Answers: Exercise Sheet No. 7

1 DAE for condenser

1. The differential variables are n_g and T. The algebraic variables are P and L. There is only one algebraic variables in the 2 algebraic equations, thus these equations cannot be used to solve for both algebraic variables. See the slides for an example of how the incidence matrix looks and how to read it.

2. One would have to specify V, volume as an algebraic variable. This would mean that the volume of the tank could change in time, which goes against the description of the unit "fixed-volume" condenser.

3. $\frac{dP}{dt}V = \frac{dn_g}{dt}RT + \frac{dT}{dt}Rn_g$

4. $\frac{dP}{dt}V = (F-L)RT + ((FC_P(T_0 - T) + \lambda L - hA_t(T - T_c))/n_g C_P)Rn_g$ and $\frac{dP}{dt} = \frac{AB}{T^2}\exp(\frac{-B}{T})((FC_P(T_0 - T) + \lambda L - hA_t(T - T_c))/n_g C_P)$

5. Index 1.

2 Collocation

2.1 Coding

```python
'''
This is an example of how the orthogonal collocation method can be
    used to solve an ODE.

@author: emturan
'''
import autograd.numpy as np
import matplotlib.pyplot as plt
from autograd import jacobian
from scipy import optimize
from scipy.integrate import solve_ivp

# t_points for the collocation method using 5 Radau points
five_rad = np.array([0.057104, 0.276843, 0.583590,0.860240,1.])

# 5 uniform points
five_uni = np.array([0.2, 0.4, 0.6, 0.8, 1.])

# t_points for the collocation method using 3 Radau points
three_rad = np.array([0.155051, 0.644949, 1.])

# t_points for the collocation method using 3 uniform points
three_uni = np.array([1/3., 2/3, 1.])

# define an easy and hard test ODE
def dxdt_easy(t,x):
  return x**2-2*x+1
def dxdt_hard(t, x):
  return -10*x + 10*np.cos(t)
# initial condition for both problems
x0= np.array([-3])

def collocation_weights(t_points):
  """
  Calculates the collocation weights for a vector of time points
    t_points.
  """
  n = len(t_points)
  M1 = np.zeros((n, n))
  M2 = np.zeros((n, n))
  for i in range(n):
    for j in range(n):
      M1[i, j] = (j+1)*t_points[i]**j
      M2[i, j] = t_points[i]**(j+1)
  M = M1 @ np.linalg.inv(M2)

  return M


def ortho_colloc(t_points, dxdt, x0):
  '''
  Performs the orthogonal collocation method for a vector of time
    points t_points, a function dxdt, and an initial state x0.
  '''
  n = len(t_points)
  M = collocation_weights(t_points)
  Minv = np.linalg.inv(M)

```

```
  def f_colloc(x_points):
    res = x_points - x0 - Minv @ np.array(list(map(dxdt, t_points, x_points)))
    return res
  jac = jacobian(f_colloc)

  sol = optimize.root(f_colloc, 0.5*np.ones(n)*x0, method='lm', jac=jac, tol=1e-10, options={"xtol":1e-8})
  print(sol) # for debugging

  x_colloc = np.concatenate((x0, sol.x))
  t_colloc = np.concatenate((np.array([0]), t_points))
  return t_colloc, x_colloc



def lagrange_basis_polynomial(t_points, t, k):
    """
    Calculates the k-th Lagrange basis polynomial for a vector of time points t_points evaluated at time t.
    """
    n = len(t_points)
    numerator = 1
    denominator = 1
    for j in range(n):
        if j != k:
            numerator *= (t - t_points[j])
            denominator *= (t_points[k] - t_points[j])
    return numerator / denominator


def continous_solution(t_colloc, x_colloc):
  """
  Calculates the collocation solution for a vector of time and state values
  """
  n = len(t_colloc)
  def c_sol(t):
    output= 0.
    for i in range(n):
      output += x_colloc[i]*lagrange_basis_polynomial(t_colloc,t,i)
    return output

  return c_sol



dxdt=dxdt_easy

five_rad_t, five_rad_x = ortho_colloc(five_rad, dxdt, x0)
five_uni_t, five_uni_x = ortho_colloc(five_uni, dxdt, x0)
three_rad_t, three_rad_x = ortho_colloc(three_rad, dxdt, x0)
three_uni_t, three_uni_x = ortho_colloc(three_uni, dxdt, x0)

fr_cont = continous_solution(five_rad_t, five_rad_x)
tr_cont = continous_solution(three_rad_t, three_rad_x)
fu_cont = continous_solution(five_uni_t, five_uni_x)
tu_cont = continous_solution(three_uni_t, three_uni_x)


t_plot = np.linspace(0, 1, 100)
c_check = solve_ivp(dxdt, [0, 1], x0, dense_output=True,method='
```

```
    LSODA',atol=1e-10,rtol=1e-10)

print( sum(abs(c_check.y[0,:] - tr_cont(c_check.t))) )
print( sum(abs(c_check.y[0,:] - tu_cont(c_check.t))) )
print( sum(abs(c_check.y[0,:] - fr_cont(c_check.t))) )
print( sum(abs(c_check.y[0,:] - fu_cont(c_check.t))) )



plt.plot(t_plot, c_check.sol(t_plot)[0,:], label='scipy.integrate.solve_ivp')
plt.plot(t_plot, fr_cont(t_plot), label='5 Radau points')
plt.plot(t_plot, fu_cont(t_plot), label='5 uniform points')
plt.plot(t_plot, tr_cont(t_plot), label='3 Radau points')
plt.plot(t_plot, tu_cont(t_plot), label='3 uniform points')
plt.legend(loc='lower right')

plt.show()
```

2.2 Theory questions

1. A description of collocation is given above and in the exercise above.

2. In general increasing the number of points for a given scheme of distributing the points (e.g. uniformly spaced, Radau etc.) increases the accuracy of the approximation. The position of the points determine the order of the method, e.g. Radau collocation gives an order of $2N-1$, where N is the number of points. If we examine the total absolute error the we see that the Radau points do better than the uniform points in terms of the error (as expected). If we wanted to improve upon the accuracy of the Radau points we could use Legendre collocation points, however these points are less stable.

3. One can subdivide the integration interval into several smaller collocation intervals. As each interval is smaller the accuracy of the methods will improve.

4. Implicit methods are (in general) better than explicit methods when solving stiff problems. Orthogonal collocation is an implicit method, as the solution requires the solution of a system of equations.

5. Yes, orthogonal collocation could be used to solve DAEs. If one had multiple equations then the same M matrix would be used for each equation, and all the equations would be solve together, e.g. If you had a degree N method and K equations in the DAE/ODE then KN equations would have to be solved simultaneously in `f_colloc`.