Process model formulation and solution, 3E4

Computer software tutorial - Tutorial 2

Kevin Dunn, dunnkg@mcmaster.ca **September 2010**

Tutorial objectives

- Some questions to help you feel comfortable deriving model equations for actual chemical engineering systems.
- Brute force solving of equation systems. The rest of the course will focus on better ways to solve these equations.
- Interpreting source code written on paper.

Recap of tutorial rules

- Tutorials can be done in groups of two - please take advantage of this to learn with each other.
- Tutorials must be handed in at the start of class on Wednesday. No electronic submissions - thanks!

Question 1 [2]

Note: Parts 1 and 2 of this question were from the 2006 final exam (slightly modified). It was worth 10% of the 3 hour exam

Consider a mixing tank with fluid volume V where fluids A and B with densities ρ_A and ρ_b, specific heat capacities $C_{p,A}$, $C_{p,B}$ and temperatures T_A and T_B are the inlet streams at volumetric flow rates F_A and F_B. The outlet stream is at a flow rate $F_A + F_B$. The specific heat capacity and density of the outlet streams is given by $C_p = (C_{p,A}F_A + C_{p,B}F_B)/F$ and $\rho = (\rho_A F_A + \rho_B F_B)/F$. The fluid also looses heat from the tank at a rate $q = k_T(T - T_{\text{wall}})$ where T_{wall} is the constant tank wall temperature, k_T is a constant and T denotes the current fluid temperature.

1. Using 3-step modelling approach shown in class, derive a dynamical balance describing the time-dependent exit stream temperature.
2. Can the steady state exit stream temperature be higher than both T_A and T_B? Explain.
3. Calculate, *by-hand*, the steady-state exit temperature, using that
   - V = 10 m^3
   - ρ_A = 1200 kg/m^3 and ρ_b = 950 kg/m^3
   - $C_{p,A}$ = 2440 J/(kg.K) and $C_{p,B}$ = 3950 J/(kg.K)
   - T_A = 320 K and T_B = 350 K and T_{wall} = 300K
   - $F_A = F_B$ = 0.05 m^3/s
   - k_T = 200 W/(m^2.K) $\times$ 24 m^2 = 4800 W/K

Solution

Step 1: Definition of the problem

What are the inputs and outputs?

In terms of material inputs/output, the process has two inputs as the flow of streams A and B. It also has one output which is a mixture of those two streams.

In terms of energy input/output, we have energy coming in and going out from the material streams as well as energy transfer through the shaft and the wall.

Lumped vs. distributed, steady-state vs. dynamic?

Since we have an impeller inside the tank, we can assume that we have a well-mixed environment which is considered a lumped system. That is, there is NO spatial distribution of quantities through the system. As for change of quantities with time, the flow rate is at steady state as the out-flow equals the sum of the in-flows. For the temperature, let us go with the dynamic formulation for the moment as requested by the problem (we will make the steady-state assumption when we want to solve for the steady-state temperature.)

Step 2: Controlling mechanisms:

This is a mixing tank problem, where there is no reaction nor mass diffusion. However, there is heat transfer from the wall. Note that the work input the by the impeller is safely ignored. Also, radiation is negligible at normal temperatures.

Also, we neglect the contributions from the kinetic and potential energies in the energy balance.

Step 3: Development of a set of model equations

Let us denote by the subscript m the conditions of the mixed fluid that goes out of the tank. The dynamic heat balance is written as follows.

$$\frac{d(\rho_m V C_{p,m} T(t))}{dt} = \rho_A F_A C_{p,A}(T_A - T(t)) + \rho_B F_B C_{p,B}(T_B - T(t)) - k_T(T(t) - T_{wall})$$

where $F = F_A + F_B$.

Remarks and solutions to other questions

I am confused as to the sign of $k_T(T - T_{wall})$ in the heat balance equation. Should I put a negative before it or not?

OK. First you do not need to know whether the heat is lost from or gained into the system to determine the sign of $k_T(T - T_{wall})$. Remember that the heat always goes from the higher temperature to the lower one. So, you cannot even tell whether heat gets out of or comes into the system without knowing the value of T, which is indeed your unknown. But you have to put a negative sign in this case even if you had not been told about the heat loss. Here is why:

If $T > T_{wall} \rightarrow T - T_{wall} > 0$. In this case there is heat loss, which is correctly modelled as $-k_T(T - T_{wall}) < 0$ (negative heat contribution).

If $T < T_{wall} \rightarrow T - T_{wall} < 0$. In this case there is heat gain, which is again correctly modelled using $-k_T(T - T_{wall}) > 0$ (positive heat contribution)

Simplifications:

The liquid volume remains constant as the input and output flows are equal. Also, this is a liquid system where the density can be considered constant. Also, the specific heat capacity can be considered constant if there is no severe variations in T (for gaseous system we would have to consider a variable specific heat capacity). So, the volume V, density ρ_m, and specific heat capacity $C_{p,m}$ can be taken out of the derivative:

$$\rho_m V C_{p,m} \frac{dT(t)}{dt} = \rho_A F_A C_{p,A}(T_A - T(t)) + \rho_B F_B C_{p,B}(T_B - T(t)) - k_T(T(t) - T_{wall})$$

Can the steady state exit stream temperature be higher than both T_A and *T_B? Explain.*

In general, it depends. There is no heat generation or consumption (e.g. via reaction) inside the tank. If there was no heat conduction through the wall (adiabatic tank), the steady-state temperature would be between T_A and T_B, ($T_B > T_A$). However, with the heat conduction, the steady-state temperature

depends on the value of T_{wall}, and other parameters in the process (e.g. k_T). In this case where $T_{wall} < T_A < T_B$, we can say that the steady-state temperature will definitely be less than T_B. But, we cannot decide on whether the steady-state temperature is greater than T_A or not before solving the model for T.

Calculate, by-hand, the steady-state exit temperature

Recall the heat balance equation:

$$\rho_m V C_{p,m} \frac{dT(t)}{dt} = \rho_A F_A C_{p,A}(T_A - T(t)) + \rho_B F_B C_{p,B}(T_B - T(t)) - k_T(T(t) - T_{wall})$$

The term describing the time derivative becomes zero in steady state, so we are left with:

$$\rho_A F_A C_{p,A}(T_A - \overline{T}) + \rho_B F_B C_{p,B}(T_B - \overline{T}) - k_T(\overline{T} - T_{wall}) = 0$$

By substituting the values given in the problem, and solving for $\overline{T}$, the steady-state temperature is obtained as: $\overline{T} = \mathbf{336.3}$ K.

Note: A MATLAB code for solving the above equation - not required for full grade. Please read about the `fsolve` for details. We will cover this in the 3rd part of the course.

```
% Define the parameters and variables
rho_A = 1200; rho_b = 950;
C_pA = 2440;  C_pB = 3950;
T_A = 320;    T_B = 350;   T_w = 300;
F_A = 0.05;   F_B = 0.05;
k_T = 4800;

% Define the function to be solved, in the form: f(x)=0
LHS = @(T) rho_A*C_pA*F_A*(T_A - T) + rho_b*C_pB*F_B*(T_B - T) - k_T*(T - T_w);

% Solve the function for T; Initial guess= 300 K
[T, LHS_val] = fsolve(@(T) LHS(T),300)      %LHS_val should be zero at
                                            %the solution
% Solution
% --------
% T = 336.3292, though when you report your answer, you should
%               use at most 3 significant figures.
% LHS_val = -9.1386e-09
```

Question 2 [2]

Note: You don't need to write any code for this question.

Consider the reaction

$$\mathsf{P_2I_4} + n\,\mathsf{P_4} + p\,\mathsf{H_2O} \longrightarrow 4\,\mathsf{PH_4I} + q\,\mathsf{H_3PO_4}$$

where n, p and q denote the stoichiometric coefficients for $\mathsf{P_4}$, $\mathsf{H_2O}$ and $\mathsf{H_3PO_4}$ respectively.

1. Derive the equations necessary to solve for n, p, and q by equating atoms of P, H, and O on the reactant and product sides.
2. In the next section of the course we will use Gauss Elimination to solve these equations. For now though, let's describe a brute force approach. First, complete the two lines of this MATLAB function:

```
function total_error = equation_error( n, p, q )

% Given the values of n, p, and q, calculate the error of each balance equation.
% Returns the sum of squares of the errors.

error_1 = _________________  % from the P-balance
error_2 = 2*p - 3*q - 16;  % from the H-balance
error_3 = _________________  % from the O-balance

total_error = (error_1)^2 + (error_2)^2 + (error_3)^2;

end  % end of function
```

or complete this Python function:

```
def equation_error(n, p, q ):
    """
    Given the values of n, p, and q, calculate the error of each of the 3 equations.
    Returns the sum of squares of the errors.
    """
    error_1 = _________________
    error_2 = 2*p - 3*q - 16
    error_3 = _________________

    return error_1**2 + error_2**2 + error_3**2
```

3. Since we known that n, p, and q must be positive, we can construct a set of 3 nested for-loops, as shown below in MATLAB and Python. Describe in plain English what the code does.

In MATLAB:

```
smallest = 0.0;
largest = 14.9;
step_size = 0.1;
vector = smallest : step_size : largest;

% How many elements in each vector?
num = length(vector);
errors = zeros(num, num, num);

index_n = 0;
index_p = 0;
index_q = 0;
for n = smallest : step_size : largest
    index_n = index_n + 1;
    for p = smallest : step_size : largest
        index_p = index_p + 1;
        for q = smallest : step_size : largest
            index_q = index_q + 1;

            % Calculate the error at this value of n, p and q:
            errors(index_n, index_p, index_q) = equation_error(n, p, q);

        end
        index_q = 0;
    end
    index_p = 0;
end
[min_error, min_index] = min(errors(:))
[index_n, index_p, index_q] = ind2sub([num, num, num], min_index);
disp(['Solution at ', num2str([vector(index_n), vector(index_p), vector(index_q)])])
```

In Python:

```python
import numpy as np

smallest = 0.0
largest = 15.0
step_size = 0.1
vector = np.arange(smallest, largest, step_size)

# How many elements in each vector?
num = len(vector)

errors = np.zeros( (num, num, num) )

for index_n, n in enumerate(vector):
    for index_p, p in enumerate(vector):
        for index_q, q in enumerate(vector):

            # Calculate the error at this value of n, p and q:
            errors[index_n, index_p, index_q] = equation_error(n, p, q)

# Which combination had the smallest error?
min_index = np.argmin(errors)
index_n, index_p, index_q = np.unravel_index(min_index, (num, num, num))
n, p, q = vector[index_n], vector[index_p], vector[index_q]
print(n, p, q)
```

4. How many times will the function `equation_error` be called?
5. What will this function output be if $(n, p, q) = (1.0, 9.2, 2.5)$?

Solution

1. In the above reaction, component I is already balanced on both sides. Therefore, it remains to write the balance equations for components P, H, and O.

   Balance for P : $2 + 4n = q$, balance for H : $2p = 16 + 3q$, balance for O : $p = 4q$.

   This gives rise to a system of three equations with three unknowns which can be solved easily using tools from linear algebra. It was not required to solve for the unknowns.

2. The equations can be rearranged and set to set to zero. Each of the balance equations `error_1`, `error_2`, and `error_3` must be zero in order to obtain the correct n, p, and q.

   MATLAB code

```
function total_error = equation_error( n, p, q )

% Given the values of n, p, and q, calculate the error of each balance equation.
% Returns the sum of squares of the errors.


error_1 = 4*n - q - 2;     % from the P-balance
error_2 = 2*p - 3*q - 16;  % from the H-balance
error_3 = p - 4*q;         % from the O-balance

total_error = (error_1)^2 + (error_2)^2 + (error_3)^2;


end  % end of function
```

Python code

```
def equation_error(n, p, q ):
    """
    Given the values of n, p, and q, calculate the error of each of the 3 equations.
    Returns the sum of squares of the errors.
    """
    error_1 = 4*n - q - 2      # from the P-balance
    error_2 = 2*p - 3*q - 16   # from the H-balance
    error_3 = p - 4*q          # from the O-balance

    return error_1**2 + error_2**2 + error_3**2
```

3. This code will find the minimum of the sum of the squared errors. The errors are the amount by which the balance equations are incorrect given the values of n, p, and q. If an exact solution is found, then `total_error` would be zero.

   A more detailed explanation: The code tries different values of n, p, and q over a specified range (using the for loops), and evaluates the `total_error` for each set of values through the function `equation_error`. The resulting values of the `total_error` are stored in a 3D array called `errors`. The `index_n`, `index_p`, and `index_q` determine the location of an entry in the 3D array, and correspond to the values of n, p, and q at which the `total_error` has been evaluated.

   At the end of the for loops, we find the minimum value in the resulting array `errors`, which means that we find the minimum `total_error`. Then, using matrix tools we pull out the indices where this minimum `total_error` has been stored, and subsequently, we find the corresponding n, p, and q that have minimized our `total_error`. As mentioned earlier, the `total_error` should ideally be zero. Note that the individual errors error_1, error_2, and error_3 are squared because negative and positive values are both deviation and should not cancel out each other.

4. The equation_error function is called as many times as the most inner loop (`q`-loop) is triggered. The `q`-loop by itself is run from `q=0` to `q=14.9` with step size of 0.1. Therefore, it is run 1+(14.9-0.0)/0.1=150 times. However, the `q`-loop itself is inside an outer loop, the `p`-loop, which is also run 150 times. These two loops themselves are inside the `n`-loop that again is run 150 times. Therefore, the total number of calls to the equation_error function will be `150*150*150` = **3375000** times.

   You can have your code return the total number of calls quite simply as seen in this solution code given in the *bonus question*.

5. To obtain the total_error at $(n, p, q) = (1.0, 9.2, 2.5)$, we can simply call the function `equation_error` with input arguments being the values of n, p, and q. In both MATLAB and Python, the function output is **26.9**.

Bonus question [0.5]

Using the code given in question 2, report what the `min_index` variable is and what are the values of n, p, and q which give minimum error to the set of equations. How long did it take to find the solution of this simple linear equation system?

Solution

MATLAB

The `min_index` is the index of the entry in the array `errors` at which the minimum `total_error` has been stored. Note that `errors` is converted to a column vector through `errors(:)`. Therefore, this `min_index` returns the index in the resulting vector. Its value is `min_index = 739214`. In order to locate the corresponding values of n, p, and q, we need to convert this vector-wise index to an array-wise subscript. Recall that we have a 3D array with each dimension corresponding to one of the unknowns. So, if we convert the `min_index` to a subscript

form of `(index_n, index_p, index_q)`, we can then find the values of n, p, and q that correspond to the `min_error`. This is done through the `ind2sub` command. The answer will be: **n=1.3, p=12.8, q=3.2**.

The time taken to do the 3 nested loops can be easily found from the `tic` and `toc` commands. The solution code appended below shows how it is used.

Note the discrepancy in timing between MATLAB and Python. Recent MATLAB versions have technology built-in to accelerate code by vectorization. Transparent to the user, they will rewrite your MATLAB code to speed it up. If you turn this acceleration off, by typing `feature accel off` before you run the script, the time taken is roughly 94 seconds instead of 3.8 seconds.

Python

The Python code works exactly the same way, except using a different function, `np.unravel_index` to locate the smallest entry in the 3D array. The time taken on my machine was around 54 seconds. It can be reduced by using vectorized calculations.

MATLAB code

```
smallest = 0.0;
largest = 14.9;
step_size = 0.1;
vector = smallest : step_size : largest;

% How many elements in each vector?
num = length(vector);

% Create an empty 3D array to store the errors
errors = zeros(num, num, num);

tic

index_n = 0;
index_p = 0;
index_q = 0;
func_call = 0;    % counts how often equation_error is called

for n = smallest : step_size : largest
    index_n = index_n + 1;
    for p = smallest : step_size : largest
        index_p = index_p + 1;
        for q = smallest : step_size : largest
            index_q = index_q + 1;

            % Calculate the error at this value of n, p and q:
            errors(index_n, index_p, index_q) = equation_error(n, p, q);
            func_call = func_call + 1;  % update the counter

        end
        index_q = 0;
    end
    index_p = 0;
end

toc                          % Elapsed time is 3.32 seconds.

[min_error, min_index] = min(errors(:))  % 739214
[index_n, index_p, index_q] = ind2sub([num, num, num], min_index);
disp(['Solution at ', num2str([vector(index_n), vector(index_p), vector(index_q)])])
% Solution at which the total_error is minimum: n=1.3, p=12.8, q=3.2

%Print how many time equation_error was called:
```

```
fprintf('\n equation_error called %d times\n',func_call) % func_call = 3375000

%Now evaluate equation_error at (1.0,9.2,2.5):
fprintf('\n total_error at (1.0, 9.2, 2.5)=%d\n', equation_error(1.0,9.2,2.5))
%The returned value will be: 26.9000
```

Python code

```
import numpy as np

def equation_error(n, p, q ):
    """
    Given the values of n, p, and q, calculate the error of each of the 3 equations.
    Returns the sum of squares of the errors.
    """
    error_1 = 4*n - q - 2
    error_2 = 2*p - 3*q - 16
    error_3 = p - 4*q

    return error_1**2 + error_2**2 + error_3**2

import time
start_time = time.time()

smallest = 0.0
largest = 15.0
step_size = 0.1
vector = np.arange(smallest, largest, step_size)

func_call = 0 # how often is ''equation_error'' called

# How many elements in each vector?
num = len(vector)

# Create an empty 3D array to store the errors
errors = np.zeros( (num, num, num) )

for index_n, n in enumerate(vector):
    for index_p, p in enumerate(vector):
        for index_q, q in enumerate(vector):

            # Calculate the error at this value of n, p and q:
            errors[index_n, index_p, index_q] = equation_error(n, p, q)
            func_call += 1

# Which combination had the smallest error?
min_index = np.argmin(errors)
print(min_index)
index_n, index_p, index_q = np.unravel_index(min_index, (num, num, num))
n, p, q = vector[index_n], vector[index_p], vector[index_q]
print(n, p, q)    # n=1.3, p=12.8, q=3.2

print('The ''equation_error'' function was called %d times' % func_call) #
# The ''equation_error'' function was called 3375000 times

print(time.time() - start_time)    # varies; on my computer = 53.5 seconds

print('error at (1.0, 9.2, 2.5) = %g' % equation_error(1.0, 9.2, 2.5))
# error at (1.0, 9.2, 2.5) =  26.9000
```