```
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
```

Loading of Stochastic Variables from MLE Code

```
N_pop = 1000000

# Load the dictionary of inputs from the file outputted by MLE code
with open('retaining_wall_MLE_data.pkl', 'rb') as f:
    loaded_arrays_dict = pickle.load(f)

# Access the arrays from the loaded dictionary
X_pop = loaded_arrays_dict['X_pop']
Y_pop = loaded_arrays_dict['Y_pop']
X_val = loaded_arrays_dict['X_val']
Y_val = loaded_arrays_dict['Y_val']
initial_X_DoE = loaded_arrays_dict['initial_X_DoE']
initial_Y_DoE = loaded_arrays_dict['initial_Y_DoE']
N_initial_DoE = len(initial_X_DoE)

phi_pop = X_pop[:,0]
c_pop = X_pop[:,1]
```

Generation of Initial DoE

```
true_function = true_function_retaining_wall
noise_scale = 0.1

X_DoE = initial_X_DoE
Y_DoE = initial_Y_DoE

g_i = initialize_PLAXIS()

for X in X_DoE:
    Y_DoE.append(true_function(X, g_i))
```

Training with Initial DoE

```
n_chain = 4                                                 # number of
hyperparameter sampling chains
n_trace = 1000                                              # number of
hyperparameter samples per chain
thinning_factor = 40
n_thin = int(n_trace/thinning_factor)                       # number of
hyperparameters per thinned chain
n_pred_samples = 10                                         # number of
predictive samples for each hyperparameter sample
N_pred_samples = int(n_chain*n_thin*n_pred_samples)         # total number
of predictive samples

y_hat_pop = np.zeros((N_pred_samples, len(X_pop)))          # predictive
samples for population set
y_hat_val = np.zeros((N_pred_samples, len(X_val)))          # predictive
samples for validation set

with pm.Model() as GP_model:
    # Hyperparameters for the Gaussian Process
    log_ls_phi = pm.Uniform("log_ls_phi",
lower=np.log(np.min(phi_pop)/100), upper=np.log(np.max(phi_pop)*100))
    log_ls_c = pm.Uniform("log_ls_c", lower=np.log(np.min(c_pop)/100),
upper=np.log(np.max(c_pop)*100))
    log_cov_scale = pm.Uniform("log_cov_scale",
lower=np.log(0.000001), upper=np.log(10))
    log_sigma = pm.Uniform("log_sigma", lower=np.log(0.000001),
upper=np.log(10))

    ls_phi = pm.Deterministic('ls_phi', pm.math.exp(log_ls_phi))
    ls_c = pm.Deterministic('ls_c', pm.math.exp(log_ls_c))
    cov_scale = pm.Deterministic('cov_scale',
pm.math.exp(log_cov_scale))
    sigma = pm.Deterministic('sigma', pm.math.exp(log_sigma))

    # 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)

    trace = pm.sample(n_trace, return_inferencedata=True, chains = 4,
```

```
tune=1000, target_accept=0.95)   #  if a chain fails to unpickle,
cores = 1 to use only one core
    idata = trace.sel(draw=slice(None, None, thinning_factor))

    count = 0

    # cycle through every hyperparameter in the thinned traces
    for i in range(len(idata.posterior.chain)):
        for j in range(len(idata.posterior.draw)):

            hyperparam = {
                "log_ls_phi": idata.posterior['log_ls_phi'][i][j],
                "log_ls_c": idata.posterior['log_ls_c'][i][j],
                "log_cov_scale": idata.posterior['log_cov_scale'][i]
[j],
                "log_sigma": idata.posterior['log_sigma'][i][j],

                "ls_phi": idata.posterior['ls_phi'][i][j],
                "ls_c": idata.posterior['ls_c'][i][j],
                "cov_scale": idata.posterior['cov_scale'][i][j],
                "sigma": idata.posterior['sigma'][i][j]
            }

            # calculate predictive mean and variance
            mean_pop, var_pop = gp.predict(X_pop, point = hyperparam,
diag=True)
            mean_val, var_val = gp.predict(X_val, point = hyperparam,
diag=True)

            # generate predictive samples using predictive mean and
variance
            y_hat_pop[count:count+n_pred_samples] =
np.random.normal(mean_pop, np.sqrt(var_pop), (n_pred_samples,
len(X_pop)))
            y_hat_val[count:count+n_pred_samples] =
np.random.normal(mean_val, np.sqrt(var_val), (n_pred_samples,
len(X_val)))

            count += n_pred_samples

# sort predictive samples to conveniently obtain predictive
percentiles
y_hat_pop = np.sort(y_hat_pop, axis = 0)

index_2_5 = int(N_pred_samples*0.025)
index_50 = int(N_pred_samples*0.5)
index_97_5 = int(N_pred_samples*0.975)

p2_5_pop = y_hat_pop[index_2_5]
p50_pop = y_hat_pop[index_50]
```

```
p97_5_pop = y_hat_pop[index_97_5]

# calculate median of validation predictive samples to calculate
MSE_val
p50_val = np.median(y_hat_val, axis = 0)

# store relevant results
trace_list = [trace]
idata_list = [idata]
MSE_list = [MSE_solver(p50_val, Y_val)]
```

Enrichment

```
learning_function_type = 'var'
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_bayesian(mean_pop, var_pop,
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
        log_ls_phi = pm.Uniform("log_ls_phi",
lower=np.log(np.min(phi_pop)/100), upper=np.log(np.max(phi_pop)*100))
        log_ls_c = pm.Uniform("log_ls_c",
lower=np.log(np.min(c_pop)/100), upper=np.log(np.max(c_pop)*100))
        log_cov_scale = pm.Uniform("log_cov_scale",
lower=np.log(0.000001), upper=np.log(10))
        log_sigma = pm.Uniform("log_sigma", lower=np.log(0.000001),
upper=np.log(10))
```

```
        ls_phi = pm.Deterministic('ls_phi', pm.math.exp(log_ls_phi))
        ls_c = pm.Deterministic('ls_c', pm.math.exp(log_ls_c))
        cov_scale = pm.Deterministic('cov_scale',
pm.math.exp(log_cov_scale))
        sigma = pm.Deterministic('sigma', pm.math.exp(log_sigma))

        # 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)

        trace = pm.sample(n_trace, return_inferencedata=True, chains =
4, tune=1000, target_accept=0.95)   #  if a chain fails to unpickle,
cores = 1 to use only one core
        idata = trace.sel(draw=slice(None, None, thinning_factor))

        count = 0

        # cycle through every hyperparameter in the thinned traces
        for i in range(len(idata.posterior.chain)):
            for j in range(len(idata.posterior.draw)):

                hyperparam = {
                    "log_ls_phi": idata.posterior['log_ls_phi'][i][j],
                    "log_ls_c": idata.posterior['log_ls_c'][i][j],
                    "log_cov_scale": idata.posterior['log_cov_scale']
[i][j],
                    "log_sigma": idata.posterior['log_sigma'][i][j],

                    "ls_phi": idata.posterior['ls_phi'][i][j],
                    "ls_c": idata.posterior['ls_c'][i][j],
                    "cov_scale": idata.posterior['cov_scale'][i][j],
                    "sigma": idata.posterior['sigma'][i][j]
                }

                # calculate predictive mean and variance
                mean_pop, var_pop = gp.predict(X_pop, point =
hyperparam, diag=True)
                mean_val, var_val = gp.predict(X_val, point =
hyperparam, diag=True)
```

```
                # generate predictive samples using predictive mean
and variance
                y_hat_pop[count:count+n_pred_samples] =
np.random.normal(mean_pop, np.sqrt(var_pop), (n_pred_samples,
len(X_pop)))
                y_hat_val[count:count+n_pred_samples] =
np.random.normal(mean_val, np.sqrt(var_val), (n_pred_samples,
len(X_val)))

                count += n_pred_samples

    # sort predictive samples to conveniently obtain predictive
percentiles
    y_hat_pop = np.sort(y_hat_pop, axis = 0)

    p2_5_pop = y_hat_pop[index_2_5]
    p50_pop = y_hat_pop[index_50]
    p97_5_pop = y_hat_pop[index_97_5]

    # calculate median of validation predictive samples to calculate
MSE_val
    p50_val = np.median(y_hat_val, axis = 0)

    trace_list.append(trace)
    idata_list.append(idata)
    MSE_list.append(MSE_solver(p50_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,
    'trace_list': trace_list,
    'idata_list': idata_list,
    'MSE_list': MSE_list
}

file_path = 'retaining_wall_bayesian_data.pkl'

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