BIEN/CENG 2310
MODELING FOR CHEMICAL AND BIOLOGICAL ENGINEERING

HONG KONG UNIVERSITY OF SCIENCE AND TECHNOLOGY, FALL 2022

HOMEWORK #2 SOLUTION

1. In this problem, you will practice your MATLAB programming skills by computing the Fibonacci series 1, 1, 2, 3, 5, 8, 13, …, defined by the recurrent relation:

$$F_1 = F_2 = 1$$

$$F_i = F_{i-1} + F_{i-2} \quad for \;\; i = 3, 4, 5, 6, \ldots, n$$

You may recall that in the MATLAB tutorial video "Functions and Scripts," an implementation using recursion (that is, a function calling itself) was shown. The code is elegant, but not very efficient. It also only outputs the Fibonacci number F_n but does not store the whole series.

(a) Write a MATLAB function that, given a positive integer n, compute the first n Fibonacci numbers $F_1, F_2, F_3, \ldots, F_n$, and returns them as a column vector. Try to make it as efficient as possible and do not recompute any value that you have obtained before. Your function definition should be:

```
function F = fiboseries(n)
```

Do not use MATLAB's `fibonacci` function or other direct formulae for Fibonacci numbers (e.g., Binet's formula). Do not use recursion as in the MATLAB tutorial video.

(b) Write another MATLAB script (name it `fiboplot_<LastName>_<FirstName>.m`) to plot the ratio F_i/F_{i-1} versus i, for $2 \le i \le 20$. You may only call the function from Part (a) ***<u>once</u>***, and are ***<u>not</u>*** allowed to use `for` or `while` loops in this script.

(c) Rewrite your function in Part (a), but this time, do not use `for` or `while` loops. Instead, use matrix multiplication and exponentiation. The MATLAB function to construct diagonal matrices `diag` will be useful. Your function definition should be:

```
function F = fiboseries_noloop_<LastName>_<FirstName>(n)
```

SOLUTION

PART (a)

The MATLAB program `fiboseries.m` is attached. Note how it handles the exceptions, including when n is 1 (when it should not produce a two-element vector), or when n is not a positive integer (when it should raise an error message or at least return an empty vector).

PART (b)

The plot can be produced in a few lines of code:

```
F = fiboseries(20);
ratio = F(2:end) ./ F(1:end - 1);
plot((2:1:20)', ratio, 'r-*');
xlabel('i');
ylabel('F_i / F_{i-1}');
title('LAM, Henry HW2_Q1(b)');
```

Note that we only need to call `fiboseries` once, and do not need any loop. Rather, we rely on element-wise division between two vectors (both extracted from the Fibonacci number vectors returned by `fiboseries`).

The ratio between adjacent Fibonacci numbers tends to 1.61803... as i increases. This number is the famous golden ratio: $\varphi = \left(1 + \sqrt{5}\right)/2$.

PART (c)

The MATLAB program `fiboseries_noloop.m` is attached. The key idea is to use matrix multiplication to execute the recurrent relation:

$$F_i = F_{i-1} + F_{i-2} \quad for \;\; i = 3, 4, 5, 6, \dots, n$$

To illustrate how it works, for $n = 6$, we define the matrix **A** to be:

$$\mathbf{A} = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 & 0 \end{bmatrix}$$

When multiplied with a column vector $\underline{x}$, it will keep the first two elements of $\underline{x}$ the same, put the sum the 1st and 2nd elements in the 3rd element, the sum of the 2nd and 3rd elements in the 4th element, and so on.

To initiate the Fibonacci series, we put 1's in the first two elements of $\underline{x}$:

$$\underline{x} = \begin{bmatrix}1\\1\\0\\0\\0\\0\end{bmatrix}$$

And we multiple **A** to the left of $\underline{x}$ successively to produce the Fibonacci series in the resulting column vector:

$$\mathbf{A}\underline{x} = \begin{bmatrix}1&0&0&0&0&0\\0&1&0&0&0&0\\1&1&0&0&0&0\\0&1&1&0&0&0\\0&0&1&1&0&0\\0&0&0&1&1&0\end{bmatrix}\begin{bmatrix}1\\1\\0\\0\\0\\0\end{bmatrix} = \begin{bmatrix}1\\1\\2\\0\\0\\0\end{bmatrix} \quad ; \quad \mathbf{A}\left(\mathbf{A}\underline{x}\right) = \begin{bmatrix}1&0&0&0&0&0\\0&1&0&0&0&0\\1&1&0&0&0&0\\0&1&1&0&0&0\\0&0&1&1&0&0\\0&0&0&1&1&0\end{bmatrix}\begin{bmatrix}1\\1\\2\\0\\0\\0\end{bmatrix} = \begin{bmatrix}1\\1\\2\\3\\0\\0\end{bmatrix}$$

$$\mathbf{A}^3\underline{x} = \begin{bmatrix}1&0&0&0&0&0\\0&1&0&0&0&0\\1&1&0&0&0&0\\0&1&1&0&0&0\\0&0&1&1&0&0\\0&0&0&1&1&0\end{bmatrix}\begin{bmatrix}1\\1\\2\\3\\0\\0\end{bmatrix} = \begin{bmatrix}1\\1\\2\\3\\5\\0\end{bmatrix} \quad ; \quad \mathbf{A}^4\underline{x} = \begin{bmatrix}1&0&0&0&0&0\\0&1&0&0&0&0\\1&1&0&0&0&0\\0&1&1&0&0&0\\0&0&1&1&0&0\\0&0&0&1&1&0\end{bmatrix}\begin{bmatrix}1\\1\\2\\3\\5\\0\end{bmatrix} = \begin{bmatrix}1\\1\\2\\3\\5\\8\end{bmatrix}$$

This way, we do not need to write any loop to produce the Fibonacci series. Instead, we use MATLAB's functions for matrix construction, multiplication and exponentiation.

2. Cramer's Rule (http://mathworld.wolfram.com/CramersRule.html) is a century-old method to solve systems of linear equations, though not a very good one. A system of linear equations can be written in matrix form as:

$$\mathbf{A}\,\underline{x} = \underline{b}$$

where $\mathbf{A}$ is the coefficient matrix, $\underline{x}$ is a column vector of the unknowns, and $\underline{b}$ is a column vector of the constant terms:

$$\mathbf{A} = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{bmatrix} \qquad \underline{x} = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} \qquad \underline{b} = \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{bmatrix}$$

Write a MATLAB function that takes in a ***square*** matrix $\mathbf{A}$ (of any size n), and a vector $\underline{b}$, and solves for $\underline{x}$ using Cramer's Rule, keeping in mind that Cramer's Rule only works when the system has a unique solution. Use the following function definition:

```
function x = cramer_<LastName>_<FirstName>(A, b)
```

where `<LastName>` is your last name, and `<FirstName>` is your first name. You are allowed to use the built-in function `det` to calculate the determinant of a matrix. Try to anticipate exceptions and handle them gracefully. That is, catch such exceptions and print out understandable messages rather than causing MATLAB to crash or spit out some cryptic error message.

Hint : To check your answers, you can simply run the command "`x = A \ b`*" which means* $\underline{x} = \mathbf{A}^{-1}\underline{b}$*. However, instead of actually computing* $\mathbf{A}^{-1}$*, the inverse of* $\mathbf{A}$*, MATLAB chooses the most efficient and numerically stable method based on the properties of* $\mathbf{A}$*. If you are curious, you can read: https://www.mathworks.com/help/matlab/ref/mldivide.html.*

Solution

We first have to find out how to apply Cramer's Rule to a linear system. This can be easily learned from many online resources. Given a linear system written in matrix form:

$$\mathbf{A}\,\underline{x} = \underline{b}$$

where

$$\mathbf{A} = \begin{bmatrix} a_{1,1} & a_{1,2} & \cdots & a_{1,n} \\ a_{2,1} & a_{2,2} & \cdots & a_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n,1} & a_{n,2} & \cdots & a_{n,n} \end{bmatrix} \qquad \underline{x} = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} \qquad \underline{b} = \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{bmatrix}$$

The solutions for $\underline{x}$ can be obtained by the following direct formula:

$$x_i = \frac{|\mathbf{A}_i|}{|\mathbf{A}|}$$

for $i = 1, 2, 3 \ldots n$, where $|.|$ denotes the determinant of a matrix, and $\mathbf{A}_i$ is formed by *replacing the i-th column of* **A** by $\underline{b}$**:**

$$\mathbf{A}_i = \begin{bmatrix} a_{1,1} & a_{1,2} & \cdots & a_{1,i-1} & b_1 & a_{1,i+1} & \cdots & a_{1n} \\ a_{2,1} & a_{2,2} & \cdots & a_{2,i-1} & b_2 & a_{2,i+1} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots & \vdots & \vdots & \ddots & \vdots \\ a_{n,1} & a_{n,2} & \cdots & a_{n,i-1} & b_n & a_{n,i+1} & \cdots & a_{nn} \end{bmatrix}$$

We can replace the i-th column of a matrix in MATLAB easily by:

```
A(:, i) = b;
```

And we can use MATLAB's determinant function `det(.)` to calculate the determinant. (It would be quite a hassle to code the determinant function by ourselves.) We just need to write a loop to iterate over $i = 1, 2, 3 \ldots n$ and do the column replacement, apply the formula to obtain x_i one by one, and insert the x_i's back into a column vector $\underline{x}$.

However, we need to check for exceptional (error) cases. The first exception is when **A** is not square, or if the number of rows of **A** is different from the number of rows of $\underline{b}$. This is out of the range of applicability of Cramer's Rule – the column replacement will not work and the determinant is not defined for a non-square matrix. We should check the sizes of **A** and $\underline{b}$ in the user's input, and quit the program with a clear error message if they are incompatible with Cramer's Rule.

The second exception is when **A** is singular, which means $|\mathbf{A}| = 0$. Applying Cramer's Rule to this case will lead to division by zero. So we should calculate $|\mathbf{A}|$ before we do any column replacement, and check if it is zero. (In practice, due to numerical rounding errors, we need to check if $|\mathbf{A}|$ is extremely small, rather than strictly equal to zero.) In linear algebra, a singular matrix means that the system is underspecified: at least one equation is not independent of the rest. This means we have more unknowns than equations, and there will not be a unique solution for this system. Cramer's Rule is also not applicable in this case.

Please see the attached `cramer.m` file for an example of this function.

3. Although we can perform many modeling tasks in Excel, a full-fledged scientific computing programming language like MATLAB can do much more and do so more efficiently. In this problem, we will learn how to write simple MATLAB programs to solve an ODE numerically.

(a) In the last module, we learned how to use Euler method to solve an ODE numerically:

$$\frac{dy}{dt} = f(t, y) \quad ; \quad y(t = 0) = y_0$$

Write a MATLAB function that implements Euler method. The definition should be:

```
[tt, yy] = feuler(fun, y0, tf, h)
```

where `fun` is a user-defined function that should take in two arguments, t and y, in that order, and return $f(t, y)$. The program will simulate the ODE from $t = 0$ to $t = t_f$ with time step h, and return the simulated data points (t_i, y_i) as two vectors `tt` and `yy`, consisting of the t_i's and the corresponding y_i's, respectively.

(b) The Euler method we learned is properly called the "forward Euler" method since it assumes that the slope between (t_i, y_i) and (t_{i+1}, y_{i+1}) can be approximated by the slope evaluated at (t_i, y_i). The formula is:

$$y_{i+1} = y_i + hf(t_i, y_i)$$

Another variant of Euler method is the "backward Euler" method, which uses the slope evaluated at the point (t_{i+1}, y_{i+1}) instead, resulting in the formula:

$$y_{i+1} = y_i + hf(t_{i+1}, y_{i+1})$$

Since we actually do not know y_{i+1} yet, we cannot evaluate $f(t_{i+1}, y_{i+1})$ right away and plug it into the formula. Instead, we need to solve this implicit formula to calculate y_{i+1}. In MATLAB, you can do this with the `fzero` function (see the MATLAB tutorial video on "Root Finding").

Write a MATLAB function that implements "backward Euler" method. The definition should be:

```
[tt, yy] = beuler_<LastName>_<FirstName>(fun, y0, tf, h)
```

(c) In Homework 1, we model the well-stirred tank problem with this ODE:

$$\frac{dC}{dt} = \frac{1}{\tau}(C_{in} - C)$$

with initial condition $C(t = 0) = 0$. Here, we combined V and F into a parameter $\tau = V/F$ called the "residence time" that governs the dynamics of the system.

Write a MATLAB function to solve this ODE by calling the `feuler` function from Part (a), and the `beuler` function from Part (b), and overlay the two numerical solutions in the same plot. For comparison, also overlay the analytical solution:

$$C(t) = C_{in}\left[1 - \exp\left(-\frac{t}{\tau}\right)\right]$$

The function definition should be:

```
[] = stirredtank_<LastName>_<FirstName>(Cin, tau, tf, h)
```

(d) Try a large step size h to better observe the error behaviors of the numerical methods. Which one, forward Euler or backward Euler, seems to be more accurate? Increase the step size further until the forward Euler solution fails to approximate the analytical solution before the concentration C stabilizes. (You should see the forward Euler solution fluctuates wildly, which is clearly unphysical.) Do you see the same behavior for the backward Euler solution? Can you rationalize your observation?

For the written answers to the questions, use the "Text box" function in MATLAB (under the "Insert" menu of the plot) to create a text box in some empty part of the plot, and include your answers there. This way, you don't need to submit a separate file with the written answers.

SOLUTION

PART (a)

Please see the attached `feuler.m` for the program. Note that since h may not divide t_f exactly, we may simulate *past* t_f. That is, $t_n \geq t_f$ where (t_n, y_n) is our last data point.

PART (b)

Please see the attached `beuler.m` for the program. The only difference is in how we calculate y_{i+1}. We use `fzero` to find the root of the following equation (y_{i+1} being the variable):

$$y_i + hf(t_{i+1}, y_{i+1}) - y_{i+1} = 0$$

Note that `fzero` requires an initial guess (it is an iterative method). For simplicity, we use y_i as the guess, since we expect the true value of y_{i+1} not to be too far from y_i (provided our step size is small and the function is continuous).

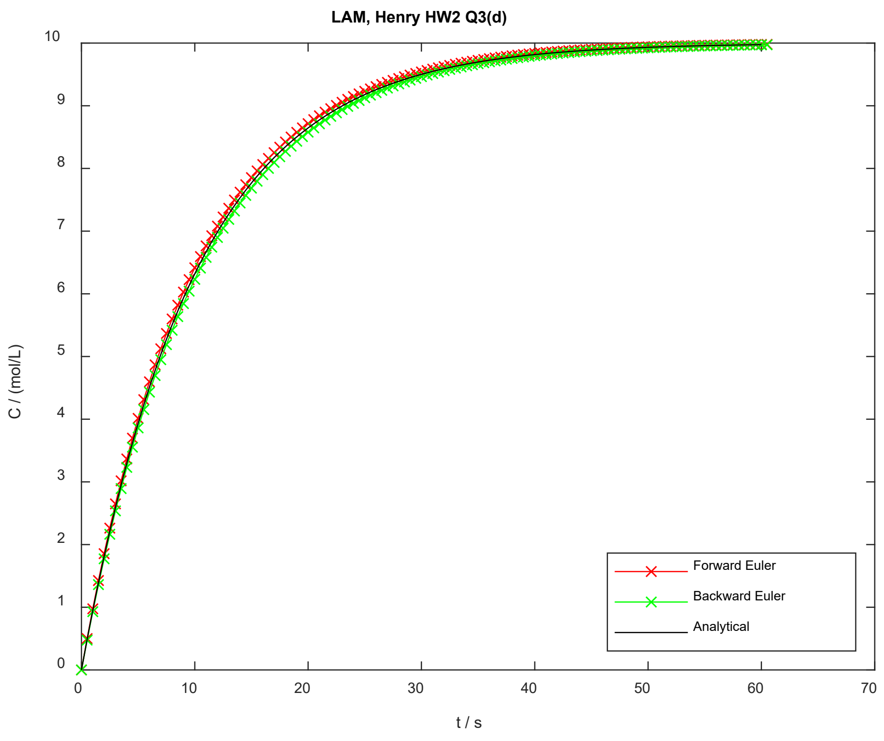
PART (c)

Please see the attached `stirredtank.m` for the program to compare the three solutions to the stirred tank problem of HW1, Q1.

PART (d)

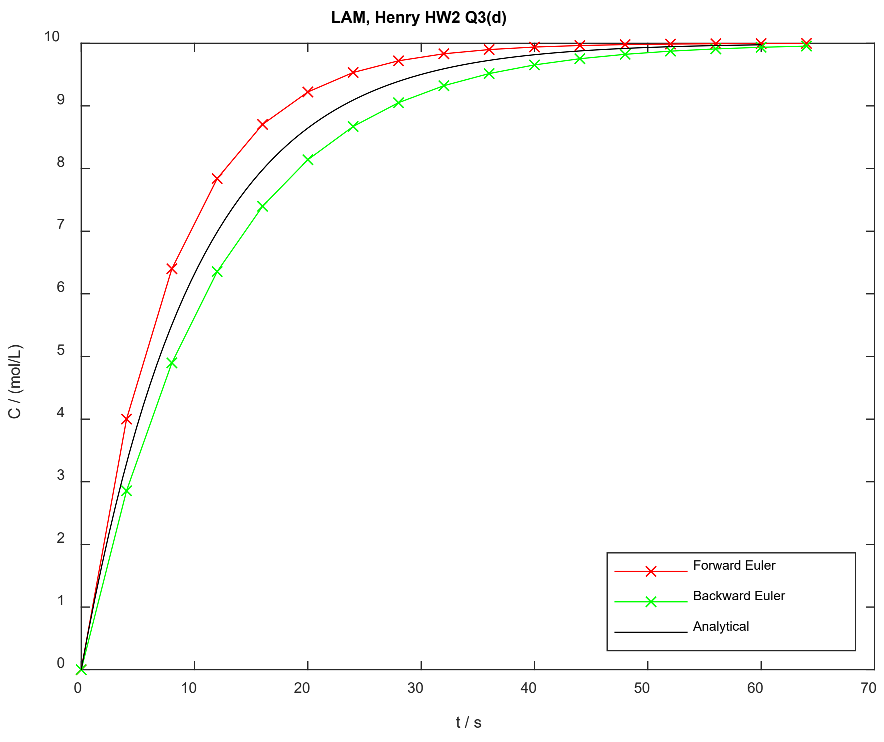
Typical plots by `stirredtank.m` are shown below. The parameters are $C_{in} = 10$ mol/L and $\tau = 10$ s and $t_f = 60$ s. The other difference is the step size h.

For $h = 0.5$ s:



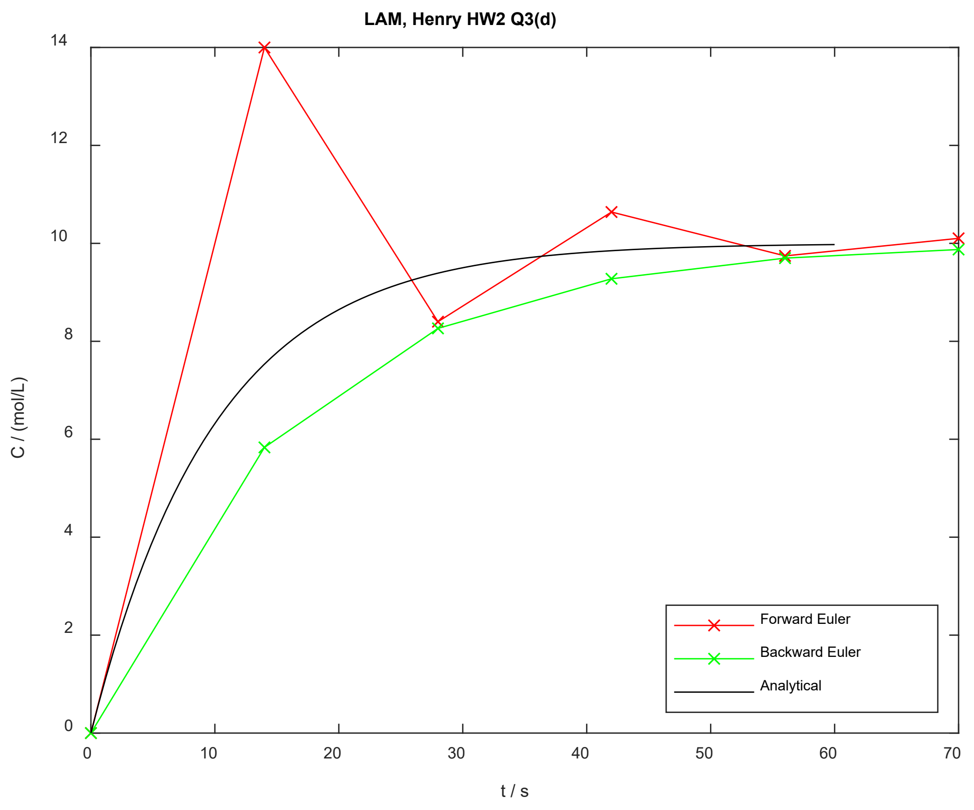
One can see that both numerical methods provided good approximations of the analytical solution at such small step sizes.

For $h = 4$ s:



At larger step sizes, the error becomes more apparent. But both methods seem to deviate from the analytical solution by about the same amount, but in different directions.

For $h = 14$ s:



At even larger step sizes, the forward Euler solution begins to exhibit some crazy fluctuations that are totally unphysical, but the backward Euler solution does not (though it is quite off from the analytical solution). In other words, backward Euler still give us the right trend and a somewhat acceptable approximation, but forward Euler can completely mislead us. In the language of numerical methods, backward Euler is considered more "stable." This is because backward Euler chooses the next data point with more constraints – the location and the computed slope at that location must be such that it is connected with the last data point by a line segment with exactly that slope. Forward Euler, on the other hand, only projects forward with no such mechanism to "check answer." A big error in one data point may potentially produce a horribly wrong slope and an even bigger error for the next data point.