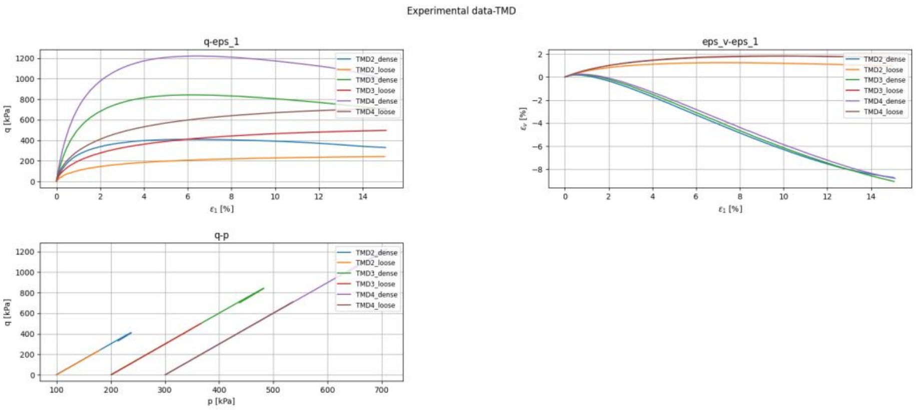
# Assignment 2

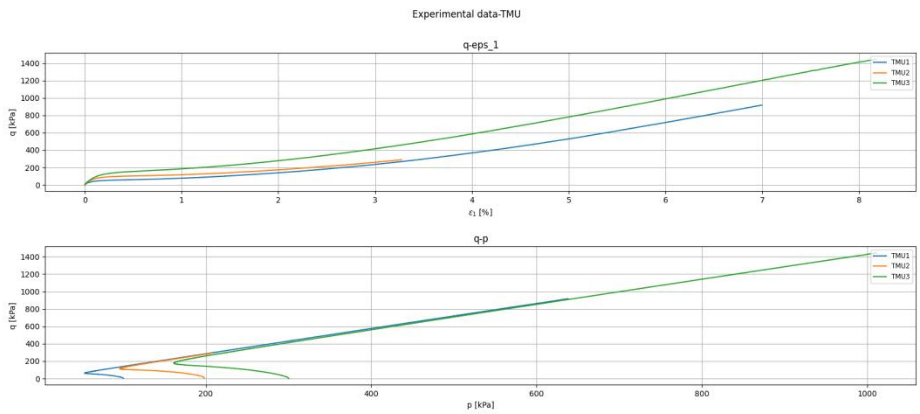
## Task A: Determination of the Mohr-Coulomb model parameters

### 1. Experimental data plot

#### (1) Drained triaxial test




#### (2) Undrained triaxial test




#### (3) Plot code

```
#================================
# Plot the experimental data
#================================
import numpy as np
import matplotlib.pyplot as plt
from plottingTaskA import plotting_testData
from dataExperiment import *
from scipy.optimize import curve_fit
import math

# Data group
expData_TMD =
[expData()[0],expData()[1],expData()[2],expData()[3],expData()[4],expData()[5]]
labels_TMD = ('TMD2_dense', 'TMD2_loose', 'TMD3_dense', 'TMD3_loose',
'TMD4_dense', 'TMD4_loose')
expData_TMU = [expData()[6],expData()[7],expData()[8]]
labels_TMU = ('TMU1', 'TMU2', 'TMU3')

# Emperimental data plotting
print("The experimental data you want to plot: ",
```

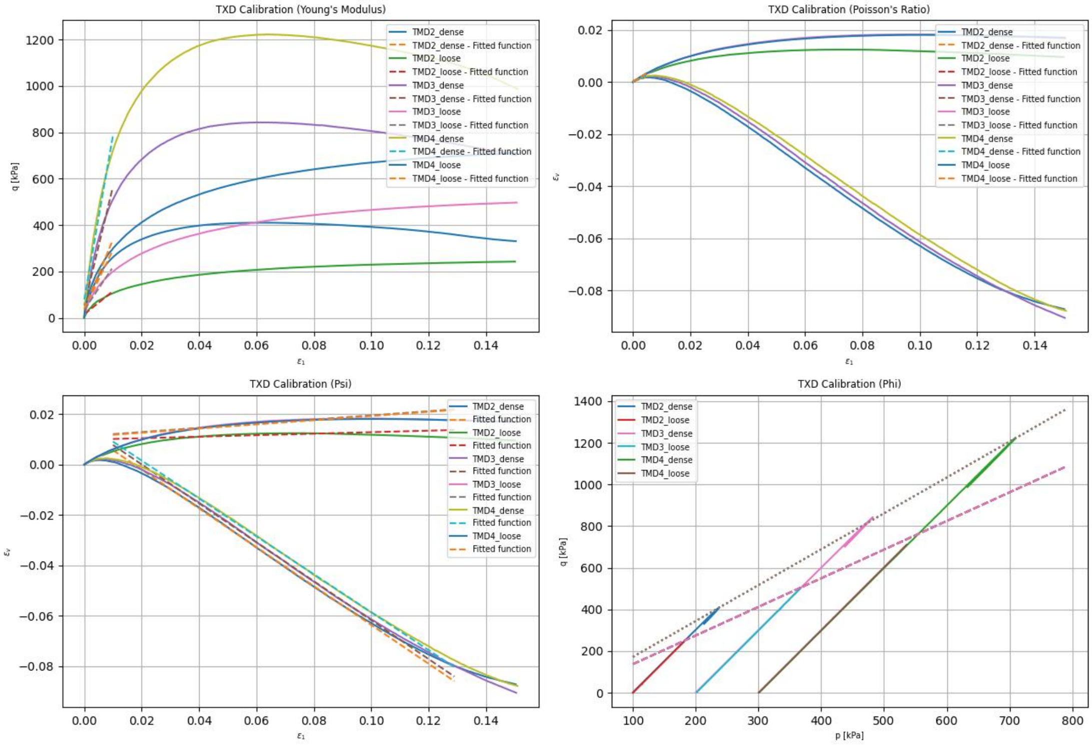
```
    "1: txd, 2: txu"
   )
testType = input("which kind of plot you want to realize: ")
if testType == '1':
   plotting_testData(expData_TMD,labels_TMD,'txd',None,None,None)
elif testType == '2':
   plotting_testData(expData_TMU,labels_TMU,'txu',None,None,None)
```

## 2. Parameters calibration and Plot results

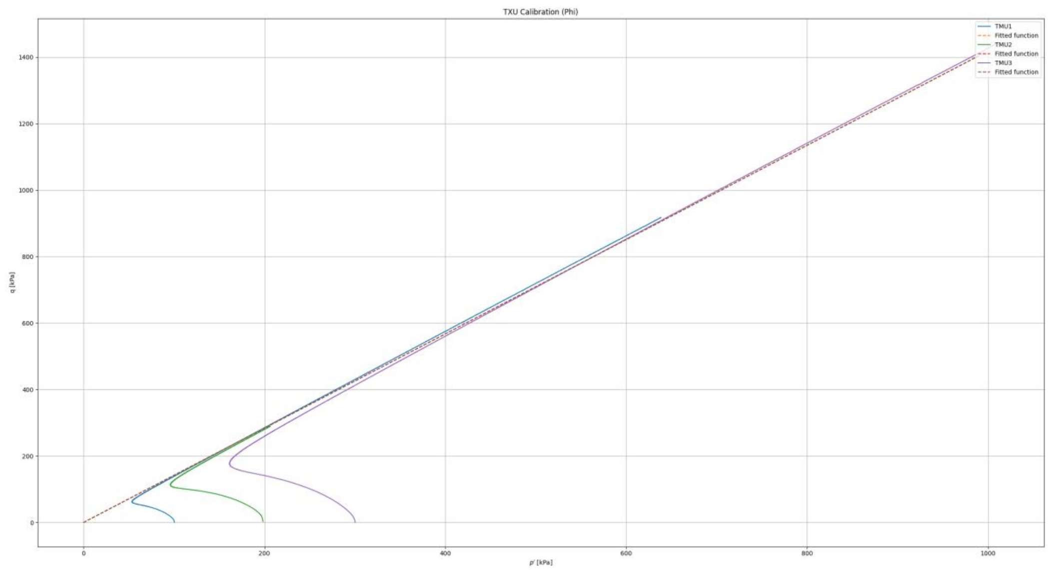
**(1) Parameters calibration results:**

| | E<br>[kPa] | nu<br>[-] | Phi<br>[ ∘ ] | Psi<br>[ ∘ ] |
|---|---|---|---|---|
| **TMD2_dense** | 25630.5 | 0.325 | 42 | 16.2 |
| **TMD3_dense** | 51363.0 | 0.282 | 42 | 16.2 |
| **TMD4_dense** | 70741.3 | 0.260 | 42 | 15.9 |
| **TMD2_loose** | 10003.7 | 0.161 | 34 | -0.9 |
| **TMD3_loose** | 18461.5 | 0.167 | 34 | -2.5 |
| **TMD4_loose** | 30030.6 | 0.146 | 34 | -2.5 |
| **TMU1** | 25647.6 | 0.325 | 35 | 16.2 |
| **TMU2** | 51397.3 | 0.325 | 35 | 16.2 |
| **TMU3** | 70788.5 | 0.325 | 35 | 16.2 |

**(2) Parameters calibration plot under drained triaxial test:**




**(3) Parameters calibration plot under undrained triaxial test:**




**(4) Calibration code**

```
#================================
# Calibration of the parameters
#================================


def test_func_linear(x, a, b):
    return a * x + b

# Data group
TMD = [expData()[0],expData()[1],expData()[2],expData()[3],expData()[4],expData()[5]]
labels1 = ['TMD2_dense', 'TMD2_loose', 'TMD3_dense', 'TMD3_loose', 'TMD4_dense',
'TMD4_loose']
```

```
TMU = [expData()[6],expData()[7],expData()[8]]
labels2 = ['TMU1', 'TMU2', 'TMU3']

# Fit parameters
def plot_phi_fit(ax, phi, p, q, line_style):
    M = 6 * math.sin(math.radians(phi)) / (3 - math.sin(math.radians(phi)))
    ax.plot(np.arange(100, 800, 10), test_func_linear(np.arange(100, 800, 10), M, 0),
line_style)
    return phi

def fit_params(data_type, labels):
    if data_type == 'TMD':
        fig, axs = plt.subplots(2, 2, figsize=(13, 9))
        for idx, (data, label) in enumerate(zip(TMD, labels)):
            eps_1 = data[:, 0]/100
            q = data[:, 5]
            eps_v = data[:, 1]/100
            p = data[:, 6]

            # E-fit
            indices = np.where(eps_1 < 0.01)
            eps_1_fit = eps_1[indices]
            q_fit = q[indices]
            p0 = [10000, 0]
            params, params_covariance = curve_fit(test_func_linear, eps_1_fit, q_fit, p0)

            # nu-fit
            indices_nu = np.where(eps_1 < 0.005)
            eps_1_fitNu = eps_1[indices_nu]
            eps_v_fitNu = eps_v[indices_nu]
            p1 = [0.5, 0]
            paramsNu, paramsNu_covariance = curve_fit(test_func_linear, eps_1_fitNu,
eps_v_fitNu, p1)
            paramsNu = [1/2 * (1-i) for i in paramsNu]

            # Psi-fit
            eps_1_fitPsi = eps_1[np.where((eps_1 > 0.025) & (eps_1 < 0.1))]
            eps_vPsi = eps_v[np.where((eps_1 > 0.025) & (eps_1 < 0.1))]
            p2 = [1, -1]
            paramsPsi, paramsPsi_covariance = curve_fit(test_func_linear, eps_1_fitPsi,
eps_vPsi, p2)
            sin_Psi = paramsPsi[0] / (paramsPsi[0] - 2)
            Psi_deg = math.degrees(math.asin(sin_Psi))

            # Fitting and Plotting
```

```
        axs[0, 0].plot(eps_1, q, label=label)
        axs[0, 0].plot(eps_1_fit, test_func_linear(eps_1_fit, *params), '--', label=f'{label} -
Fitted function')
        axs[0, 1].plot(eps_1, eps_v, label=label)
        axs[0, 1].plot(eps_1_fitNu, test_func_linear(eps_1_fitNu, *paramsNu), '--',
label=f'{label} - Fitted function')
        axs[1, 0].plot(eps_1, eps_v, label=label)
        axs[1, 0].plot(np.arange(0.01, 0.13, 0.001), test_func_linear(np.arange(0.01, 0.13,
0.001), paramsPsi[0], paramsPsi[1]),'--',
                label='Fitted function')
        # Phi-fit
        axs[1,1].plot(p, q, label=label)
        phi_loose = plot_phi_fit(axs[1, 1], 34, p, q, '--')
        phi_dense = plot_phi_fit(axs[1, 1], 42, p, q, ':')

        print(f"Optimized parameters for {label} (E, C):", params)
        print(f"Young's modulus E = {params[0]:.1f} kPa")
        print(f"Optimized parameters for {label} (Nu, C):", paramsNu)
        print(f"Poisson's ratio Nu = {paramsNu[0]:.3f}")
        print(f"Optimized parameters for {label} (Psi, C):", paramsPsi)
        print(f"Dilatancy angle Psi = {Psi_deg:.1f} degrees")
        print(f"Phi for loose sample = {phi_loose} degrees")
        print(f"Phi for dense sample = {phi_dense} degrees")

    axs[0, 0].set_title('TXD Calibration (Young\'s Modulus)', fontsize='small')
    axs[0, 0].set_xlabel('$\\epsilon_{1}$', fontsize='x-small')
    axs[0, 0].set_ylabel('q [kPa]', fontsize='x-small')
    axs[0, 0].legend(loc='upper right', fontsize='x-small')
    axs[0, 0].grid(which='both')

    axs[0, 1].set_title('TXD Calibration (Poisson\'s Ratio)', fontsize='small')
    axs[0, 1].set_xlabel('$\\epsilon_{1}$', fontsize='x-small')
    axs[0, 1].set_ylabel('$\\epsilon_{v}$', fontsize='x-small')
    axs[0, 1].legend(loc='upper right', fontsize='x-small')
    axs[0, 1].grid(which='both')

    axs[1, 0].set_title('TXD Calibration (Psi)', fontsize='small')
    axs[1, 0].set_xlabel('$\\epsilon_{1}$', fontsize='x-small')
    axs[1, 0].set_ylabel('$\\epsilon_{v}$', fontsize='x-small')
    axs[1, 0].legend(loc='upper right', fontsize='x-small')
    axs[1, 0].grid(which='both')

    axs[1, 1].set_title('TXD Calibration (Phi)', fontsize='small')
    axs[1, 1].set_xlabel("p [kPa]", fontsize='x-small')
    axs[1, 1].set_ylabel('q [kPa]', fontsize='x-small')
```

```
        axs[1, 1].legend(loc='upper left', fontsize='x-small')
        axs[1, 1].grid(which='both')

    elif data_type == 'TMU':
        fig, axs = plt.subplots(1, 1, figsize=(10, 7))
        for idx, (data, label) in enumerate(zip(TMU, labels)):
            q = data[:, 7]
            p = data[:, 6]

            # Phi-fit
            phi = 35
            M  = 6*math.sin(math.radians(phi))/(3-math.sin(math.radians(phi)))

            # Fitting and Plotting
            axs.plot(p, q, label=label)
            axs.plot(np.arange(0,1000,10), test_func_linear(np.arange(0,1000,10), M, 0),'--',
label='Fitted function')

        axs.set_title('TXU Calibration (Phi)')
        axs.set_xlabel(r"$p'$ [kPa]")
        axs.set_ylabel('q [kPa]')
        axs.legend(loc='upper right', fontsize='medium')
        axs.grid(which='both')
        print(f"Friction angle Phi = {phi:.1f} degrees")

    plt.subplots_adjust(hspace=0.6, wspace=0.4)
    plt.tight_layout()
    plt.show()

fit_params('TMD', labels1)
fit_params('TMU', labels2)
```

## 3. Original source code:

**dataExperiment.py:** for combine the given experimental data by defining the function “dataExperiment”

**taskA.py:** for experimental plotting and parameters calibration

**plottingTaskA.py:** for taskA plottin

## TaskB: Extend the Python routines and run simulations

### 1. Implement the Drucker-Prager failure criterion in the python code

```
elif model_pl == "dp":
    # converting friction and dilatancy angles from [°] to [rad]
    phi = math.radians(modelParam["phi"])
    psi = math.radians(modelParam["psi"])
    p = stress[0]
    q = stress[1]

    # Drucker-Prager failure criterion
    f = q - (6 * math.sin(phi) / (3 - math.sin(phi))) * p

    # gradient of yield surface
    df_dp = -(6 * math.sin(phi) / (3 - math.sin(phi)))
    df_dq = 1
    # gradient of plastic potential
    dg_dp = -(6 * math.sin(psi) / (3 - math.sin(psi)))
    dg_dq = 1
    # gradient of the additional state variables
    df_dchi = 0
    dchi_depsp = 0
    dchi_depsq = 0
```

### 2. Simulate both txd and txu under MC and DP model and compare with expeimental results.

**Remark:**

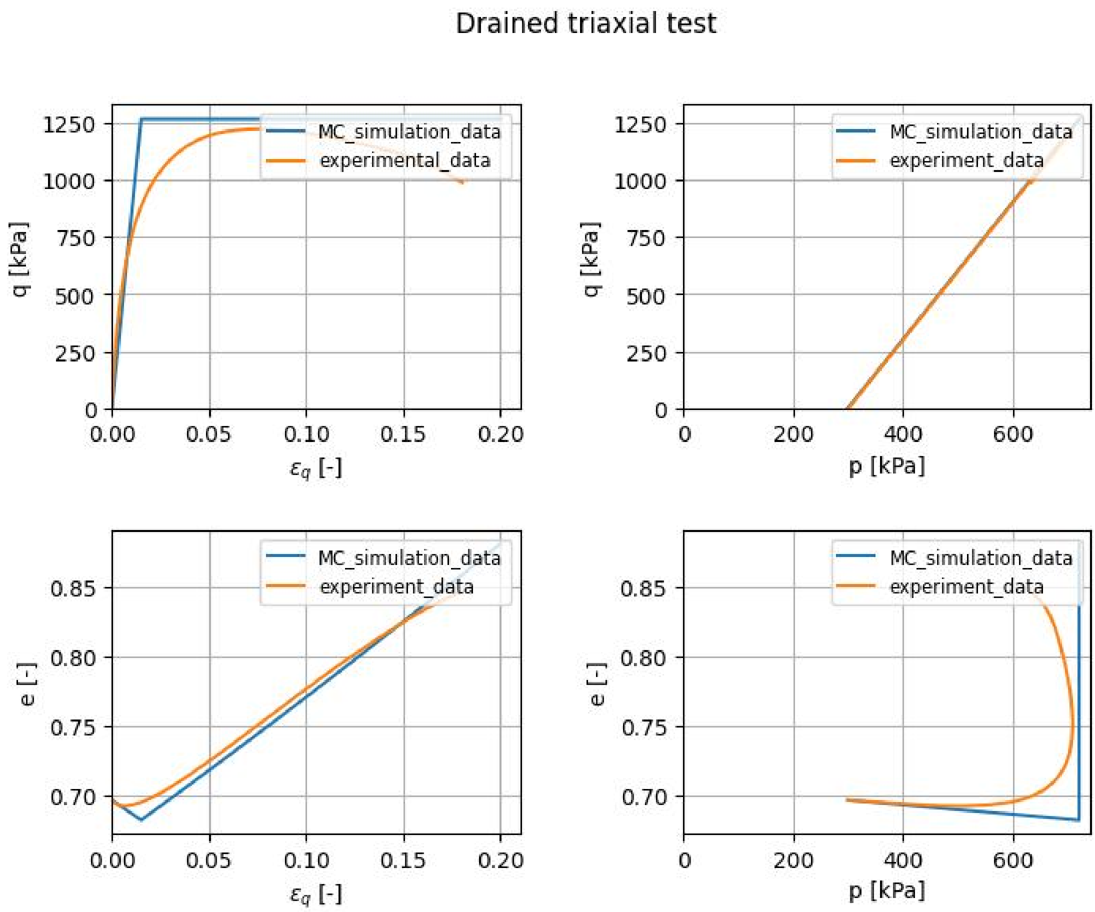
**1) MC: Mohr-Coulomb, DP: Drucker-Prager, txd: drained triaxial test, txu: undrained triaxial test**

**a) MC model_dense soil_txd**

```
import numpy, sys
# import additional functions
from functions_elementTest import *
from functions_matModels import *
from plotting_routines import *
from dataExperiment import *

#================================
# Input
#================================

# initial state
epor = 0.697   # void ratio
p = 300            # [kPa]
q = 2.00       # [kPa]
eps_1 = 0      # [-]
eps_2 = 0     # [-]

# material models for the elastic and the plastic part
```

```
# "linelast" ... linear elasticity
model_el = "linelast"
# "mc"  ... mohr-coulomb
# "dp" ... drucker-prager
model_pl = "mc"
# model_pl = "dp"
model={"model_el":model_el, "model_pl":model_pl}

# material parameters
E = 70741.3
nu =  0.260
phi = 42
psi = 15.9
modelParam={"E":E, "nu":nu, "phi":phi, "psi":psi}

# test control
# "oed" ... oedometer test
# "txd" ... drained triaxial test
# "txu" ... undrained triaxial test
testType = 'txd'

# stop criteria for calculation
p_max = 1000   # [kPa]
epsq_max = 0.2 # [%]
i_max = 1000    # max iterations

# numerical parameter
dt = 0.001        # time step (numerical integration)

#======================================================================
========
# Calculation
#======================================================================
========

# inital stress and strain in volumetric and deviatoric invariants
stress = numpy.array([p, q])
strain = eps2pq(eps_1,eps_2)
stateVar = numpy.array([epor,0])
sigma_1 = pq2sigma(stress[0], stress[1])[0]
sigma_2 = pq2sigma(stress[0], stress[1])[1]

# record the state variables in a numpy array in the following order
# column: eps_p, eps_q, p, q, eps_1, eps_2, sigma_1, sigma_2, epor
# every row represents a time step
data = numpy.array([[strain[0], strain[1], stress[0], stress[1], eps_1, eps_2, sigma_1,
sigma_2, epor, stateVar[1] ]])

# initialise the loop
step = 1
while (stress[0] < p_max) and (strain[1] < epsq_max) and (step < i_max):
```

```
    dstress, dstrain, dStateVar = rates(stress,strain,stateVar,testType,model,modelParam)
    stress, strain, stateVar = integration(stress,strain,dstress,dstrain,stateVar,dStateVar,dt)
    data = recordData(data,stress,strain,stateVar)
    step += 1

# open output file for saving the data and the plot
filename = 'output_' + testType + 'expDataV.S.MC'+'_TMD4_dense'

# save results to a text file
printResults(filename,data)
data2 = expData()[4]

# plot the results
plotting_testData(data, testType='txd', outputName=filename, data2=data2,
data3=None)
```



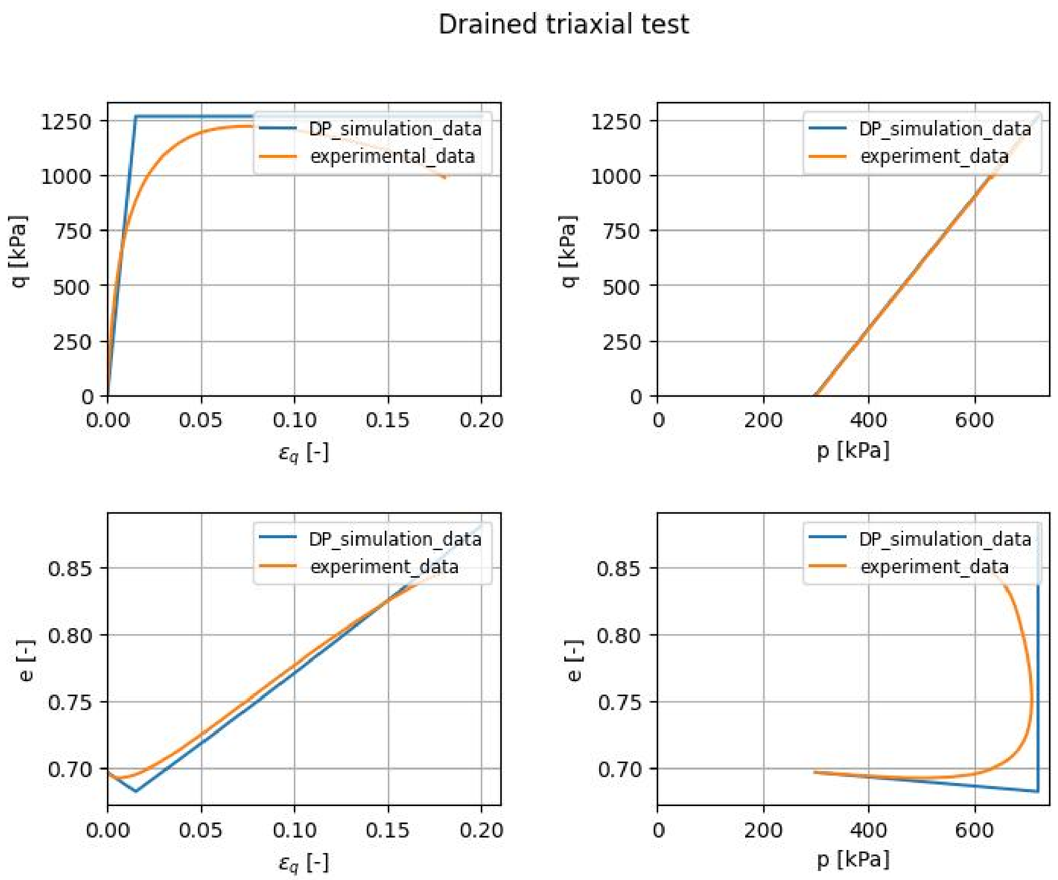

**Results analysis:**

The overall simulation of MC model looks good, the reason is the good calibration of parameters and MC model is suitable for this soil situation. In details, for E, we can see at the elastic part of q-eps_q diagram, the experimental data and simulation data is approach, for phi, the plastic part of of q-eps_q diagram, and q-p diagram, the experimental data and simulation data is closed. for nu, we can see the linear part of the diagram e-eps_q, the two lines almost coincided. and the plastic range of e-eps_q, the psi also looks good. The last diagram can get a good simulation results of the dilatancy.

**b) DP model_dense soil_txd**

```
import numpy, sys
# import additional functions
from functions_elementTest import *
from functions_matModels import *
from plotting_routines import *
from dataExperiment import *

#================================
# Input
#================================

# initial state
epor = 0.697   # void ratio
p = 300           # [kPa]
q = 2.00        # [kPa]
eps_1 = 0       # [-]
eps_2 = 0      # [-]

# material models for the elastic and the plastic part
# "linelast" ... linear elasticity
model_el = "linelast"
# "mc"  ... mohr-coulomb
# "dp" ... drucker-prager
# model_pl = "mc"
model_pl = "dp"
model={"model_el":model_el, "model_pl":model_pl}

# material parameters
E = 70741.3
nu = 0.260
phi = 42
psi = 15.9
modelParam={"E":E, "nu":nu, "phi":phi, "psi":psi}

# test control
# "oed" ... oedometer test
# "txd" ... drained triaxial test
# "txu" ... undrained triaxial test
testType = 'txd'

# stop criteria for calculation
p_max = 1000   # [kPa]
epsq_max = 0.2 # [%]
i_max = 1000    # max iterations

# numerical parameter
dt = 0.001        # time step (numerical integration)
```

```
#==========================================================================
========
# Calculation
#==========================================================================
========

# inital stress and strain in volumetric and deviatoric invariants
stress = numpy.array([p, q])
strain = eps2pq(eps_1,eps_2)
stateVar = numpy.array([epor,0])
sigma_1 = pq2sigma(stress[0], stress[1])[0]
sigma_2 = pq2sigma(stress[0], stress[1])[1]

# record the state variables in a numpy array in the following order
# column: eps_p, eps_q, p, q, eps_1, eps_2, sigma_1, sigma_2, epor
# every row represents a time step
data = numpy.array([[strain[0], strain[1], stress[0], stress[1], eps_1, eps_2, sigma_1,
sigma_2, epor, stateVar[1] ]])

# initialise the loop
step = 1
while (stress[0] < p_max) and (strain[1] < epsq_max) and (step < i_max):
   dstress, dstrain, dStateVar = rates(stress,strain,stateVar,testType,model,modelParam)
   stress, strain, stateVar = integration(stress,strain,dstress,dstrain,stateVar,dStateVar,dt)
   data = recordData(data,stress,strain,stateVar)
   step += 1

# open output file for saving the data and the plot
filename = 'output_' + testType + 'expDataV.S.DP'+'_TMD4_dense'

# save results to a text file
printResults(filename,data)
data2 = expData()[4]

# plot the results
plotting_testData(data, testType='txd', outputName=filename, data2=data2,
data3=None)
```




**Results analysis:**

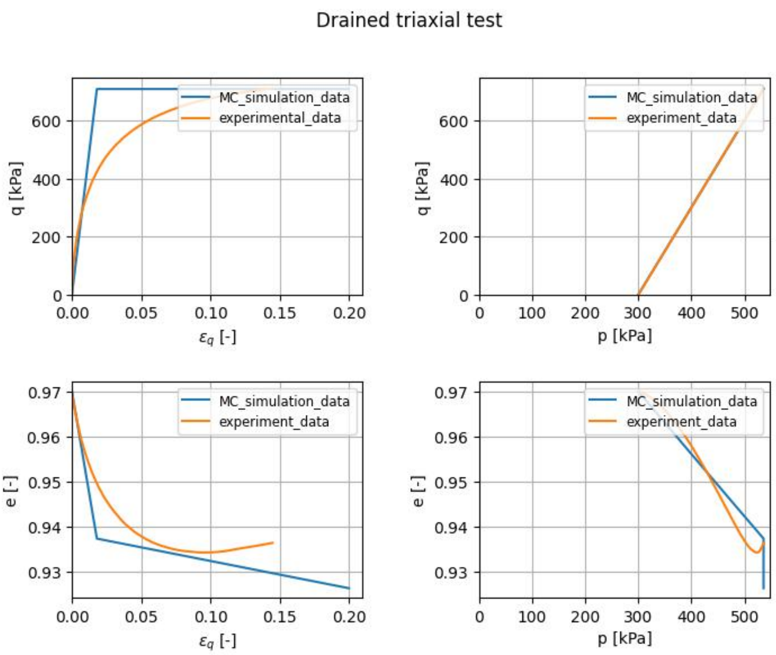
The simulation of DP model also looks good. And the results looks very similar to the MC model, the reason is that we only simulate the compression test.

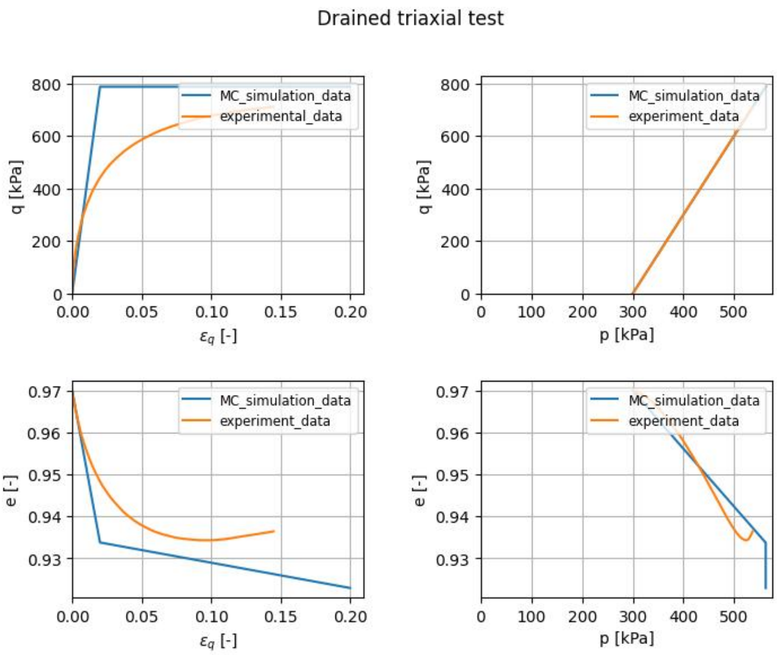
**c) MC model_loose soil_txd**

```
import numpy, sys
# import additional functions
from functions_elementTest import *
from functions_matModels import *
from plotting_routines import *
from dataExperiment import *

#================================
# Input
#================================

# initial state
epor = 0.97   # void ratio
p = 300          # [kPa]
q = 1.16       # [kPa]
eps_1 = 0      # [-]
eps_2 = 0     # [-]

# material models for the elastic and the plastic part
# "linelast" ... linear elasticity
model_el = "linelast"
# "mc"  ... mohr-coulomb
# "dp" ... drucker-prager
model_pl = "mc"
```

```
# model_pl = "dp"
model={"model_el":model_el, "model_pl":model_pl}

# material parameters
E = 30030.6
nu = 0.146
phi = 33
psi = -0.9
modelParam={"E":E, "nu":nu, "phi":phi, "psi":psi}

# test control
# "oed" ... oedometer test
# "txd" ... drained triaxial test
# "txu" ... undrained triaxial test
testType = 'txd'

# stop criteria for calculation
p_max = 1000   # [kPa]
epsq_max = 0.2# [%]
i_max = 1000    # max iterations

# numerical parameter
dt = 0.001         # time step (numerical integration)

#=====================================================================
==========
# Calculation
#=====================================================================
==========

# inital stress and strain in volumetric and deviatoric invariants
stress = numpy.array([p, q])
strain = eps2pq(eps_1,eps_2)
stateVar = numpy.array([epor,0])
sigma_1 = pq2sigma(stress[0], stress[1])[0]
sigma_2 = pq2sigma(stress[0], stress[1])[1]

# record the state variables in a numpy array in the following order
# column: eps_p, eps_q, p, q, eps_1, eps_2, sigma_1, sigma_2, epor
# every row represents a time step
data = numpy.array([[strain[0], strain[1], stress[0], stress[1], eps_1, eps_2, sigma_1,
sigma_2, epor, stateVar[1] ]])

# initialise the loop
step = 1
while (stress[0] < p_max) and (strain[1] < epsq_max) and (step < i_max):
   dstress, dstrain, dStateVar =
rates(stress,strain,stateVar,testType,model,modelParam)
   stress, strain, stateVar =
integration(stress,strain,dstress,dstrain,stateVar,dStateVar,dt)
   data = recordData(data,stress,strain,stateVar)
   step += 1
```

```
# open output file for saving the data and the plot
filename = 'output_' + testType + 'expDataV.S.MC'+'_TMD4_loose'

# save results to a text file
printResults(filename,data)
data2 = expData()[5]

# plot the results
plotting_testData(data, testType='txd', outputName=filename, data2=data2,
data3=None)
```




**Results analysis:**

**1)** The only difference about the 2 diagrams above is phi, the upper one phi=33, and the lower one is phi=34(the results from taskA's calibration). Although both simulation results seems not bad for comparison with the experimental results, obviously the lower one seems better than the upper. And the only slight calibration and can get the better simulation.

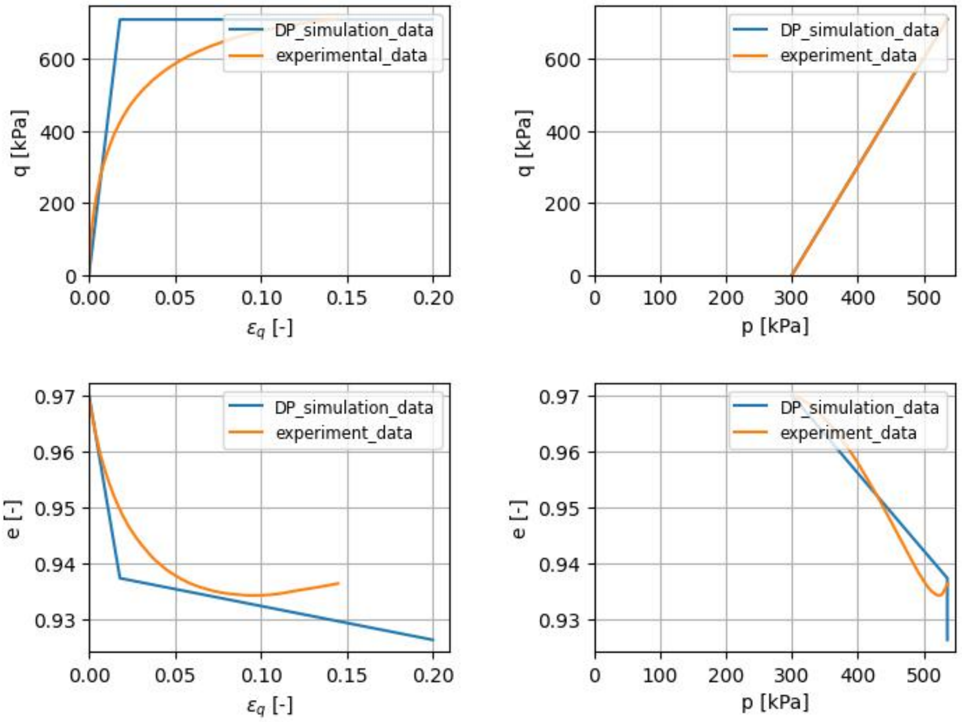
**2)** Compare with the dense soil simulation results that we got above, from the q-eps_q diagram, the friction angle of dense soil is larger than the loose. And from e-eps_q, the big difference is the plastic part, the dense soil exhibits dialatancy and loose soil exhibits contraction. And seem in e-p diagram, for loose soil, with p increasing, the void ratio decrease significantly, but for dense soil, with p increasing, the void ratio change little at the beginning, after about p=690kpa, the void ratio increase significantly.

**d) DP model_loose soil_txd**

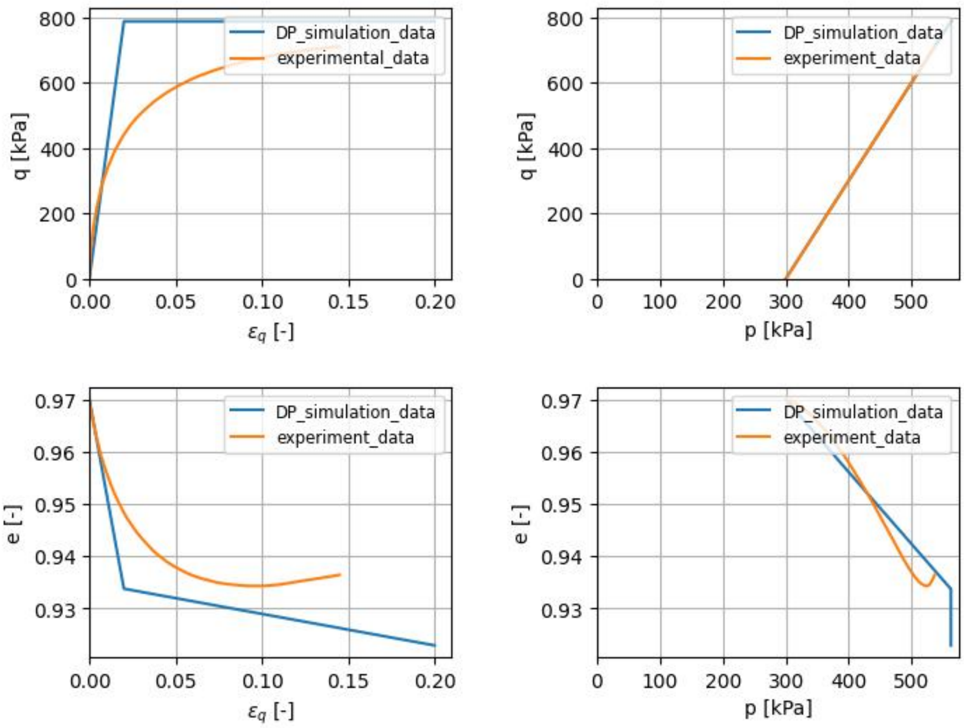
```
import numpy, sys
# import additional functions
from functions_elementTest import *
from functions_matModels import *
from plotting_routines import *
from dataExperiment import *

#================================
# Input
#================================

# initial state
epor = 0.97   # void ratio
p = 300           # [kPa]
q = 1.16       # [kPa]
eps_1 = 0      # [-]
eps_2 = 0     # [-]

# material models for the elastic and the plastic part
# "linelast" ... linear elasticity
model_el = "linelast"
# "mc"  ... mohr-coulomb
# "dp" ... drucker-prager
# model_pl = "mc"
model_pl = "dp"
model={"model_el":model_el, "model_pl":model_pl}

# material parameters
E = 30030.6
nu = 0.146
phi = 33
psi = -0.9
modelParam={"E":E, "nu":nu, "phi":phi, "psi":psi}

# test control
# "oed" ... oedometer test
# "txd" ... drained triaxial test
# "txu" ... undrained triaxial test
testType = 'txd'

# stop criteria for calculation
p_max = 1000   # [kPa]
epsq_max = 0.2# [%]
i_max = 1000    # max iterations

# numerical parameter
```

```
dt = 0.001        # time step (numerical integration)

#================================================================
==========
# Calculation
#================================================================
==========

# inital stress and strain in volumetric and deviatoric invariants
stress = numpy.array([p, q])
strain = eps2pq(eps_1,eps_2)
stateVar = numpy.array([epor,0])
sigma_1 = pq2sigma(stress[0], stress[1])[0]
sigma_2 = pq2sigma(stress[0], stress[1])[1]

# record the state variables in a numpy array in the following order
# column: eps_p, eps_q, p, q, eps_1, eps_2, sigma_1, sigma_2, epor
# every row represents a time step
data = numpy.array([[strain[0], strain[1], stress[0], stress[1], eps_1, eps_2, sigma_1,
sigma_2, epor, stateVar[1] ]])

# initialise the loop
step = 1
while (stress[0] < p_max) and (strain[1] < epsq_max) and (step < i_max):
   dstress, dstrain, dStateVar =
rates(stress,strain,stateVar,testType,model,modelParam)
   stress, strain, stateVar =
integration(stress,strain,dstress,dstrain,stateVar,dStateVar,dt)
   data = recordData(data,stress,strain,stateVar)
   step += 1

# open output file for saving the data and the plot
filename = 'output_' + testType + 'expDataV.S.DP'+'_TMD4_loose'

# save results to a text file
printResults(filename,data)
data2 = expData()[1]

# plot the results
plotting_testData(data, testType='txd', outputName=filename, data2=data2,
data3=None)
```

Drained triaxial test




Drained triaxial test




**Results analysis:**

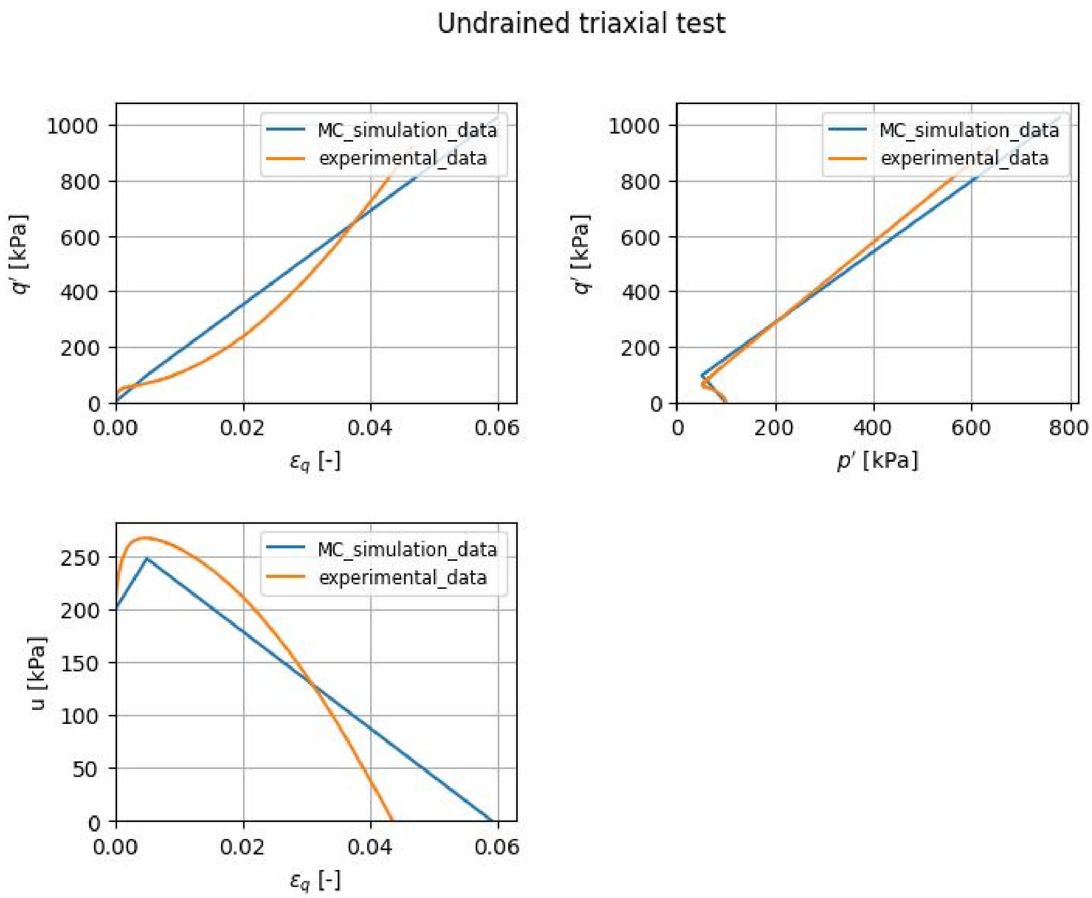
The simulation of DP model also looks good. And the results looks very similar to the MC model, the reason is that we only simulate the compression test.

**e) MC model_txu**

```
import numpy, sys
# import additional functions
from functions_elementTest import *
from functions_matModels import *
from plotting_routines import *
```

```
from dataExperiment import *

#================================
# Input
#================================

# initial state
epor = 0.828    # void ratio
p = 100            # [kPa]
q = 0.559      # [kPa]
eps_1 = 0       # [-]
eps_2 = 0      # [-]
# material models for the elastic and the plastic part
# "linelast" ... linear elasticity
model_el = "linelast"
# "mc"  ... mohr-coulomb
# "dp" ... drucker-prager
model_pl = "mc"
# model_pl = "dp"
model={"model_el":model_el, "model_pl":model_pl}

# material parameters
E = 25647.6      #[kPa] ... Young's modulus
nu = 0.325        # [-] ... Poisson's ratio
phi = 35            # [°] ... friction angle
psi = 16.2          # [°] ... dilatancy angle
modelParam={"E":E, "nu":nu, "phi":phi, "psi":psi}

# test control
# "oed" ... oedometer test
# "txd" ... drained triaxial test
# "txu" ... undrained triaxial test
testType = 'txu'

# stop criteria for calculation
p_max = 1000   # [kPa]
epsq_max = 0.06       # [%]
i_max = 1000    # max iterations

# numerical parameter
dt = 0.001         # time step (numerical integration)

#========================================================================
==========
# Calculation
#========================================================================
==========

# inital stress and strain in volumetric and deviatoric invariants
stress = numpy.array([p, q])
strain = eps2pq(eps_1,eps_2)
```

```
stateVar = numpy.array([epor,0])
sigma_1 = pq2sigma(stress[0], stress[1])[0]
sigma_2 = pq2sigma(stress[0], stress[1])[1]

# record the state variables in a numpy array in the following order
# column: eps_p, eps_q, p, q, eps_1, eps_2, sigma_1, sigma_2, epor
# every row represents a time step
data = numpy.array([[strain[0], strain[1], stress[0], stress[1], eps_1, eps_2, sigma_1, sigma_2, epor, stateVar[1] ]])

# initialise the loop
step = 1
while (stress[0] < p_max) and (strain[1] < epsq_max) and (step < i_max):
   dstress, dstrain, dStateVar = rates(stress,strain,stateVar,testType,model,modelParam)
   stress, strain, stateVar = integration(stress,strain,dstress,dstrain,stateVar,dStateVar,dt)
   data = recordData(data,stress,strain,stateVar)
   step += 1

# open output file for saving the data and the plot
filename = 'output_' + testType + 'expDataV.S.MC'+'_TMU1'

# save results to a text file
printResults(filename,data)
data2 = expData()[6]

# plot the results
plotting_testData(data, testType='txu', outputName=filename, data2=data2, data3=None)
```



**Results analysis:**

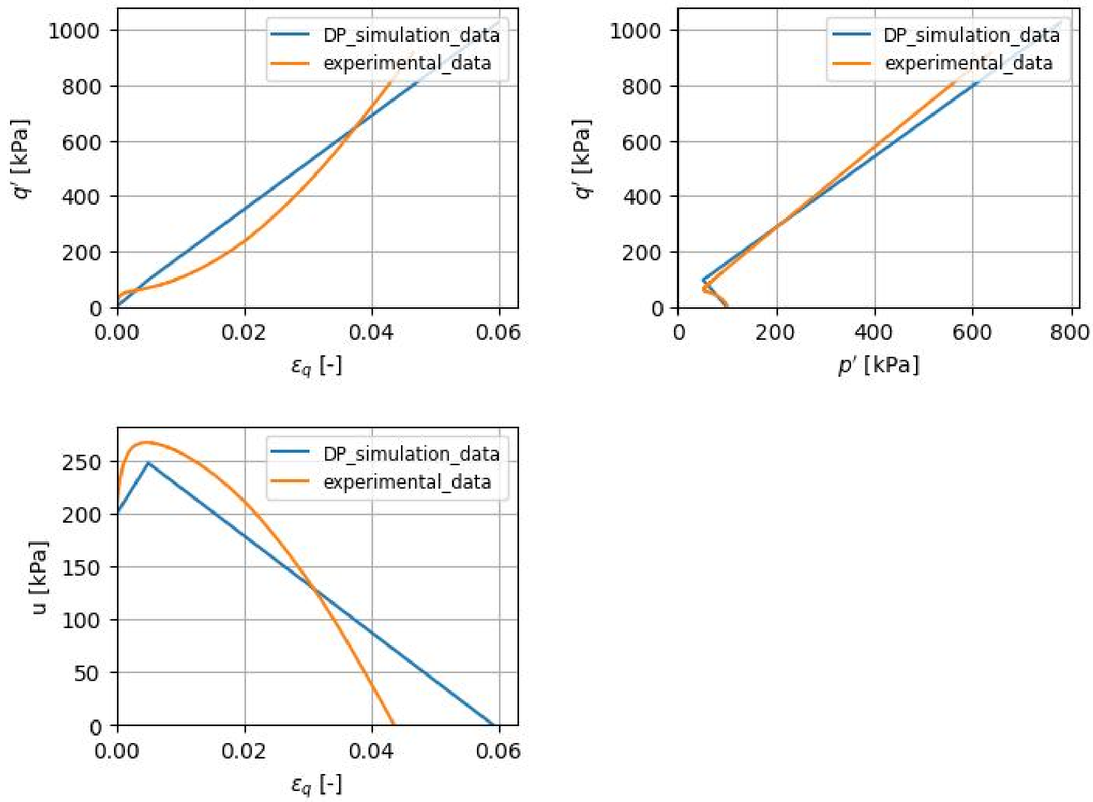
Compare with the drained triaxial simulation results, the MC simulation in undrained triaxial test looks not at that good, the reason is Mohr-Coulomb model is based on the effective stress, the pore water pressure obviously will have an effect on the simulation results. But the overall trend from the simulation results and experimental data are similar.

**f) DP model_TXU**

```
import numpy, sys
# import additional functions
from functions_elementTest import *
from functions_matModels import *
from plotting_routines import *
from dataExperiment import *

#================================
# Input
#================================

# initial state
epor = 0.828    # void ratio
p = 100            # [kPa]
q = 0.559      # [kPa]
eps_1 = 0       # [-]
eps_2 = 0     # [-]
# material models for the elastic and the plastic part
# "linelast" ... linear elasticity
model_el = "linelast"
# "mc"  ... mohr-coulomb
# "dp" ... drucker-prager
# model_pl = "mc"
model_pl = "dp"
model={"model_el":model_el, "model_pl":model_pl}

# material parameters
E = 25647.6      #[kPa] ... Young's modulus
nu = 0.325       # [-] ... Poisson's ratio
phi = 35           # [°] ... friction angle
psi = 16.2         # [°] ... dilatancy angle
modelParam={"E":E, "nu":nu, "phi":phi, "psi":psi}

# test control
# "oed" ... oedometer test
# "txd" ... drained triaxial test
# "txu" ... undrained triaxial test
testType = 'txu'

# stop criteria for calculation
p_max = 1000   # [kPa]
epsq_max = 0.06       # [%]
i_max = 1000    # max iterations
```

```
# numerical parameter
dt = 0.001         # time step (numerical integration)

#=====================================================================
==========
# Calculation
#=====================================================================
==========

# inital stress and strain in volumetric and deviatoric invariants
stress = numpy.array([p, q])
strain = eps2pq(eps_1,eps_2)
stateVar = numpy.array([epor,0])
sigma_1 = pq2sigma(stress[0], stress[1])[0]
sigma_2 = pq2sigma(stress[0], stress[1])[1]

# record the state variables in a numpy array in the following order
# column: eps_p, eps_q, p, q, eps_1, eps_2, sigma_1, sigma_2, epor
# every row represents a time step
data = numpy.array([[strain[0], strain[1], stress[0], stress[1], eps_1, eps_2, sigma_1,
sigma_2, epor, stateVar[1] ]])

# initialise the loop
step = 1
while (stress[0] < p_max) and (strain[1] < epsq_max) and (step < i_max):
   dstress, dstrain, dStateVar =
rates(stress,strain,stateVar,testType,model,modelParam)
   stress, strain, stateVar =
integration(stress,strain,dstress,dstrain,stateVar,dStateVar,dt)
   data = recordData(data,stress,strain,stateVar)
   step += 1

# open output file for saving the data and the plot
filename = 'output_' + testType + 'expDataV.S.DP'+'_TMU1'

# save results to a text file
printResults(filename,data)
data2 = expData()[6]

# plot the results
plotting_testData(data, testType='txu', outputName=filename, data2=data2,
data3=None)
```

Undrained triaxial test




**Results analysis:**

The simulation results of DP model looks very similar to the MC model, the reason is that we only simulate the compression test.

## 3. Original source code:

**Remark:** see details in python scripts

**functions_elementTest.py:** for define the txd and txu element test

**fnctions_matModels.py:** for extend the Mohr-Coulomb model and the Drucker-Prager model

**plotting_routines.py:** for plot task B

**taskB_TXD.py:**for simulate the two models under drained triaxial test

**taskB_TXU.py:** for simulate the two models under undrained triaxial test