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

from learning_function_library import *
from plxscripting.easy import *
```

Definition of PLAXIS Functions

```
def initialize_PLAXIS():
    s_i, g_i = new_server('localhost', 10000, password =
'Mypassword1')

    g_i.clear()

    g_i.SoilContour.initializerectangular(0, -50, 150, 10)

    clay_params = [('SoilModel', 'Modified Cam-Clay'),
                   ('Identification', 'Clay'),
                   ('gammaUnsat', 17),
                   ('gammaSat', 17),
                   ('nuUR', 0.25),
                   ('GroundwaterSoilClassStandard', 'Fine'),
                   ('CInter', 15)]

    sand_params = [('SoilModel', 'Hardening Soil'),
                   ('Identification', 'Sand'),
                   ('gammaUnsat', 18),
                   ('gammaSat', 20),
                   ('nuUR', 0.2),
                   ('PowerM', 0.5),
                   ('GroundwaterSoilClassStandard', 'Coarse')]

    embankment_params = [('SoilModel', 'Hardening Soil'),
                   ('Identification', 'Embankment'),
                   ('gammaUnsat', 18),
                   ('gammaSat', 20),
                   ('E50Ref', 75000),
                   ('nuUR', 0.2),
                   ('PowerM', 0.5),
                   ('cRef', 2),
                   ('phi', 35),
                   ('psi', 5),
                   ('GroundwaterSoilClassStandard', 'Coarse'),
                   ('PermHorizontalPrimary', 1e-3*86400),
                   ('PermVertical', 1e-3*86400)]
```

```
    concrete_params = [('MaterialType', 'Elastic'),
                ('Identification', 'Concrete'),
                ('Gamma', 24),
                ('LSpacing', 4),
                ('PredefinedCrossSectionType', 'Solid square beam'),
                ('Width', 0.4),
                ('E', 30000000),
                ('AxialSkinResistance', 'Layer dependent')]

    clay_MCC = g_i.soilmat(*clay_params)
    sand_Hardening = g_i.soilmat(*sand_params)
    embankment_Hardening = g_i.soilmat(*embankment_params)
    concrete = g_i.embeddedbeammat(*concrete_params)

    g_i.borehole(0)
    g_i.Boreholes[0].Head = -2

    g_i.soillayer(2)
    g_i.soillayer(13)

    g_i.Soillayers[0].Soil.Material = sand_Hardening
    g_i.Soillayers[1].Soil.Material = clay_MCC

    g_i.polygon((53, 0), (56, 1), (94, 1), (97, 0))
    g_i.polygon((56, 1), (59, 2), (91, 2), (94, 1))
    g_i.polygon((59, 2), (62, 3), (88, 3), (91, 2))
    g_i.polygon((62, 3), (65, 4), (85, 4), (88, 3))

    for polygon in g_i.Polygons:
        polygon.Soil.Material = embankment_Hardening

    for x in ([55, 59, 63, 67, 71, 75, 79, 83, 87, 91, 95]):
        g_i.embeddedbeam((x, 0), (x, -8))

    for line in g_i.Lines:
        line.EmbeddedBeam.Material = concrete

    g_i.gotomesh()
    g_i.mesh(0.01)

    g_i.gotostages()

    InitialPhase = g_i.Phases[0]

    PilePhase = g_i.phase(InitialPhase)
    g_i.sps(PilePhase, 'Identification', 'Pile', 'DeformCalcType',
'Plastic')
    g_i.activate(g_i.Lines, PilePhase)

    EmbankmentPhase1 = g_i.phase(PilePhase)
```

```
    g_i.sps(EmbankmentPhase1, 'Identification', 'Embankment1',
'DeformCalcType', 'Consolidation', 'TimeInterval', 2)
    g_i.activate(g_i.Polygons[3], EmbankmentPhase1)

    EmbankmentPhase2 = g_i.phase(EmbankmentPhase1)
    g_i.sps(EmbankmentPhase2, 'Identification', 'Embankment2',
'DeformCalcType', 'Consolidation', 'TimeInterval', 2)
    g_i.activate(g_i.Polygons[2], EmbankmentPhase2)

    EmbankmentPhase3 = g_i.phase(EmbankmentPhase2)
    g_i.sps(EmbankmentPhase3, 'Identification', 'Embankment3',
'DeformCalcType', 'Consolidation', 'TimeInterval', 2)
    g_i.activate(g_i.Polygons[1], EmbankmentPhase3)

    EmbankmentPhase4 = g_i.phase(EmbankmentPhase3)
    g_i.sps(EmbankmentPhase4, 'Identification', 'Embankment4',
'DeformCalcType', 'Consolidation', 'TimeInterval', 2)
    g_i.activate(g_i.Polygons[0], EmbankmentPhase4)

    EndOfConsolidationPhase = g_i.phase(EmbankmentPhase4)
    g_i.sps(EndOfConsolidationPhase, 'Identification',
'EndOfConsolidation', 'DeformCalcType', 'Consolidation',
'TimeInterval', 18250)

    localhostport_o = g_i.selectmeshpoints()
    s_o, g_o = new_server('localhost', localhostport_o, password =
'Mypassword1')
    g_o.addcurvepoint('Node', 75, 4)
    g_o.update()

    return g_i

def true_function_PLAXIS(X, g_i):

    phi = X[0]
    c = X[1]
    E = X[2]
    M_MCC = X[3]
    lambda_MCC = X[4]
    kappa_MCC = X[5]
    k_clay = X[6]
    k_sand = X[7]
    OCR = X[8]

    # Calculate other paramaters based on given
    psi = np.max(phi - 30, 0)

    clay_params = [ ('lambda', lambda_MCC),
                    ('kappa', kappa_MCC),
                    ('M', M_MCC),
```

```
                    ('PermHorizontalPrimary', k_clay*86400),
                    ('PermVertical', k_clay*86400),
                    ('OCR', OCR)]

    sand_params = [ ('E50Ref', E),
                    ('cRef', c),
                    ('phi', phi),
                    ('psi', psi),
                    ('PermHorizontalPrimary', k_sand*86400),
                    ('PermVertical', k_sand*86400)]


    clay_MCC = g_i.Clay
    sand_Hardening = g_i.Sand

    EndOfConsolidationPhase = g_i.Phases[-1]

    g_i.sps(clay_MCC, *clay_params)
    g_i.sps(sand_Hardening, *sand_params)

    g_i.gotostages()

    for phase in g_i.Phases:
        phase.ShouldCalculate = True

    g_i.calculate()

    phase = g_i.Phases[-1]
    localhostport_o = g_i.view(phase)
    s_o, g_o = new_server('localhost', localhostport_o, password =
'Mypassword1')

    uy_EoC = np.abs(g_o.getcurveresults(g_o.CN_1,
EndOfConsolidationPhase, g_o.ResultTypes.Soil.Uy))

    g_o.close()

    return ((0.15/uy_EoC) - 1)

def kill_Plaxis_2D_Output():
    '''Kills Plaxis 2D Output to refresh memories'''

    try:
        subprocess.run(["taskkill", "/IM", "Plaxis2DOutput.exe",
"/F"], check=True)
    except subprocess.CalledProcessError as e:
        print(f"Failed to terminate Plaxis 2D Output: {e}")
    except Exception as e:
        print(f"An error occurred: {e}")
```

Loading of Stochastic Variables from MLE Code

```
N_pop = 1000000

# Load the dictionary of inputs from the file (outputted by MLE code)
with open('piledembankment_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']
initial_X_DoE = loaded_arrays_dict['initial_X_DoE']
N_initial_DoE = len(initial_X_DoE)

phi_pop = X_pop[:,0]
c_pop = X_pop[:,1]
E_pop = X_pop[:,2]
M_pop = X_pop[:,3]
lambda_pop = X_pop[:,4]
kappa_pop = X_pop[:,5]
k_clay_pop = X_pop[:,6]
k_sand_pop = X_pop[:,7]
OCR_pop = X_pop[:,8]
```

Generation of Initial DoE

```
true_function = true_function_PLAXIS

X_DoE = initial_X_DoE
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_hyperparam_samples = int(n_chain*n_thin)   # total number of
hyperparameter samples
N_pred_samples = int(n_chain*n_thin*n_pred_samples)   # total number
of predictive samples

y_mean_pop = np.zeros((N_hyperparam_samples, len(X_pop)))  #
```

```
predictive means of GPs for each hyperparameter sample
y_hat_pop = np.zeros((N_pred_samples, len(X_pop)))   # predictive
samples

with pm.Model() as GP_model:
    # Hyperparameters for the Gaussian Process using log-uniform
priors
    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_ls_E = pm.Uniform("log_ls_E", lower=np.log(np.min(E_pop)/100),
upper=np.log(np.max(E_pop)*100))
    log_ls_M_MCC = pm.Uniform("log_ls_M_MCC",
lower=np.log(np.min(M_pop)/100), upper=np.log(np.max(M_pop)*100))
    log_ls_lambda_MCC = pm.Uniform("log_ls_lambda_MCC",
lower=np.log(np.min(lambda_pop)/100),
upper=np.log(np.max(lambda_pop)*100))
    log_ls_kappa_MCC = pm.Uniform("log_ls_kappa_MCC",
lower=np.log(np.min(kappa_pop)/100),
upper=np.log(np.max(kappa_pop)*100))
    log_ls_k_clay = pm.Uniform("log_ls_k_clay",
lower=np.log(np.min(k_clay_pop)/100),
upper=np.log(np.max(k_clay_pop)*100))
    log_ls_k_sand = pm.Uniform("log_ls_k_sand",
lower=np.log(np.min(k_sand_pop)/100),
upper=np.log(np.max(k_sand_pop)*100))
    log_ls_OCR = pm.Uniform("log_ls_OCR",
lower=np.log(np.min(OCR_pop)/100), upper=np.log(np.max(OCR_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))
    ls_E = pm.Deterministic('ls_E', pm.math.exp(log_ls_E))
    ls_M_MCC= pm.Deterministic('ls_M_MCC', pm.math.exp(log_ls_M_MCC))
    ls_lambda_MCC = pm.Deterministic('ls_lambda_MCC',
pm.math.exp(log_ls_lambda_MCC))
    ls_kappa_MCC = pm.Deterministic('ls_kappa_MCC',
pm.math.exp(log_ls_kappa_MCC))
    ls_k_clay = pm.Deterministic('ls_k_clay',
pm.math.exp(log_ls_k_clay))
    ls_k_sand = pm.Deterministic('ls_k_sand',
pm.math.exp(log_ls_k_sand))
    ls_OCR = pm.Deterministic('ls_OCR', pm.math.exp(log_ls_OCR))
    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=9,
ls=[ls_phi, ls_c, ls_E, ls_M_MCC, ls_lambda_MCC, ls_kappa_MCC,
ls_k_clay, ls_k_sand, ls_OCR])

    # 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 = sigma)

    trace = pm.sample(n_trace, return_inferencedata=True, chains = 4,
tune=2000, 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_ls_E": idata.posterior['log_ls_E'][i][j],
                "log_ls_M_MCC": idata.posterior['log_ls_M_MCC'][i][j],
                "log_ls_lambda_MCC":
idata.posterior['log_ls_lambda_MCC'][i][j],
                "log_ls_kappa_MCC":
idata.posterior['log_ls_kappa_MCC'][i][j],
                "log_ls_k_clay": idata.posterior['log_ls_k_clay'][i]
[j],
                "log_ls_k_sand": idata.posterior['log_ls_k_sand'][i]
[j],
                "log_ls_OCR": idata.posterior['log_ls_OCR'][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],
                "ls_E": idata.posterior['ls_E'][i][j],
                "ls_M_MCC": idata.posterior['ls_M_MCC'][i][j],
                "ls_lambda_MCC": idata.posterior['ls_lambda_MCC'][i]
```

```
[j],
                "ls_kappa_MCC": idata.posterior['ls_kappa_MCC'][i][j],
                "ls_k_clay": idata.posterior['ls_k_clay'][i][j],
                "ls_k_sand": idata.posterior['ls_k_sand'][i][j],
                "ls_OCR": idata.posterior['ls_OCR'][i][j],
                "cov_scale": idata.posterior['cov_scale'][i][j],
                "sigma": idata.posterior['sigma'][i][j]
            }

            # calculate predictive mean and variance and store mean in
y_mean_pop
            mean_pop, var_pop = gp.predict(X_pop, point = hyperparam,
diag=True)
            y_mean_pop[int(n_thin*i+j)] = mean_pop

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

            count += n_pred_samples

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

index_2_5_y_mean = int(N_hyperparam_samples*0.025)
index_50_y_mean = int(N_hyperparam_samples*0.5)
index_97_5_y_mean = int(N_hyperparam_samples*0.975)

index_2_5_y_hat = int(N_pred_samples*0.025)
index_50_y_hat = int(N_pred_samples*0.5)
index_97_5_y_hat = int(N_pred_samples*0.975)

p2_5_y_mean = y_mean_pop[index_2_5_y_mean]
p50_y_mean = y_mean_pop[index_50_y_mean]
p97_5_y_mean = y_mean_pop[index_97_5_y_mean]

p2_5_y_hat = y_hat_pop[index_2_5_y_hat]
p50_y_hat = y_hat_pop[index_50_y_hat]
p97_5_y_hat = y_hat_pop[index_97_5_y_hat]

_, Pf_credible_plus, Pf_credible_minus, _ =
calculate_Pf_bayesian(p2_5_y_mean, p50_y_mean, p97_5_y_mean)
Pf, Pf_plus, Pf_minus, conv_criterion =
calculate_Pf_bayesian(p2_5_y_hat, p50_y_hat, p97_5_y_hat)
end_time_iter = time.time()
```

```
# store relevant results
trace_list = [trace]
idata_list = [idata]
Pf_list = [Pf]
Pf_credible_plus_list = [Pf_credible_plus]
Pf_credible_minus_list = [Pf_credible_minus]
Pf_plus_list = [Pf_plus]
Pf_minus_list = [Pf_minus]
conv_criterion_list = [conv_criterion]
```

Enrichment

```
learning_function_type =  'U'  # in this case, negative U is used such that the highest value is selected

for iterations in range(N_pop-N_initial_DoE):

    # solve the learning function values for each point in the population set
    LFV = learning_function_bayesian(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 = true_function(x_new, g_i)
            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 using log-uniform priors
        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_ls_E = pm.Uniform("log_ls_E", lower=np.log(np.min(E_pop)/100), upper=np.log(np.max(E_pop)*100))
        log_ls_M_MCC = pm.Uniform("log_ls_M_MCC", lower=np.log(np.min(M_pop)/100), upper=np.log(np.max(M_pop)*100))
```

```
        log_ls_lambda_MCC = pm.Uniform("log_ls_lambda_MCC",
lower=np.log(np.min(lambda_pop)/100),
upper=np.log(np.max(lambda_pop)*100))
        log_ls_kappa_MCC = pm.Uniform("log_ls_kappa_MCC",
lower=np.log(np.min(kappa_pop)/100),
upper=np.log(np.max(kappa_pop)*100))
        log_ls_k_clay = pm.Uniform("log_ls_k_clay",
lower=np.log(np.min(k_clay_pop)/100),
upper=np.log(np.max(k_clay_pop)*100))
        log_ls_k_sand = pm.Uniform("log_ls_k_sand",
lower=np.log(np.min(k_sand_pop)/100),
upper=np.log(np.max(k_sand_pop)*100))
        log_ls_OCR = pm.Uniform("log_ls_OCR",
lower=np.log(np.min(OCR_pop)/100), upper=np.log(np.max(OCR_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))
        ls_E = pm.Deterministic('ls_E', pm.math.exp(log_ls_E))
        ls_M_MCC= pm.Deterministic('ls_M_MCC',
pm.math.exp(log_ls_M_MCC))
        ls_lambda_MCC = pm.Deterministic('ls_lambda_MCC',
pm.math.exp(log_ls_lambda_MCC))
        ls_kappa_MCC = pm.Deterministic('ls_kappa_MCC',
pm.math.exp(log_ls_kappa_MCC))
        ls_k_clay = pm.Deterministic('ls_k_clay',
pm.math.exp(log_ls_k_clay))
        ls_k_sand = pm.Deterministic('ls_k_sand',
pm.math.exp(log_ls_k_sand))
        ls_OCR = pm.Deterministic('ls_OCR', pm.math.exp(log_ls_OCR))
        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=9,
ls=[ls_phi, ls_c, ls_E, ls_M_MCC, ls_lambda_MCC, ls_kappa_MCC,
ls_k_clay, ls_k_sand, ls_OCR])

        # 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 = sigma)

        trace = pm.sample(n_trace, return_inferencedata=True, chains =
4, tune=2000, 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 trace
        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_ls_E": idata.posterior['log_ls_E'][i][j],
                    "log_ls_M_MCC": idata.posterior['log_ls_M_MCC'][i]
[j],
                    "log_ls_lambda_MCC":
idata.posterior['log_ls_lambda_MCC'][i][j],
                    "log_ls_kappa_MCC":
idata.posterior['log_ls_kappa_MCC'][i][j],
                    "log_ls_k_clay": idata.posterior['log_ls_k_clay']
[i][j],
                    "log_ls_k_sand": idata.posterior['log_ls_k_sand']
[i][j],
                    "log_ls_OCR": idata.posterior['log_ls_OCR'][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],
                    "ls_E": idata.posterior['ls_E'][i][j],
                    "ls_M_MCC": idata.posterior['ls_M_MCC'][i][j],
                    "ls_lambda_MCC": idata.posterior['ls_lambda_MCC']
[i][j],
                    "ls_kappa_MCC": idata.posterior['ls_kappa_MCC'][i]
[j],
                    "ls_k_clay": idata.posterior['ls_k_clay'][i][j],
                    "ls_k_sand": idata.posterior['ls_k_sand'][i][j],
                    "ls_OCR": idata.posterior['ls_OCR'][i][j],
                    "cov_scale": idata.posterior['cov_scale'][i][j],
                    "sigma": idata.posterior['sigma'][i][j]
                }

                # calculate predictive mean and variance and store
```

```
mean in y_mean_pop
                mean_pop, var_pop = gp.predict(X_pop, point =
hyperparam, diag=True)
                y_mean_pop[int(n_thin*i+j)] = mean_pop

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

                count += n_pred_samples

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

    p2_5_y_mean = y_mean_pop[index_2_5_y_mean]
    p50_y_mean = y_mean_pop[index_50_y_mean]
    p97_5_y_mean = y_mean_pop[index_97_5_y_mean]

    p2_5_y_hat = y_hat_pop[index_2_5_y_hat]
    p50_y_hat = y_hat_pop[index_50_y_hat]
    p97_5_y_hat = y_hat_pop[index_97_5_y_hat]

    _, Pf_credible_plus, Pf_credible_minus, _ =
calculate_Pf_bayesian(p2_5_y_mean, p50_y_mean, p97_5_y_mean)
    Pf, Pf_plus, Pf_minus, conv_criterion =
calculate_Pf_bayesian(p2_5_y_hat, p50_y_hat, p97_5_y_hat)

    trace_list.append(trace)
    idata_list.append(idata)
    Pf_list.append(Pf)
    Pf_credible_plus_list.append(Pf_credible_plus)
    Pf_credible_minus_list.append(Pf_credible_minus)
    Pf_plus_list.append(Pf_plus)
    Pf_minus_list.append(Pf_minus)
    conv_criterion_list.append(conv_criterion)

    # Check if training should stop
    if conv_criterion <= 0.05 and iterations >= 100:
        break
```

Saving of Results

```
results_dict = {
    'X_pop': X_pop,
    'initial_X_DoE': initial_X_DoE,
```

```
    'X_DoE': X_DoE,
    'Y_DoE': Y_DoE,
    'trace_list': trace_list,
    'idata_list': idata_list,
    'Pf_list': Pf_list,
    'Pf_credible_plus_list': Pf_credible_plus_list,
    'Pf_credible_minus_list': Pf_credible_minus_list,
    'Pf_plus_list': Pf_plus_list,
    'Pf_minus_list': Pf_minus_list,
    'conv_criterion_list': conv_criterion_list
}

file_path = 'piledembankment_bayesian_data.pkl'

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