_sliding = P_R/P_D

    # FS for overturning
    RM = P_v1*(B-l_heel/2) + P_v2*(l_toe+t/2) + P_v3*(B/2)
    OM = P_a*H/3
    FS_overturning = RM/OM

    # FS for bearing pressure
    R_y = P_v
    x = (RM-OM)/R_y
    e = np.abs(B/2-x)
```

```
    B_prime = B-2*e
    q_u = true_function_sbc(X, gamma, B_prime, 0)

    q_max = (R_y/B) * (1+6*e/B)
    FS_bp = q_u/q_max
    FS_bp[e > B/6] = 0

    FS = np.minimum(FS_sliding, FS_overturning)
    FS = np.minimum(FS, FS_bp)

    FS[FS<0] = 0

    return FS
```

Generation of Stochastic Input Variables

```
# get lognormal moments from normal moments
def get_lognormal_moments(mean, std):
    log_std = np.sqrt(np.log(1+(std/mean)**2))
    log_mean = np.log(mean) - log_std**2/2
    return [log_mean, log_std]

# sample from lognormal distribution within a range (used to apply
upper limit on friction angle)
def sample_lognormal_within_range(log_mean, log_std, low, high, size):
    samples = []
    while len(samples) < size:
        sample = np.random.lognormal(log_mean, log_std, size=1)
        if low <= sample <= high:
            samples.append(sample)
    return np.array(samples)

# define stochastic characteristics of inputs cohesion and friction
angle
N_pop = 1000000
N_val = 1000

phi_mean, phi_std = 20, 8
c_mean, c_std = 20, 5

phi_log_moments = get_lognormal_moments(phi_mean, phi_std)
c_log_moments = get_lognormal_moments(c_mean, c_std)

phi_pop = sample_lognormal_within_range(phi_log_moments[0],
phi_log_moments[1], 0, 50, N_pop)
c_pop = np.random.lognormal(c_log_moments[0], c_log_moments[1], N_pop)
X_pop = np.hstack((phi_pop.reshape(-1, 1), c_pop.reshape(-1, 1)))

phi_val = sample_lognormal_within_range(phi_log_moments[0],
phi_log_moments[1], 0, 50, N_val)
```

```
c_val = np.random.lognormal(c_log_moments[0], c_log_moments[1], N_val)

X_val = np.hstack((phi_val.reshape(-1, 1), c_val.reshape(-1, 1)))

N_initial_DoE = 5
random_indices = np.random.choice(N_pop, size=N_initial_DoE,
replace=False)
initial_X_DoE = X_pop[random_indices]
```

Generation of Initial DoE

```
true_function = true_function_retaining_wall
noise_scale = 0.1

Y_pop = true_function(X_pop) + np.random.normal(loc=0,
scale=noise_scale, size = len(X_pop))
Y_val = true_function(X_val) + np.random.normal(loc=0,
scale=noise_scale, size = len(X_val))

initial_Y_DoE = Y_pop[random_indices]

X_DoE = initial_X_DoE
Y_DoE = initial_Y_DoE

N_DoE = len(X_DoE)
```

Training with Initial DoE

```
with pm.Model() as GP_model:
    # Hyperparameters for the Gaussian Process
    ls_phi = pm.Uniform("ls_phi", lower=np.min(phi_pop)/100,
upper=np.max(phi_pop)*100)
    ls_c = pm.Uniform("ls_c", lower=np.min(c_pop)/100,
upper=np.max(c_pop)*100)
    cov_scale = pm.Uniform("cov_scale", lower=0.000001, upper=10)
    sigma = pm.Uniform("sigma", lower=0.000001, upper=10)

    # Mean functiom
    mean_func = pm.gp.mean.Zero()

    # Covariance function
    cov_func = cov_scale ** 2 * pm.gp.cov.Matern52(input_dim=2,
ls=[ls_phi, ls_c])

    # GP prior with zero mean
    gp = pm.gp.Marginal(mean_func = mean_func, cov_func = cov_func)

    # GP likelihood
    y_ = gp.marginal_likelihood("y_", X_DoE, Y_DoE, sigma)
```

```
    # obtain MAP estimate and calculate predictive mean and variance
using MAP estimate
    start = {'ls_phi': np.mean(phi_pop), 'ls_c': np.mean(c_pop),
'cov_scale': 1, 'sigma': 1}
    map_estimate = pm.find_MAP()

    mean_pop, var_pop = gp.predict(X_pop, map_estimate, diag=True)
    mean_val, var_val = gp.predict(X_val, map_estimate, diag=True)

# store relevant results
hyperparams_list = [map_estimate]
MSE_list = [MSE_solver(mean_val, Y_val)]
```

Enrichment

```
learning_function_type = 'var'

stale_iterations = 0
best_MSE = MSE_list[-1]

for iterations in range(N_pop-N_DoE):
    # solve the learning function values for each point in the
population set
    LFV = learning_function_MLE(p2_5_y_hat, p50_y_hat, p97_5_y_hat,
learning_function_type)

    reverse_sorted_indices = np.argsort(LFV)[::-1]

    # iterate through LFV values in descending order and select point
with highest LFV that is not yet in DoE
    for i in range(N_pop):
        index_max = reverse_sorted_indices[i]
        if X_pop[index_max] in X_DoE:
            continue
        else:
            x_new = X_pop[index_max]
            y_new = Y_pop[index_max]
            break

    X_DoE = np.vstack((X_DoE, np.atleast_2d(x_new)))
    Y_DoE = np.append(Y_DoE, y_new)

    with pm.Model() as GP_model:
        # Hyperparameters for the Gaussian Process
        ls_phi = pm.Uniform("ls_phi", lower=np.min(phi_pop)/100,
upper=np.max(phi_pop)*100)
        ls_c = pm.Uniform("ls_c", lower=np.min(c_pop)/100,
```

```
upper=np.max(c_pop)*100)
        cov_scale = pm.Uniform("cov_scale", lower=0.000001, upper=10)
        sigma = pm.Uniform("sigma", lower=0.000001, upper=10)

        # Mean functiom
        mean_func = pm.gp.mean.Zero()

        # Covariance function
        cov_func = cov_scale ** 2 * pm.gp.cov.Matern52(input_dim=2,
ls=[ls_phi, ls_c])

        # GP prior with zero mean
        gp = pm.gp.Marginal(mean_func = mean_func, cov_func =
cov_func)

        # GP likelihood
        y_ = gp.marginal_likelihood("y_", X_DoE, Y_DoE, sigma)

        start = {'ls_phi': np.mean(phi_pop), 'ls_c': np.mean(c_pop),
'cov_scale': 1, 'sigma': 1}
        map_estimate = pm.find_MAP()

        mean_pop, var_pop = gp.predict(X_pop, map_estimate, diag=True)
        mean_val, var_val = gp.predict(X_val, map_estimate, diag=True)

    hyperparams_list.append(map_estimate)
    MSE_list.append(MSE_solver(mean_val, Y_val))

    if MSE_list[-1] >= best_MSE:
        # Increment stale iterations counter
        stale_iterations += 1

    else:
        best_MSE = MSE_list[-1]
        stale_iterations = 0

    # Check if training should stop
    if stale_iterations >= 20 and iterations >= 100:
        break
```

Saving of Results

```
results_dict = {
    'X_pop': X_pop,
    'Y_pop': Y_pop,
    'X_val': X_val,
    'Y_val': Y_val,
    'X_DoE': X_DoE,
    'Y_DoE': Y_DoE,
```

```
    'initial_X_DoE': initial_X_DoE,
    'initial_Y_DoE': initial_Y_DoE,
    'hyperparams_list': hyperparams_list,
    'MSE_list': MSE_list
}

file_path = 'retaining_wall_MLE_data.pkl'

with open(file_path, 'wb') as file:
    pickle.dump(result_dict, file)
```