Non-Isothermal Reactor Design

(Case #3.1)

Nattana Yumee 6630887321
Pongsakorn Khaosamarng 6630204421
Supphawit Sripusitto 6630323421

April 2025

Contents

Chapter 1

Introduction

The following is the problem statement for our casework project.

The elementary reversible organic liquid phase reaction

$$\mathrm{A} + \mathrm{B} \rightleftharpoons \mathrm{C}$$

is carried out adiabatically in PFR. An equal molar feed in A and B enters at $27°C$ and the volumetric flow rate is 2.5 L/s and $C_{A0} = 0.1$ mol/L.

(a) Plot the conversion and reactor length profiles up to a PFR reactor volume of 10 L when equilibrium constant, $K_C = 10$ L/mol at 450 K. Plot equilibrium conversion profile.

(b) Repeat (a) when heat exchanger is added, $Ua = 96J/(m^3sK)$, and the coolant temperature is constant at $T_a = 450$ K.

(c) Repeat (b) for a co-current heat exchanger for a coolant flowrate of 50 g/s and $C_{pc} = 5\ J \cdot g/K$ and inlet coolant temperature of $T_{a0} = 450$ K. Vary the coolant rate ($10 < m_c < 1000\ g/s$)

(d) Repeat (c) for counter current coolant flow.

Additional information

$H_A^{\circ}(273\ K)$ = -96, $H_B^{\circ}(273\ K)$ = -72, $H_C^{\circ}(273\ K)$ = -190 kJ/mol
$C_{pA} = C_{pB} = 72$,$C_{pC} = 144\ J/(mol \cdot K)$
k = $0.01\ L/mol \cdot s$ at 300 K, $E_a = 48000$ J/mol

Chapter 2

Preparation

2.1 Assumption

1. Liquids are incompressible.
2. No moving part in the system (no shaft work)
3. No heat transfer other than defining in the problem statement
4. Steady state
5. Heat capacity is independent of temperature.

2.2 Completing the Information

Before formulating, we should get as much information as possible. In this problem, it is obvious that we need the heat of reaction at reference temperature and ΔC_p which could easily be computed from the information in the statement.

$$\Delta H^{\circ}_{rxn} = H^{\circ}_{C} - H^{\circ}_{A} - H^{\circ}_{B} = -190 + 96 + 72 = -22\ kJ/mol$$

$$\Delta C_p = C_{pC} - C_{pA} - C_{pB} = 144 - 72 \times 2 = 0$$

We can conclude here, that ΔC_p is zero, so the heat of reaction does not depend on the temperature of the system.

2.3 Formulation

In this step, we are going to formulate all relations and state all of boundary conditions of each sub-problems.

2.3.1 Problem (a)

Rate law:

$$\begin{aligned} -r_A &= k\left(C_A C_B - \frac{C_C}{K_C}\right) \\ &= k\left(C_{A0}^2(1-X_A)^2 - \frac{C_{A0}X_A}{K_C}\right) \end{aligned}$$

Design Equation:

$$\begin{aligned} \frac{dF_A}{dV} &= r_A \\ \frac{d}{dV}F_{A0}(1-X_A) &= r_A \\ F_{A0}\frac{dX_A}{dV} &= -r_A \\ \frac{dX_A}{dV} &= \frac{-r_A}{F_{A0}} \end{aligned}$$

Arrhenius Equation:

$$k = k_0 exp\left(\frac{E_a}{R}\left(\frac{1}{T_0} - \frac{1}{T}\right)\right)$$

Van't Hoff Equation:

$$\begin{aligned} \frac{d\ln K_C}{dT} &= -\frac{\Delta H^{\circ}_{rxn}}{RT^2} \\ \ln\left(\frac{K_C}{K_{C0}}\right) &= -\frac{\Delta H^{\circ}_{rxn}}{R}\left(\frac{1}{T_0} - \frac{1}{T}\right) \\ K_C &= K_{C0} exp\left(\frac{\Delta H^{\circ}_{rxn}}{R}\left(\frac{1}{T} - \frac{1}{T_0}\right)\right) \end{aligned}$$

Energy balance:

We need to write T in terms of X_A to be able to solve the system of ODEs, since it has a truly long derivation from the general energy balance equation, so we skip to this form.

$$T = T_0 - \frac{\Delta H^{\circ}_{rxn} X_A}{C_{pA} + C_{pB}}$$

All of these equations could solve for T and X_A, but we also need to solve for X_e, so we need to set an equilibrium equation for it.
At Equilibrium:

$$
\begin{aligned}
r_A &= 0 \\
C_{A0}^2(1-X_e)^2 &= \frac{X_e C_{A0}}{K_C} \\
K_C C_{A0}(X_e^2 - 2X_e + 1) &= X_e \\
K_C C_{A0} X_e^2 - (2K_C C_{A0} + 1)X_e + K_C C_{A0} &= 0 \\
X_e &= \frac{2K_C C_{A0} + 1 - \sqrt{(2K_C C_{A0}+1)^2 - 4K_C^2 C_{A0}^2}}{2K_C C_{A0}} \\
X_e &= 1 + \frac{1}{2K_C C_{A0}} - \frac{\sqrt{4K_C C_{A0}+1}}{2K_C C_{A0}}
\end{aligned}
$$

Initial condition at V = 0:

1. $X_A = 0$
2. T = 300 K

Other information are miscellaneous.

2.3.2 Problem (b)

Everything are the same as (a), except the energy balance, our reactor is no longer adiabatic, so the new energy balance equation is the ODE.

$$\frac{dT}{dV} = \frac{Ua(T_a - T) - (-r_A)\Delta H_{rxn}^{\circ}}{F_{A0}(C_{pA} + C_{pB})}$$

while T_a which is coolant temperature is constant at 450 K

2.3.3 Problem (c)

In this problem, T_a is no longer constant, so we must do an energy balance on the coolant.

$$\frac{dT_a}{dV} = \frac{Ua(T - T_a)}{\dot{m}C_{pc}}$$

Other relations are still the same as (b).

2.3.4 Problem (d)

This problem would be the most challenging problem if we did it without programming skill, since it is no longer an IVP, it is a BVP which is harder to solve with common executable software.

Let us look at the easiest point, which is the energy balance before we get into the boundary condition, which is more difficult to understand.

$$\frac{dT_a}{dV} = \frac{Ua(T_a - T)}{\dot{m}C_{pc}}$$

At V = 0 we could no longer say that T_a = 450 K, since the coolant entrance is at V = 10 L, so we need to perform a trial and error until we get T_a at V = 10 L is 450 K without any clue except temperature at V = 10 L of (c) to be an initial guess.

Chapter 3

Programming

Polymath is one of the great software which is easy to solve system of ODEs like these problems, but we need to pay 15$ for standard license. Furthermore, polymath utilize a lot of memory and difficult to customize the results. Hence, we are going to program an simple software to solve these problems from scrap by Python language.

3.1 Odeint

Odeint is the powerful library in the scipy package in python which helps us to solve IVP by the Euler method, so we do not need to code a function to solve an IVP by ourselves. To use this package, make sure that we install it on our computer by typing in the computer terminal with `pip install scipy`. In addition, we need more packages to help us manage array and make some plots, these packages are numpy and matplotlib, which can be installed by pip similar to scipy.

3.2 Problem (a)

First, we need to import all packages to our file. Then, we define our ODE function or our model which has dependent variable, independent variable as parameters, this function must return the derivative of response variable in form of number or array. Meanwhile, we assign values to the constants that are needed. Next, we must assign the space of independent variable as an array using the `linspace(start, stop, number of points)` function. Before solving the system, we must define initial conditions as numbers or iterable objects. To solve the system, we use `odeint(ODE function, initial condition, space)` to return the solution array to a variable. Since we need to solve for equilibrium conversion too, but this function returns us only actual conversion, so we use element-wise properties of numpy array to calculate temperature, equilibrium constant, and equilibrium conversion of each points on space.

Now, we use matplotlib to plot some plots using `matplotlib.plot(y,x)` then display the plot by `matplotlib.show()`

```
from scipy.integrate import odeint
import numpy as np
import matplotlib.pyplot as plt

#rate constant at 300 K Lmol-1s-1
k0 = 0.01
#Equilibrium constant at 450 K Lmol-1
K0 = 10
#Heat of Reaction kJ/mol
Hrx = -190+72+96
#initial condition
X0 = 0
V = np.linspace(0,10,100)

def Model1(x,V):
    #Energy Balance
    T = 300 - Hrx*1e3*x/(72+72)
    #Arrhenius Equation
    k = k0*np.exp(48000/8.3145*(1/300-1/T))
    #Van't Hoff Equation
    K = K0*np.exp(Hrx*1e3/8.3145*(1/T-1/450))
    #Design Equation
    dX = k*(0.1*0.1*(1-x)**2-.1*x/K)/.25

    return dX

X = odeint(Model1,X0,V)
LT = 300 - Hrx*1e3*X/(72+72)
LK = K0*np.exp(Hrx*1e3/8.3145*(1/LT-1/450))
Xe = 1+.5/.1/LK-np.sqrt(4*LK*.1+1)*.5/.1/LK

plt.plot(V,X)
plt.xlabel("Volume (L)")
plt.ylabel("Conversion")
plt.show()

plt.plot(V,Xe)
plt.xlabel("Volume (L)")
plt.ylabel("Equilibrium Conversion")
plt.show()

#For comparison
plt.plot(V,X,label = "Actual Conversion")
plt.plot(V,Xe,label = "Equilibrium Conversion")
plt.xlabel("Volume (L)")
plt.ylabel("Conversion")
plt.legend()
plt.show()
```

3.3 Problem (b)

In this problem, there are few changes from the previous code, we add the energy balance equation to the function and modify the initial condition to be a tuple `(0,300)`, so the odeint will give us an array of solutions of each reactor sizes with conversions stored in the zeroth column and temperatures stored in the first.

```
#Heat transfer coefficient J/m^3sK
Ua = 96
#initial condition
L0 = (0,300)
V = np.linspace(0,10,100)

def Model2(L,V):
    #x and T is elements of list L
    x,T = L
    #Arrhenius Equation
    k = k0*np.exp(48000/8.3145*(1/300-1/T))
    #Van't Hoff Equation
    K = K0*np.exp(Hrx*1e3/8.3145*(1/T-1/450))
    #rate law
    minusrA = k*(0.1*0.1*(1-x)**2-.1*x/K)
    #Energy Balance
    dT = (Ua*(450-T)-minusrA*Hrx*1e3)/.25/(72+72)
    #Design Equation
    dX = minusrA/.25

    return [dX,dT]

L = odeint(Model2,L0,V)
LK = K0*np.exp(Hrx*1e3/8.3145*(1/L[:,1]-1/450))
Xe = 1+.5/.1/LK-np.sqrt(4*LK*.1+1)*.5/.1/LK

plt.plot(V,Xe,label = "Equilibrium Conversion")
plt.plot(V,L[:,0],label = "Actual Conversion")
plt.xlabel("Volume (L)")
plt.ylabel("Conversion")
plt.legend()
plt.show()

plt.plot(V,L[:,1], label = "Reactor Temperature")
plt.plot([0,10],[450,450], label = "Ambient Temperature")
plt.xlabel("Volume (L)")
plt.ylabel("Temperature (K)")
plt.legend()
plt.show()
```

3.4 Problem (c)

As we said before, the coolant temperature is no longer constant, so we add the coolant system energy balance equation to the function while others are the same as in the previous problem. In addition, we store the final temperature at V = 10 L to be an initial guess for the next problem in `Ta0g` variable.

```
#initial condition
L0 = (0,300,450)
V = np.linspace(0,10,100)

def Model3(L,V):
    #x and T is elements of list L
    x,T,Ta = L
    #Arrhenius Equation
    k = k0*np.exp(48000/8.3145*(1/300-1/T))
    #Van't Hoff Equation
    K = K0*np.exp(Hrx*1e3/8.3145*(1/T-1/450))
    #rate law
    minusrA = k*(0.1*0.1*(1-x)**2-.1*x/K)
    #Energy Balance
    dT = (Ua*(Ta-T)-minusrA*Hrx*1e3)/.25/(72+72)
    dTa = Ua*(T-Ta)/5/50
    #Design Equation
    dX = minusrA/.25

    return [dX,dT,dTa]

L = odeint(Model3,L0,V)

#This final temperature will be initial guess of coolant
   temperature of counter current heat exchanger.
Ta0g = L[-1,-1]

LK = K0*np.exp(Hrx*1e3/8.3145*(1/L[:,1]-1/450))
Xe = 1+.5/.1/LK-np.sqrt(4*LK*.1+1)*.5/.1/LK

plt.plot(V,Xe,label = "Equilibrium Conversion")
plt.plot(V,L[:,0],label = "Actual Conversion")
plt.xlabel("Volume (L)")
plt.ylabel("Conversion")
plt.legend()
plt.show()

plt.plot(V,L[:,1],label = "Reactor Temperature")
plt.plot(V,L[:,-1], label = "Ambient Temperature")
plt.xlabel("Volume (L)")
plt.ylabel("Temperature (K)")
plt.legend()
plt.show()

```

```
# For comparing with countercurrent heat exchanger
Xcoc = L[:,0]
```

After we solved at a coolant flow rate equal to 50 g/s, now we fixed the reactor volume at 10 L and varying coolant flow rates from 10 to 1000 g/s. This is more challenging, since we could not change local parameter to the function directly, so the looping method is the suitable one. We set an initial value `m = 10` and assign the space `M = linspace(10,1000,100)` then we run over a components in M to calculate conversions and temperatures at every point in the space to store in the null array which is the solution array. Similarly to the first part of this problem, we store the final coolant temperature as an initial guess of the next problem in an array `Ta0`. We also reduce the number of points in the volume space to 2, for managing memory and process time to be more efficient, since we need only the value at 10 L of the reactor, so there is no need to be fine as in the previous part.

```
#initial condition
L0 = (0,300,450)

V = np.linspace(0,10,2)
M = np.linspace(10,1000,100)
S = []
m = 10
#This list is to store initial guess to next session.
Ta0 = []

def Model3(L,V):
    #x and T is elements of list L
    x,T,Ta = L
    #Arrhenius Equation
    k = k0*np.exp(48000/8.3145*(1/300-1/T))
    #Van't Hoff Equation
    K = K0*np.exp(Hrx*1e3/8.3145*(1/T-1/450))
    #rate law
    minusrA = k*(0.1*0.1*(1-x)**2-.1*x/K)
    #Energy Balance
    dT = (Ua*(Ta-T)-minusrA*Hrx*1e3)/.25/(72+72)
    dTa = Ua*(T-Ta)/5/m
    #Design Equation
    dX = minusrA/.25

    return [dX,dT,dTa]

for i in M:
    m = i
    Si = odeint(Model3,L0,V)
    S.append(Si[-1,0:2])
    Ta0.append(Si[-1,1])

S = np.array(S)
```

```
T = S[:,-1]
LK = K0*np.exp(Hrx*1e3/8.3145*(1/T-1/450))
Xe = 1+.5/.1/LK-np.sqrt(4*LK*.1+1)*.5/.1/LK

plt.plot(M,S[:,0], label = "Actual Conversion")
plt.plot(M,Xe,label = "Equilibrium Conversion")
plt.xlabel("Coolant flowrate (g/s)")
plt.ylabel("Conversion")
plt.legend()

plt.show()

#For comparison
Xcocm = S[:,0]
```

3.5 Problem (d)

As we said before, this problem is the most challenging problem in this casework #3.1, if we have a poor programming skill, since we need to use iterative estimation to allow the program to give the final temperature of the coolant equal to 450 K. Our idea to solve this problem efficiently is to use the bisection method which is the basic method that everyone has studied in the 2110101 Computer Programming course, but we will optimize this algorithm to have more efficiency in exchange for a small amount of decreasing in accuracy. We have observed that the final coolant temperature is a nondecreasing function of the initial coolant temperature, so if the final coolant temperature is more than 450 K the guess temperature will be the upper bound. In contrast, if the final coolant temperature is lower than 450 K, the guess coolant temperature will be the lower bound.

To program this, we initialize all significant variables which are upper bound equal to the initial guess from the previous problem, the lower bound is approximately set at 330-350 K, and the squared error is set to 1 K^2 (`e = 1`). Now, we set up the "While Loop" which is iterative if `e > 1e-25`. In this loop, we guess `Tg = (U+L)/2` while U and L are the supremum and infermum of the interval that contain the actual initial coolant temperature, we also calculate the final temperature to compute the error `e = (Tf-450)**2` consequently. To optimize and make this loop work, we set the first condition that if the error is less than our expectation, the loop is ended, otherwise if `Tf > 450` we will set the supremum to be the guess temperature, so the last thing that would occur if these conditions could not be done is `Tf < 450` we will set the infermum to be the guess temperature.

After finishing this algorithm, we could display the result like the previous problem, also display the final temperature to check that it is close to 450 K enough.

```
V = np.linspace(0,10,100)

def Model4(L,V):
    #x and T are elements of list L
    x,T,Ta = L
    #Arrhenius Equation
    k = k0*np.exp(48000/8.3145*(1/300-1/T))
    #Van't Hoff Equation
    K = K0*np.exp(Hrx*1e3/8.3145*(1/T-1/450))
    #rate law
    minusrA = k*(0.1*0.1*(1-x)**2-.1*x/K)
    #Energy Balance
    dT = (Ua*(Ta-T)-minusrA*Hrx*1e3)/.25/(72+72)
    dTa = -Ua*(T-Ta)/5/50
    #Design Equation
    dX = minusrA/.25

    return [dX,dT,dTa]

U = Ta0g
L = 350
e = 1
while e > 1e-25:
    Tg = (U+L)/2
    S0 = (0,300,Tg)
    S = odeint(Model4,S0,V)
    Tf = S[-1,-1]
    e = (450-Tf)**2
    if e <= 1e-25:
        break
    elif Tf > 450:
        U = Tg
    else:
        L = Tg
print(Tf)

LK = K0*np.exp(Hrx*1e3/8.3145*(1/S[:,1]-1/450))
Xe = 1+.5/.1/LK-np.sqrt(4*LK*.1+1)*.5/.1/LK

plt.plot(V,S[:,0], label = "Actual Conversion")
plt.plot(V,Xe, label = "Equilibrium Conversion")
plt.xlabel("Volume (L)")
plt.ylabel("Conversion")
plt.legend()

plt.show()

plt.plot(V,S[:,1], label = "Reactor Temperature")
plt.plot(V,S[:,-1], label = "Ambient Temperature")
plt.xlabel("Volume (L)")
plt.ylabel("Temperature (K)")
```

```
plt.legend()

plt.show()

# For comparison
Xcou = S[:,0]
```

The task is not yet complete, we need to vary the coolant flow rate at 10 L volume of the reactor, which is more difficult than that we did above.

To review, we could solve this part utilizing the "For loop" to run over components in the coolant flow rate space. However, it is necessary to use loop to perform trial and error. Hence, we have to do the nested structure of the loop, which combined two loops that we introduced above into a nested loop which runs by a canonical index instead of component from an array. To avoid the infinite loop situation, we expand the range of [L,U] to be broader.

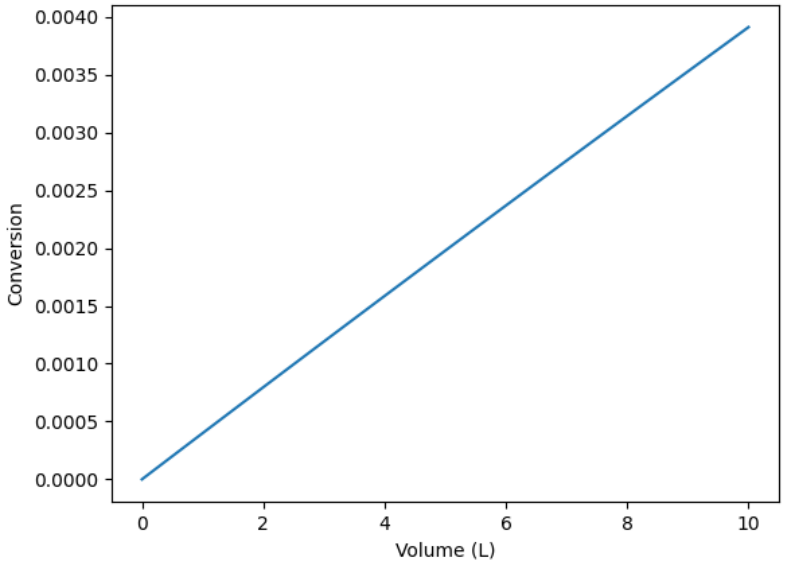
```
V = np.linspace(0,10,2)
M = np.linspace(10,1000,100)
m = 10
def Model4(L,V):
    #x and T is elements of list L
    x,T,Ta = L
    #Arrhenius Equation
    k = k0*np.exp(48000/8.3145*(1/300-1/T))
    #Van't Hoff Equation
    K = K0*np.exp(Hrx*1e3/8.3145*(1/T-1/450))
    #rate law
    minusrA = k*(0.1*0.1*(1-x)**2-.1*x/K)
    #Energy Balance
    dT = (Ua*(Ta-T)-minusrA*Hrx*1e3)/.25/(72+72)
    dTa = -1*Ua*(T-Ta)/5/m
    #Design Equation
    dX = minusrA/.25

    return [dX,dT,dTa]
S = []
for i in range(len(M)):
    m = M[i]
    U = 2*Ta0[i]-330
    L = 330
    e = 1
    while e > 1e-24:
        Tg = (U+L)/2
        S0 = (0,300,Tg)
        Si = odeint(Model4,S0,V)
        Tf = Si[-1,-1]
        e = (450-Tf)**2
        if e <= 1e-24:
            break
        elif Tf > 450:
            U = Tg
```

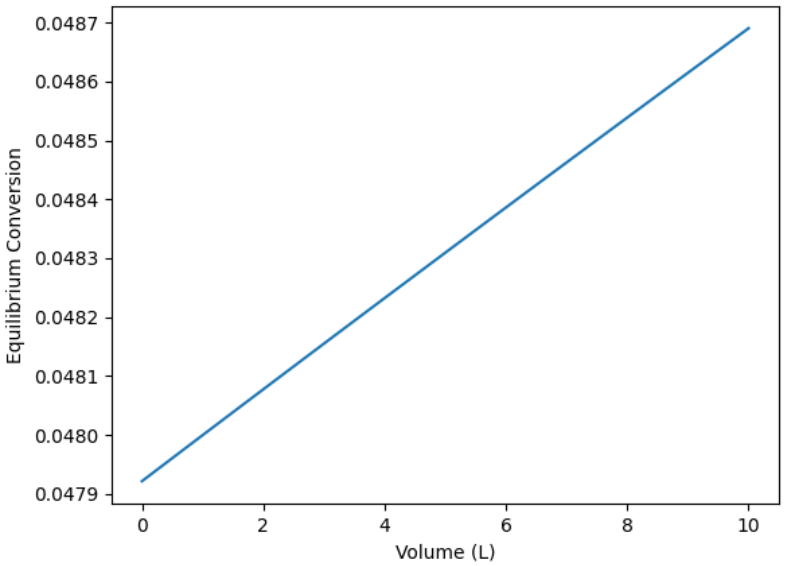
```
        else:
            L = Tg
    S.append(Si[-1,0:2])

S = np.array(S)
T = S[:,-1]
LK = K0*np.exp(Hrx*1e3/8.3145*(1/T-1/450))
Xe = 1+.5/.1/LK-np.sqrt(4*LK*.1+1)*.5/.1/LK

plt.plot(M,S[:,0], label = "Actual Conversion")
plt.plot(M,Xe,label = "Equilibrium Conversion")
plt.xlabel("Coolant flowrate (g/s)")
plt.ylabel("Conversion")
plt.legend()

plt.show()

#For comparison
Xcoum = S[:,0]
```

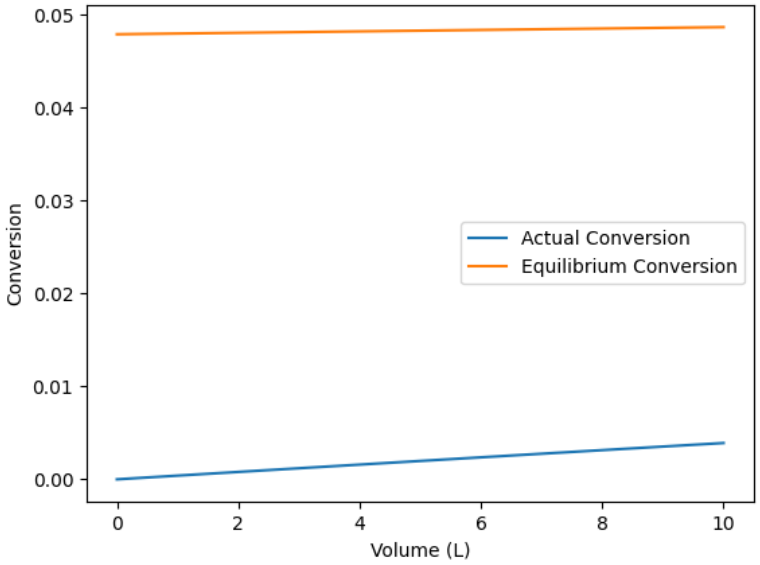
Chapter 4

Results and Conclusions

4.1 Problem (a)

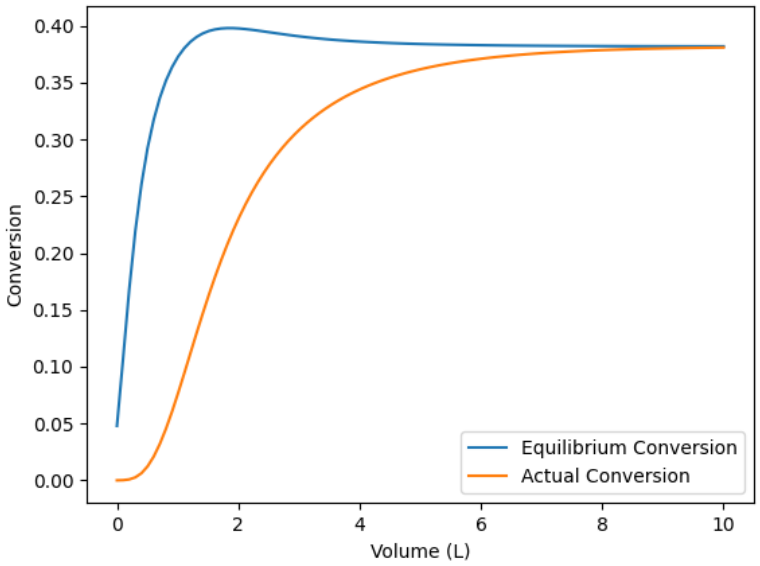


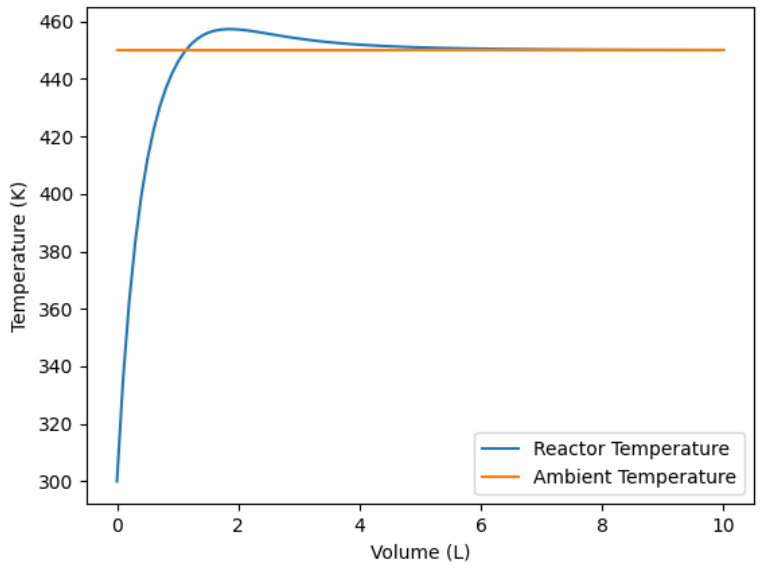




It is obvious that the actual conversion is extremely smaller than the equilibrium conversion, plots could be approximated to linear graphs, since the curvature is truly small compared to this volume space.

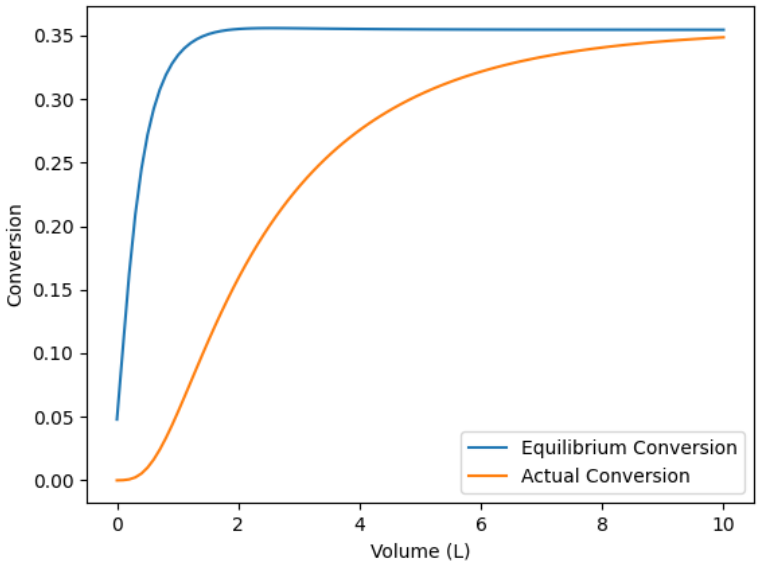
4.2 Problem (b)

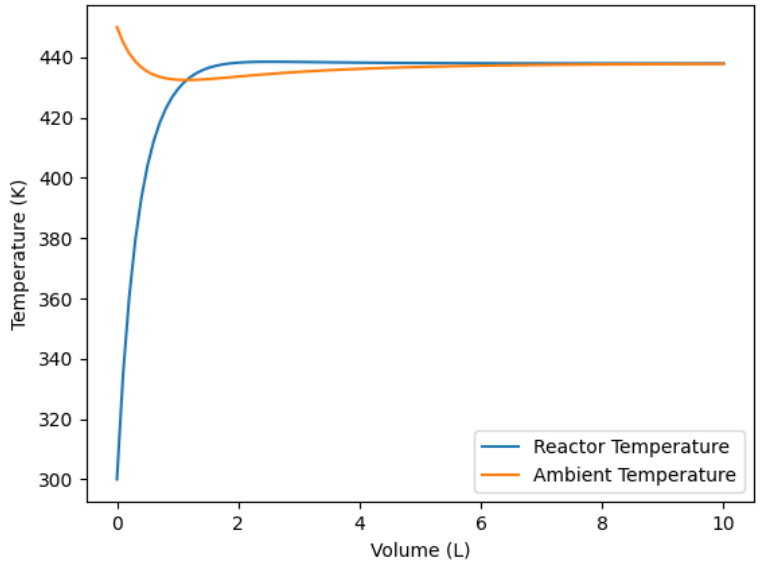


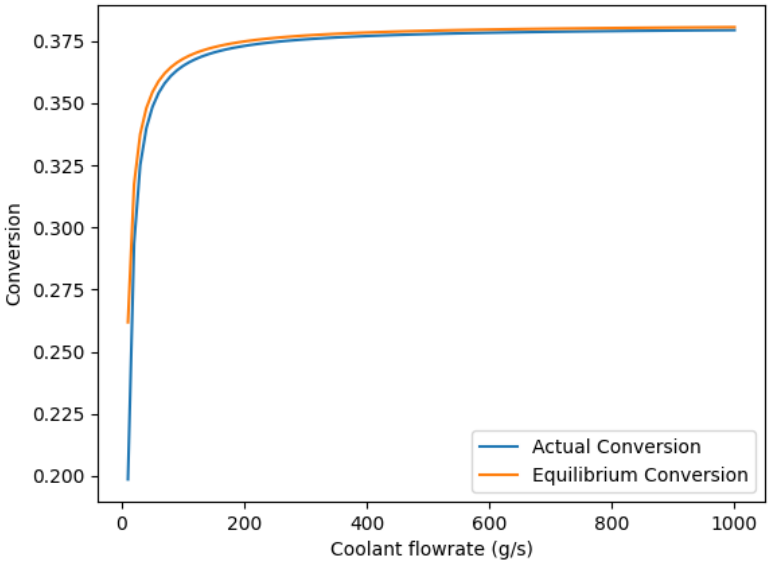


After installing the constant temperature coolant heat exchanger, the actual conversion in stable state is close to the equilibrium conversion. Observe that at the initial rate of reaction has low value, since at the initial rate constant has more influence than the equilibrium constant, after increasing reactor temperature, the equilibrium constant has more influence, so the plot gives this s-curve. Meanwhile in equilibrium conversion plot, it tends to increase while the equilibrium constant decreases, so the equilibrium conversion tends to have a positive correlation to the temperature. That is, the equilibrium conversion increases to the peak at the highest temperature, then the coolant decreases the reactor temperature to be stable at some value, which makes the conversion stable.

4.3 Problem (c)

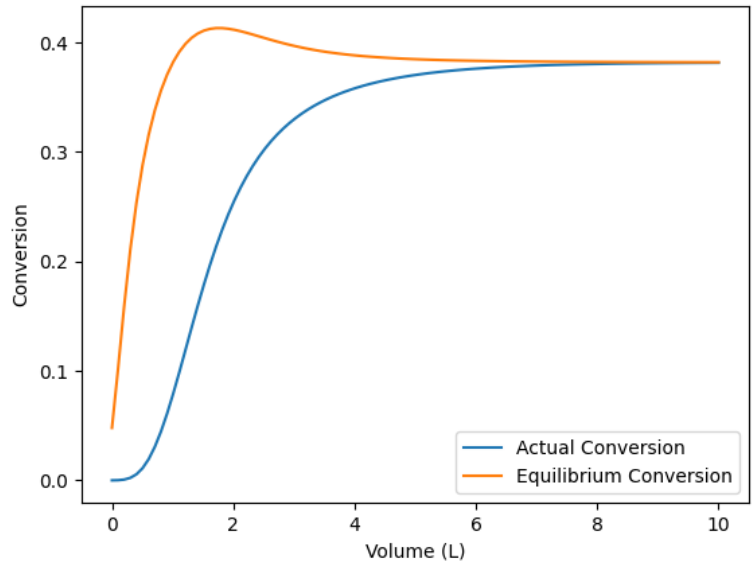

Conversion
Volume (L)
Equilibrium Conversion
Actual Conversion

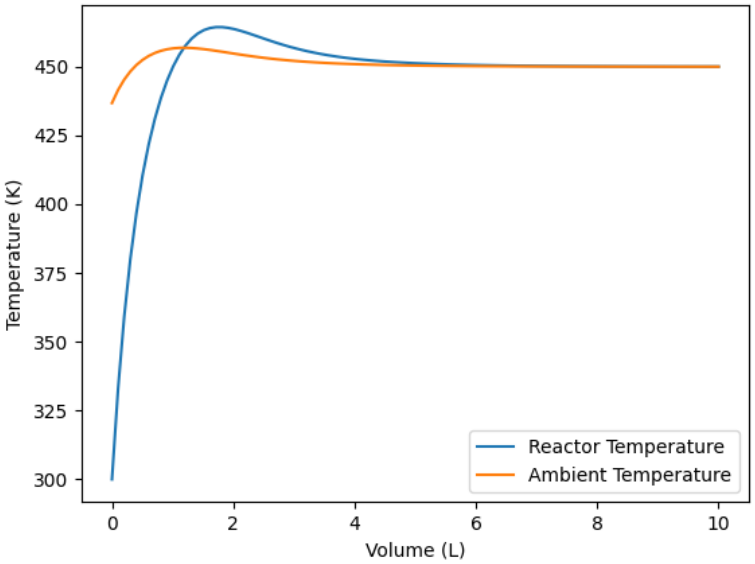

Temperature (K)
Volume (L)
Reactor Temperature
Ambient Temperature

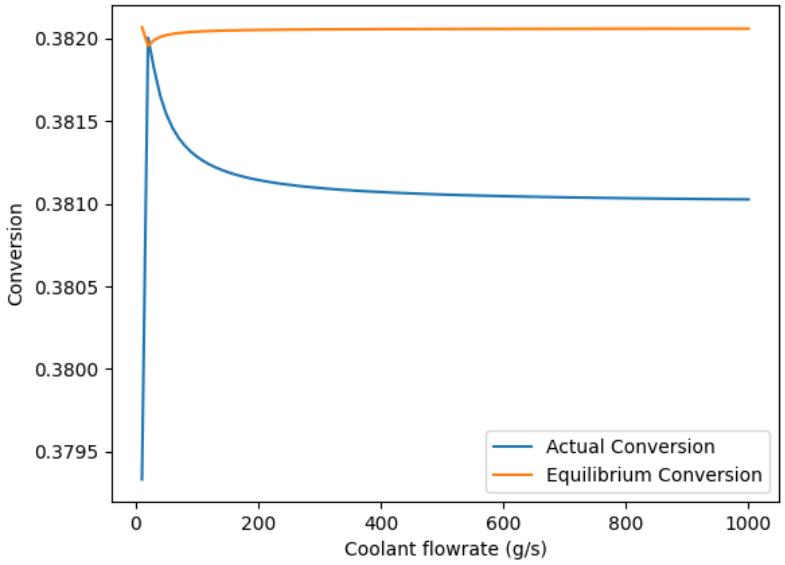


Plots of conversion versus the reactor volume are similar to problem (b), but the maximum conversion is slightly lower, since the coolant temperature is no longer constant, causing poorer heat transfer and the equilibrium conversion is consequently decreasing. Considering the coolant flow rate, it is obvious that conversion is an increasing function of the coolant flow rate. However, it might be approximately stable around $\dot{m} = 450\ g/s$, so there is no need to put more coolant flow rate than that point to significantly change the conversion.

4.4 Problem (d)


0.4
0.3
0.2
0.1
0.0
Conversion
0
2
4
6
8
10
Volume (L)
Actual Conversion
Equilibrium Conversion

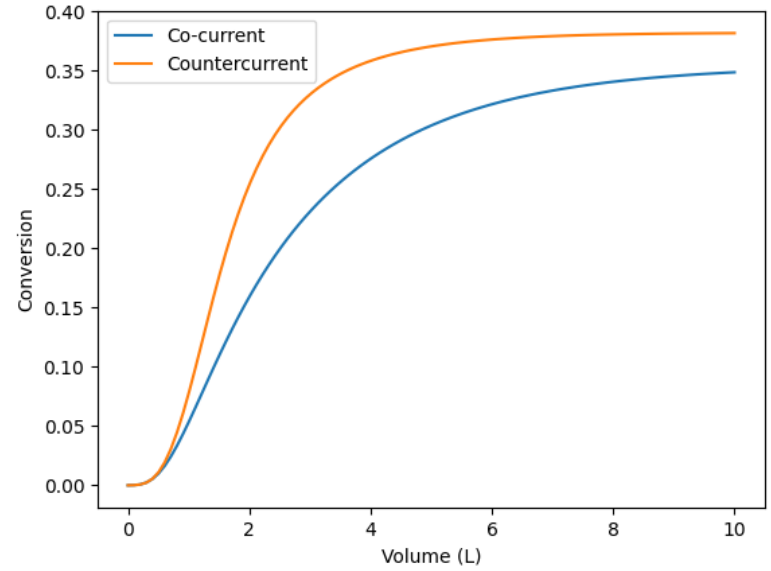

450
425
400
375
350
325
300
Temperature (K)
0
2
4
6
8
10
Volume (L)
Reactor Temperature
Ambient Temperature

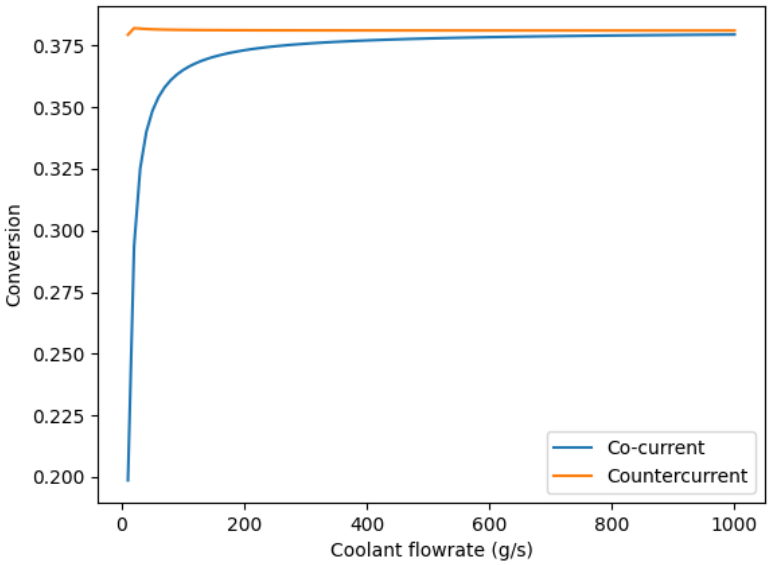


First two graph are similar to the previous, but the latter two are completely different. First, we need to confirm that these peak is not effects of software bug, so we attempt to increase some points in space for smoother line, we found that plots are not significantly change, so that peak is one of the result. After confirming that there is no bug effect on this plot, we could say that at the peak conversion is at maximum, in other words, the actual conversion is equal to the equilibrium conversion. Certainly, we want our reactor to reach that peak, so we use numpy to find the index that has the maximum conversion by `numpy.argmax(array)`, then use that index to display the coolant flow rate. In 100 points space of flow rate, we have found that that point is at 20 g/s, but we think that it is not accurate enough, so we tried for a few tests in 500-1000 points space and found that the optimum flow rate is about 11-12 g/s. This shows us the power of the Python language that Polymath could not do.

4.5 More conclusion about heat exchangers

After knowing the optimal operation state of each type of heat exchanger, we should know which type gives us more conversion under similar conditions, so we plot two plots which are conversion vs volume and conversion vs coolant flow rate to determine whether countercurrent heat exchanger gives us more conversion or not. The programming part of this does not appear in the programming chapter, since we just utilize the same datasets from problems (c) and (d) to plot some comparative plots.





From these plots, we should immediately realize that countercurrent heat exchanger is absolutely better to give high conversion than the co-current.