Matthew Lacaire
UID: 205288439
CEE/MAE M20
October 31, 2020 (🎃)

Homework 4

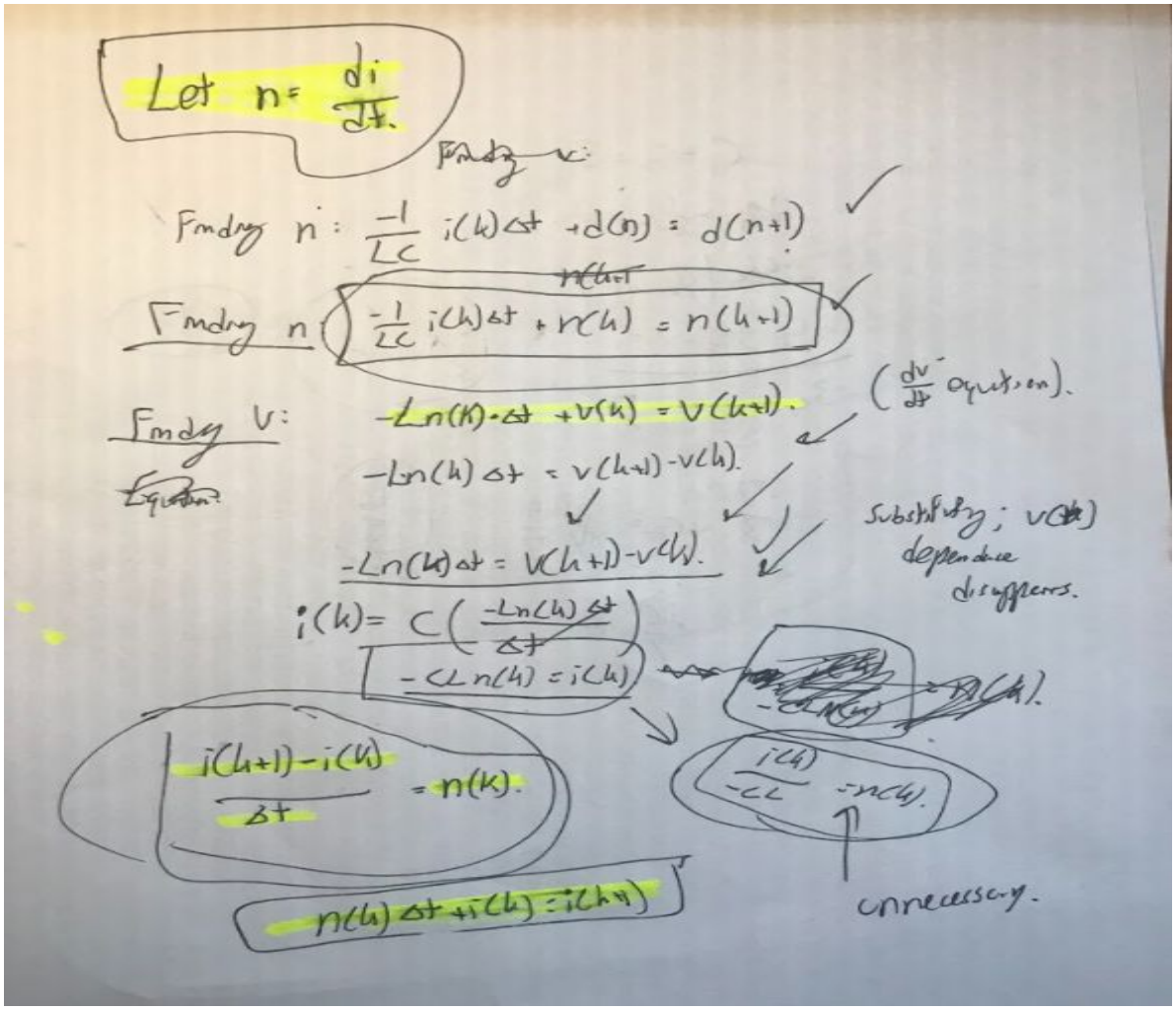
1. LC Circuit Problem: Explicit Euler

a. Introduction

Given an L-C circuit, I use and rearrange the differential equations to set up an explicit Euler method calculation for current in the circuit from 0s-20s with step size 0.01s.

b. Model and Methods

In the process of solving this problem, I first had to deal with the differential equations as given, and had to sub them into one another so as to remove any v(t) dependence (v wasn't given as an initial condition). I took di/dt to be a variable called "n", and the following is an example of my scratchwork:



Everything highlighted is what I ended up using in my final code. See calculations for more info.

c. Calculations and Results

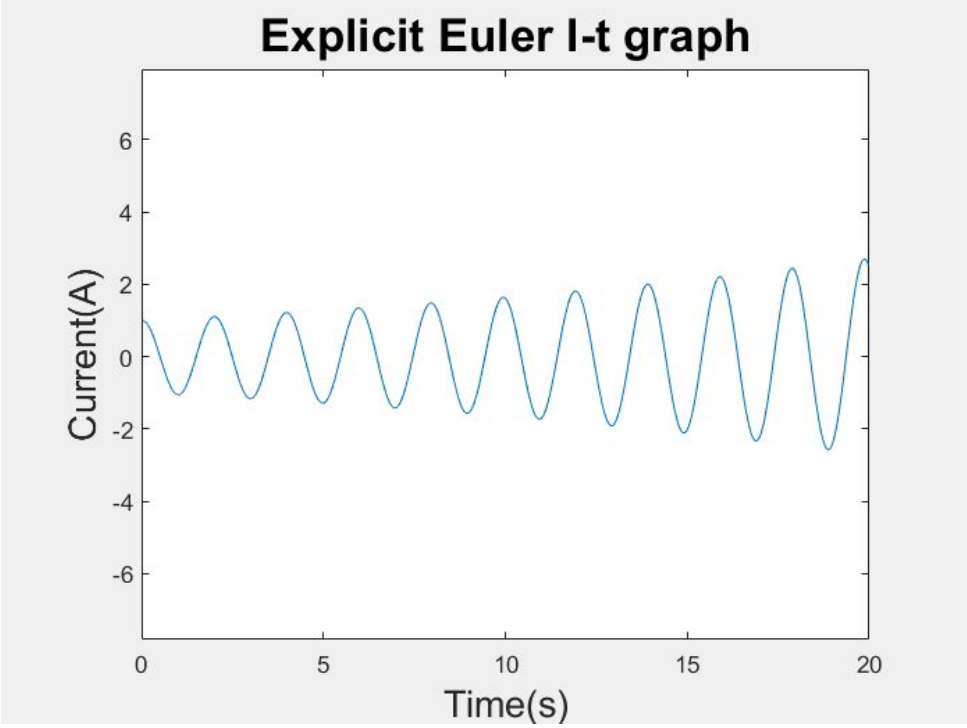
More concretely, I began the actual calculation process by defining several arrays: "times", which has length 2001 and goes from 0-20 (therefore step size is 0.01), "i" (which is zeroes, and also has length 2001), and "n" with also zeros. I determined a dt as 0.01 (step size). I input set values for L and C, and set i(1) = 1 A, n(1) = 0 as specified in the problem description. Then, using a for loop iterated 2,000 times (of the 2001 entries, we still don't know the last 2000; first one is initial conditions), I rearranged equations shown above and inputted them into MATLAB. This is how they looked:

```
i(k+1) = n(k)*dt+i(k);
n(k+1) = (-1/(L*C)) * i(k)*dt + n(k);
```

What we're looking for, of course, is the i, which is calculated in the first line. In the second, we find the di/dt for the next step so it can be used again in the first step in the next iteration.

Then, the arrays i and n having been filled up to 2001, I plot. I use the plot() function, and label the axes (xlabel, ylabel commands). I also use the title() command to create a title. I also play with the font, and make sure the axes are equal in scaling with "axis equal". See code for more explanation.

Then, visually inspecting the graph, which is oscillating and increasing in amplitude, I find that the peaks are around two integer values from one another, making period ~2s. Thus, frequency is 1/T, making it ~0.5s. I print this as well at the end. See graph below:



d. Discussion

As discussed in class, this entire graph is based on approximations. Furthermore, each approximation is based on the calculation before it- error will propagate and become more noticeable if present. That is seen here, where energy appears to be added to the system, increasing the amplitude. In reality, this is just a product of using a naturally imperfect (and frankly, somewhat rough) approximation method.

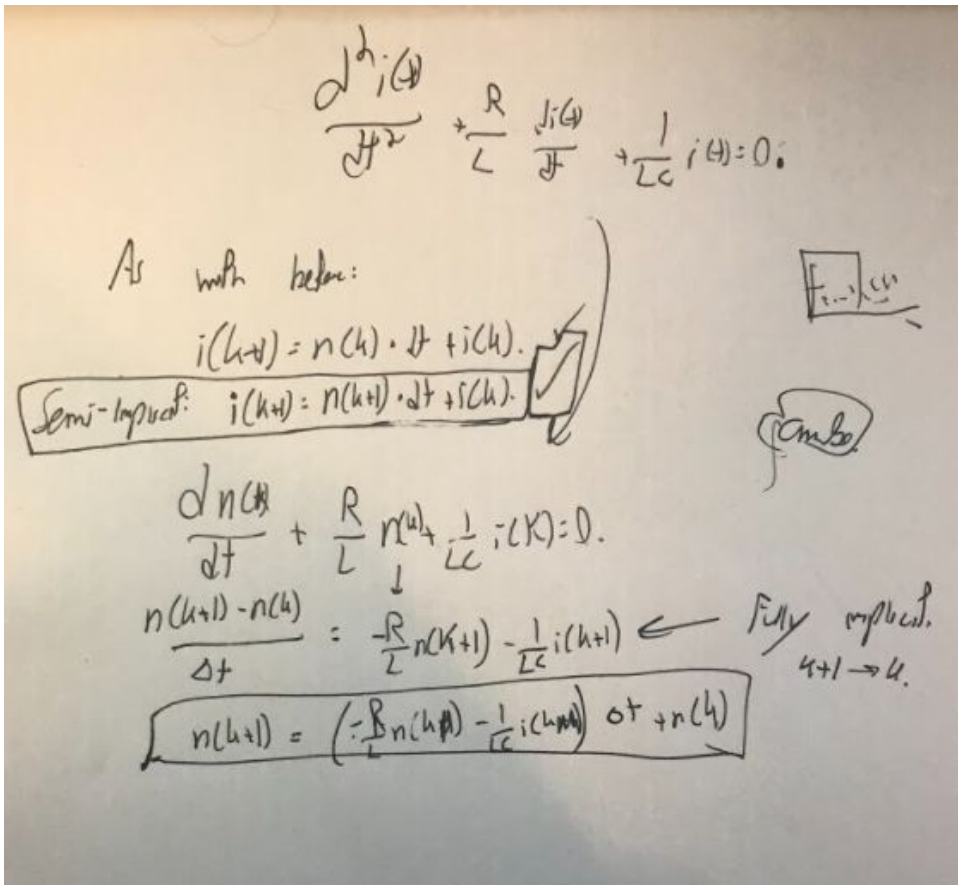
2. RLC Circuit Problem: Semi-Implicit Euler

a. Introduction

In contrast to the first problem, I now use a semi-implicit method to calculate current over time in an RLC circuit, using a new differential equation. Additionally, I graph three different curves based on three different resistance values. I also calculate the damping coefficient of each curve/resistor value.

Model and Methods

First, I find it necessary to manipulate the given equations to find something that can be put into MATLAB. A snapshot of my thinking is below. Note that in the boxed answer, I have all things in terms of k and not k+1 on the right side. The two boxed answers shown were inputted into MATLAB. Again n represents di/dt.



b. Calculations and Results

Much like in problem 1, I define several arrays of length 2001, and initialize the i and n values. This time, there being three resistor values, I also define:

```
Rvals = [0.2, 2, 20];
```

Also, the fact that three different resistors are present means that I need to create three i arrays, and three n arrays. Each array will fill with different values depending on R, considering the semi-explicit approximation (which I will explain shortly) depends on R:

```
iR1 = zeros(length(times),1);
nR1 = zeros(length(times),1);
iR2 = zeros(length(times),1);
nR2 = zeros(length(times),1);
iR3 = zeros(length(times),1);
nR3 = zeros(length(times),1);
```

"R1" refers to the first resistor value, namely 0.2 ohms. The others follow this convention. These are also initialized to i(1) = 0A, n(1) = 1 A/s.

Then, I have three separate for loops, one for each resistor value. The following is the for loop for R=0.2 ohms:

```
for k = 1:length(times)-1
        nR1(k+1) = (((-Rvals(1) / L) * nR1(k)) -
(1/(L*C))*iR1(k))*dt + nR1(k);
        iR1(k+1) = nR1(k+1)*dt + iR1(k);
end
```

Basically, this runs 2000 times, and uses the equations that I hand wrote above to find the next i and n values. This approximation is semi-implicit because the finding of the n value at k+1 depends on values at k, but finding i at k+1 depends on this new k+1 n value (di/dt).

I get these values and plot, much like in the first problem. Notably, I use the "hold on" command to allow multiple plots to be graphed together. I also create a legend:

```
legend({'R = 0.2 ohms','R = 2 ohms','R = 20 ohms'}, 'box','off')
```

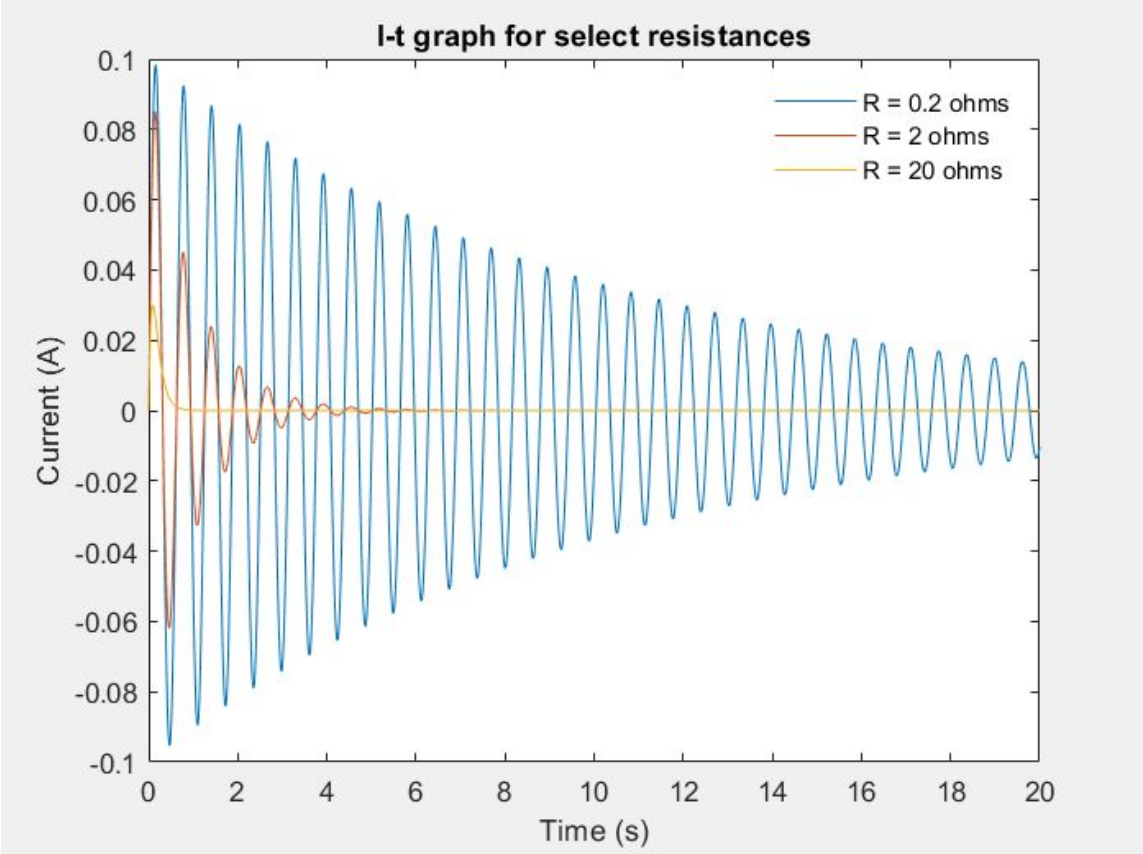
This legend automatically assigns corresponding colors to the curves I graphed, namely the times-iR1, times-iR2, and times-iR3 graphs. I also turned off the border box.

Lastly, I calculate and output damping coefficients. Consider sample code for R1's damping coefficient:

```
cof1 = (Rvals(1)/2)*sqrt(C/L);
```

Then, I print this along with cof2, and cof3 (corresponding to R=2, R=20)

See graph and console output below:




```
The 0.2ohm circuit has damping coefficient 0.01000, 2ohm is
0.10000, and 20 ohm is 1.00000
```

c. Discussion

I can confirm that this graph makes physical sense by comparing with what's expected- damping is indeed present, and subsequently decreases the amplitude over time of the current. For a higher resistance R, the system has much higher damping (it's directly proportional to the damping constant), which brings current closer to 0 more rapidly. For R = 20 ohms, the system doesn't even change current direction (current less than 0), but rather travels there without any further oscillation. I believe this is called critical damping.

3. RLC Circuit: Implicit Euler

a. Introduction

In the same type of RLC circuit with the same constants as number 2, I use a fully implicit method to calculate and plot i-t graphs for each resistance value. I do this with Newton's method.

b. Model and Methods

In an identical vein to the last two problems, I define a times(), constants, and three i zeros arrays/three "n" arrays. In keeping with the requirements of the question, *note that instead of "n" = di/dt I now use a*.

There isn't too much here to say about the structure of the program that wouldn't be redundant- like question two, it's composed of a defining section, three for loops, then a displaying section. For exact specifics on calculation, see next section.

c. Calculations and Results

The following is an entire for loop for R = 0.2 ohms, which I will annotate and describe. Note that the next two for loops are identical, but rather provide values for the i arrays corresponding to their particular R value.

```
%% R = 0.2 ohms
for k = 1:length(times)-1
    x = [iR1(k);aR1(k)];
     f1 = (x(2) - aR1(k))/dt + (Rvals(1)/L)*x(2)+(1/(L*C))*x(1);
    f2 = x(2) - (x(1) - iR1(k))/dt;
    y = [f1; f2];
    while norm(y) > 1e-6
        J = [1/(L*C), (1/dt + (Rvals(1)/L)); -1/dt, 1];
    x = x - J\y;
    f1 = (x(2) - aR1(k))/dt + (Rvals(1)/L)*x(2)+(1/(L*C))*x(1);
    f2 = x(2) - (x(1) - iR1(k))/dt;
    y = [f1; f2];
    end
    iR1(k+1)=x(1);
    aR1(k+1)=x(2);
end
```

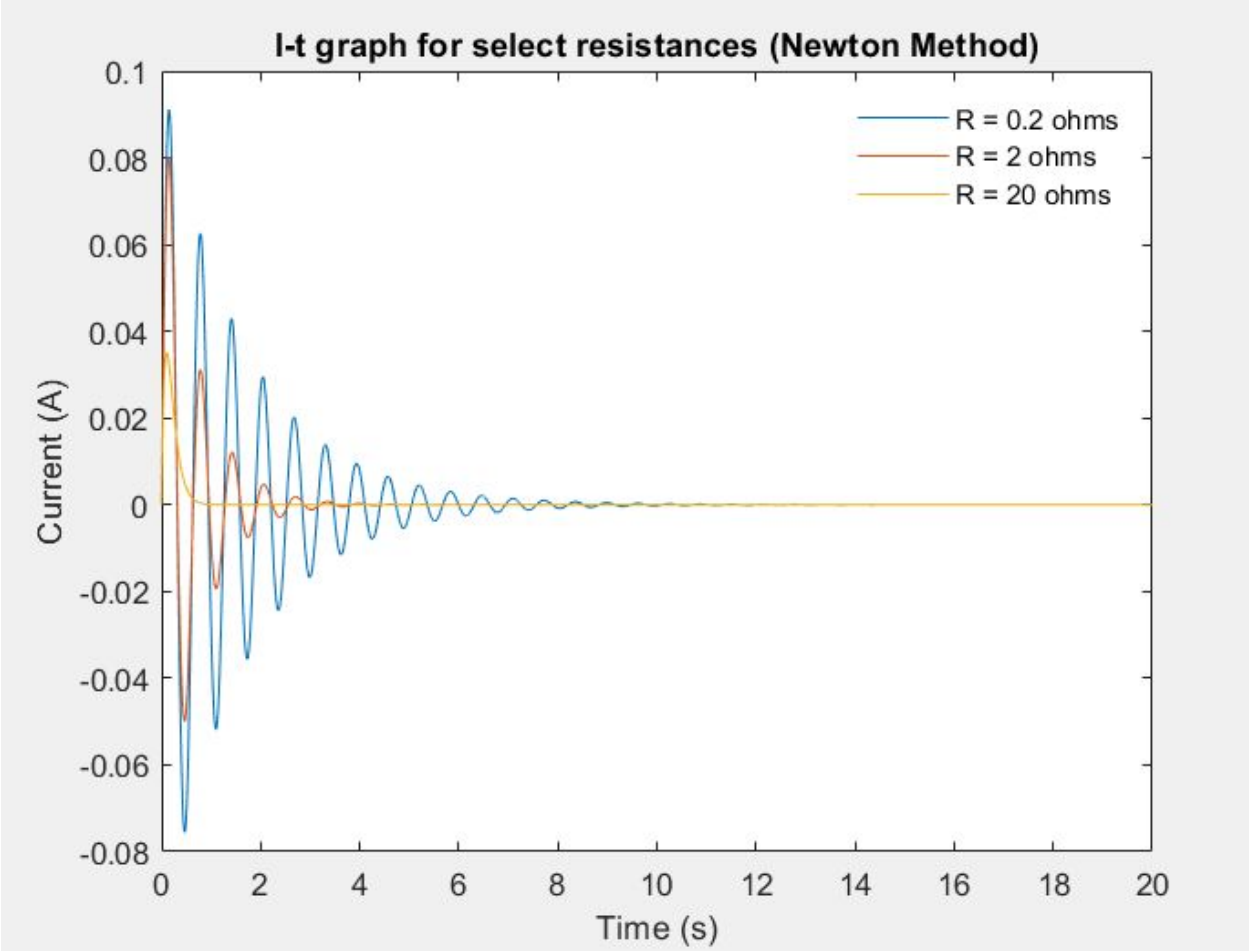
This is the bulk of the Newton method. I get the two equations used in question 2, and equate them to zero in keeping with the Newton method. Then, I define an x column vector, containing the current iR1 and aR1, which are the i(k) and a(k) (di(k)/dt), respectively. Then, I create column vector y, which stores outputs of f1, f2.

In a while loop, I check to see if the *norm* of y is bigger than 1e-6. Norm is essentially square root of (f1(k+1)^2) + (f2(k+1)^2), and, per linear algebra, this is just the magnitude of the 2D vector y. I define a Jacobian matrix and calculate the required derivatives. I differentiate with respect to the required quantities, and fill the jacobian. I treat the constants with arguments (k) as constants, and only differentiate with respect to (k+1) values. Results are in the J seen above. In the J\y line, I calculate some x such that J(x) = y. That's what the backslash shorthand does. Then, I perform matrix subtraction, and redefine

x. I then redefine the f1, f2 values with new x values. New f1, f2 are stored in a new y matrix. This continues until the value of norm(y) is sufficiently small.

Then, I set the k+1 i and a values to x(1) and x(2). Then, the loop shifts to the next k and repeats.

After all of this, I graph, using code almost identical to that in question 2. I get this:



d. Discussion

The graph in this problem, though using the same constants as the second problem, is remarkably more damped. This probably comes about due to the fact that semi-implicit estimates are still based on rough approximations and estimates. This method, the Newton method, works fundamentally differently, and actually checks to ensure that the error of each step doesn't cross a certain error threshold. No such check exists for the implicit euler method, meaning that error can simply build upon error.