Answers: Exercise Sheet No. 8

1 Sequential modular approach

```python
# -*- coding: utf-8 -*-
"""
Created on Thu Feb 29 12:31:19 2024

@author: rafaed
"""

import autograd.numpy as np
from autograd import jacobian
import scipy.optimize as sp
from numpy import linalg as LA


#parameters
sF = 1.0
sT = 1.0
rho = 50.0
a1 = 5.9755e9
a2 = 2.5962e12
a3 = 9.6283e15
V = 0.03
T =  6.744*sT
Fa = 13.357*sF
Fb = 24.482193508*sF
n = 0.9

#loop initial guess
Fr_a_inp = 4.69
Fr_b_inp = 14.54
Fr_c_inp = 0.769
Fr_e_inp = 14.40
Fr_p_inp = 1.91
Fr_g_inp = 0.318

# List of variables reactor (23)
# Fsum, Fa_eff, Fb_eff, Fc_eff, Fe_eff, Fp_eff, Fg, F_purge, k1, k2, k3, k1e (5.71), k2e(7.58), k3e (8.80),
#  T2 (0.14), r1 (8.516), r2 (7.096), r3 (2.084), xa (0.128), xb (0.397), xc (0.020), xe (0.393), xp (0.052), xg (0.00867), Fr_a (4.69), Fr_b (14.54), Fr_c (0.769), Fr_e (14.40), Fr_p (1.91), Fr_g (0.318)


ini_guess_reactor = np.array ([366.369, 46.907, 145.444, 7.692, 144.033, 19.115, 3.178
                  , 111.7, 567.6, 1270.91521720,5.71,7.58,8.80,0.14, 8.516,7.096, 2.084,0.128, 0.397,0.020,0.393, 0.052,0.00867 ])



def sequential_modular(Fr_a_inp,Fr_b_inp,Fr_c_inp,Fr_e_inp,Fr_p_inp,Fr_g_inp):
# define the residual function
 def reactor (inp):
    Fsum = inp[0]*sF
```

```
    Fa_eff = inp[1]*sF
    Fb_eff= inp[2]*sF
    Fc_eff= inp[3]*sF
    Fe_eff= inp[4]*sF
    Fp_eff= inp[5]*sF
    Fg= inp[6]*sF
    k1= inp[7]
    k2= inp[8]
    k3= inp[9]
    k1e = inp[10]
    k2e = inp[11]
    k3e = inp[12]
    T2 = inp[13]
    r1 = inp[14]
    r2 = inp[15]
    r3 = inp[16]
    xa = inp[17]
    xb = inp[18]
    xc = inp[19]
    xe = inp[20]
    xp = inp[21]
    xg = inp[22]

    res0 = -k1e + np.log(a1) - 120.0*T2
    res1 = -k2e + np.log(a2) - 150.0*T2
    res2 = -k3e + np.log(a3) - 200*T2
    res3 = T2*T - 1.0
    res4 = k1 - np.exp(k1e)
    res5 = k2 - np.exp(k2e)
    res6 = k3 - np.exp(k3e)
    res7 = -r1 + k1*xa*xb*V*rho
    res8 = -r2 + k2*xc*xb*V*rho
    res9 = -r3 + k3*xp*xc*V*rho
    res10 = -Fa_eff + Fa + Fr_a_inp - r1
    res11 = -Fb_eff + Fb + Fr_b_inp - (r1 + r2)
    res12 = -Fc_eff + Fr_c_inp + 2.0*r1 - 2.0*r2 - r3
    res13 = -Fe_eff + Fr_e_inp + 2.0*r2
    res14 = -Fp_eff + 0.1*Fr_e_inp + r2 - 0.5*r3
    res15 = -Fg + 1.5*r3
    res16 = Fa_eff + Fb_eff + Fc_eff + Fe_eff + Fp_eff + Fg - Fsum
    res17 = -Fa_eff + Fsum*xa
    res18 = -Fb_eff + Fsum*xb
    res19 = -Fc_eff + Fsum*xc
    res20 = -Fe_eff + Fsum*xe
    res21 = -Fp_eff + Fsum*xp
    res22 = -Fg + Fsum*xg



    res = np.array([res0 , res1 , res2, res3, res4, res5, res6,
    res7, res8, res9, res10, res11, res12, res13, res14, res15,
    res16, res17, res18, res19, res20, res21, res21, res22],dtype =
    np.float64)

    return  res # return a vector of residuals


# List of variables reactor (6)
#F_p, F_purge, Fr_a , Fr_b, Fr_c, Fr_e , Fr_p, Fr_g
 def separator(inp_sep):

    Fa_eff = inp_sep[0]
```

```python
        Fb_eff = inp_sep[1]
        Fc_eff = inp_sep[2]
        Fe_eff = inp_sep[3]
        Fp_eff = inp_sep[4]
        Fg = inp_sep[5]

        Fp =  Fp_eff - 0.1*Fe_eff
        F_purge =  n*(Fa_eff + Fb_eff + Fc_eff + 1.1*Fe_eff)
        Fr_a =   (1.0-n)*Fa_eff
        Fr_b =   (1.0-n)*Fb_eff
        Fr_c =   (1.0-n)*Fc_eff
        Fr_e =   (1.0-n)*Fe_eff
        Fr_p =   (1.0-n)*Fp_eff
        Fr_g =   (1.0-n)*Fg

        return Fp, F_purge, Fr_a, Fr_b, Fr_c, Fr_e, Fr_p, Fr_g


    jac_reactor = jacobian(reactor) # autograd to calculate the jacobian

    sol_reactor = sp.root (reactor,ini_guess_reactor,jac=jac_reactor, method ="lm") # solve the problem

#print (sol_reactor)

    inp_sep = sol_reactor.x[1:7]


    Fp, F_purge, Fr_a_out, Fr_b_out, Fr_c_out, Fr_e_out, Fr_p_out, Fr_g_out = separator(inp_sep)

    res =np.zeros(6)
    res [0] = Fr_a_inp - Fr_a_out
    res [1] =Fr_b_inp - Fr_b_out
    res [2] =Fr_c_inp - Fr_c_out
    res [3] =Fr_e_inp - Fr_e_out
    res [4] =Fr_p_inp - Fr_p_out
    res [5] =Fr_g_inp - Fr_g_out

    return res, Fr_a_out, Fr_b_out, Fr_c_out, Fr_e_out, Fr_p_out, Fr_g_out

#Convergence loop - Successive Substitutions
tol=1e-2
res_norm = 100.0
i=1
while res_norm > tol:

    print(i)
    i = i +1
    res_k, Fr_a_out, Fr_b_out, Fr_c_out, Fr_e_out, Fr_p_out, Fr_g_out = sequential_modular(Fr_a_inp,Fr_b_inp,Fr_c_inp,Fr_e_inp, Fr_p_inp,Fr_g_inp)


    res_norm = LA.norm(res_k)
    print(res_norm)

    Fr_a_inp = Fr_a_out
    Fr_b_inp = Fr_b_out
    Fr_c_inp = Fr_c_out
```

```
    Fr_e_inp = Fr_e_out
    Fr_p_inp = Fr_p_out
    Fr_g_inp = Fr_g_out

print(Fr_a_out, Fr_b_out, Fr_c_out, Fr_e_out, Fr_p_out, Fr_g_out)
```

1. //
2. //
3. Yes, we could instead make the convergence loop between the two models. An initial guess for F_g and F_eff would be needed.

2 Equation Oriented approach

```python
    # -*- coding: utf-8 -*-
"""
Created on Thu Feb 29 12:31:19 2024

@author: rafaed
"""

import autograd.numpy as np
from autograd import jacobian
import scipy.optimize as sp


#parameters
sF = 1.0
sT = 1.0
rho = 50.0
a1 = 5.9755e9
a2 = 2.5962e12
a3 = 9.6283e15
V = 0.03
T =  6.744*sT
Fa = 13.357*sF
Fb = 24.482193508*sF
n = 0.9

# List of variables (31)
# Fsum, Fa_eff, Fb_eff, Fc_eff, Fe_eff, Fp_eff, Fp, Fg, F_purge, k1
    , k2, k3, k1e (5.71), k2e(7.58), k3e (8.80),
#  T2 (0.14), r1 (8.516), r2 (7.096), r3 (2.084), xa (0.128), xb
    (0.397), xc (0.020), xe (0.393), xp (0.052), xg (0.00867), Fr_a
     (4.69), Fr_b (14.54), Fr_c (0.769), Fr_e (14.40), Fr_p (1.91),
     Fr_g (0.318)


ini_guess = np.array ([366.369, 46.907, 145.444, 7.692, 144.033,
    19.115, 4.712, 3.178, 35.910
                      , 111.7, 567.6,
    1270.91521720,5.71,7.58,8.80,0.14, 8.516,7.096, 2.084,0.128,
    0.397,0.020,0.393, 0.052,0.00867, 4.69, 14.54, 0.769,14.40,
    1.91, 0.318  ])


 # solution = np.array([4.11439856e+01, 6.24409991e+00, 1.25004944e
    +01, 8.72477464e-01,
 #        1.22098633e+01, 3.76628040e+00, 2.54529407e+00, 5.55077018
    e+00,
 #        2.97431293e+01, 1.11870204e+02, 5.68543070e+02, 1.27091522
    e+03,
 #        4.71733931e+00, 6.34307707e+00, 7.14749256e+00, 1.48279953
    e-01,
 #        7.73731008e+00, 5.49443850e+00, 3.70051345e+00, 1.51762155
    e-01,
 #        3.03823127e-01, 2.12054678e-02, 2.96759372e-01, 9.15390267
    e-02,
 #        1.34910853e-01, 6.24409991e-01, 1.25004944e+00, 8.72477464
    e-02,
 #        1.22098633e+00, 3.76628040e-01, 5.55077018e-01])


# define the residual function
```

```
def fun (inp):
    Fsum = inp[0]*sF
    Fa_eff = inp[1]*sF
    Fb_eff= inp[2]*sF
    Fc_eff= inp[3]*sF
    Fe_eff= inp[4]*sF
    Fp_eff= inp[5]*sF
    Fp= inp[6]*sF
    Fg= inp[7]*sF
    F_purge= inp[8]*sF
    k1= inp[9]
    k2= inp[10]
    k3= inp[11]
    k1e = inp[12]
    k2e = inp[13]
    k3e = inp[14]
    T2 = inp[15]
    r1 = inp[16]
    r2 = inp[17]
    r3 = inp[18]
    xa = inp[19]
    xb = inp[20]
    xc = inp[21]
    xe = inp[22]
    xp = inp[23]
    xg = inp[24]
    Fr_a = inp[25]
    Fr_b = inp[26]
    Fr_c = inp[27]
    Fr_e = inp[28]
    Fr_p = inp[29]
    Fr_g = inp[30]



    res0 = -k1e + np.log(a1) - 120.0*T2
    res1 = -k2e + np.log(a2) - 150.0*T2
    res2 = -k3e + np.log(a3) - 200*T2
    res3 = T2*T - 1.0
    res4 = k1 - np.exp(k1e)
    res5 = k2 - np.exp(k2e)
    res6 = k3 - np.exp(k3e)
    res7 = -r1 + k1*xa*xb*V*rho
    res8 = -r2 + k2*xc*xb*V*rho
    res9 = -r3 + k3*xp*xc*V*rho
    res10 = -Fa_eff + Fa + Fr_a - r1
    res11 = -Fb_eff + Fb + Fr_b - (r1 + r2)
    res12 = -Fc_eff + Fr_c + 2.0*r1 - 2.0*r2 - r3
    res13 = -Fe_eff + Fr_e + 2.0*r2
    res14 = -Fp_eff + 0.1*Fr_e + r2 - 0.5*r3
    res15 = -Fg + 1.5*r3
    res16 = Fa_eff + Fb_eff + Fc_eff + Fe_eff + Fp_eff + Fg - Fsum
    res17 = -Fa_eff + Fsum*xa
    res18 = -Fb_eff + Fsum*xb
    res19 = -Fc_eff + Fsum*xc
    res20 = -Fe_eff + Fsum*xe
    res21 = -Fp_eff + Fsum*xp
    res22 = -Fg + Fsum*xg
    res23 =  -Fp_eff + 0.1*Fe_eff + Fp
    res24 = -F_purge + n*(Fa_eff + Fb_eff + Fc_eff + 1.1*Fe_eff)
    res25 = -Fr_a + (1.0-n)*Fa_eff
    res26 = -Fr_b + (1.0-n)*Fb_eff
```

```
    res27 = -Fr_c + (1.0-n)*Fc_eff
    res28 = -Fr_e + (1.0-n)*Fe_eff
    res29 = -Fr_p + (1.0-n)*Fp_eff
    res30 = -Fr_g + (1.0-n)*Fg



    res = np.array([res0 , res1 , res2, res3, res4, res5, res6, res7, res8, res9, res10, res11, res12, res13, res14, res15, res16, res17, res18, res19, res20, res21, res21, res22, res23, res24, res25, res26, res27, res28, res29, res30],dtype =np.float64)

    return  res # return a vector of residuals


jac = jacobian(fun) # autograd to calculate the jacobian

sol = sp.root (fun,ini_guess,jac=jac,method ="lm") # solve the problem


print (sol)
```

1. We can use the solution of the sequential modular approach as an initial guess.
2. //
3. ODE (DAE) equations with the initial conditions. The equation-oriented approach would be the most suitable method since performing the convergence loop for each point in time would be time-consuming.