University of the Philippines Diliman
College of Engineering
INSTITUTE OF CIVIL ENGINEERING

Machine Problem:

LEAKING TANK and COUPLED SPRING-MASS SYSTEM

Litana, Gerome R.

Mosquera, Jericho Just P.

In Partial Fulfillment

Of the Requirements in

CE 27: Analytical and Computational Methods in Civil Engineering II

Ms. Noriza Tibon

December 4, 2018

ABSTRACT

Leaking tank problems are common applications of ordinary differential equations in Civil Engineering. This machine problem aims to determine the time required to drain a water-filled tank using analytical, Euler's, midpoint, and 4^{th} order Runge-Kutta methods wherein three different step sizes (0.1, 0.01, 0.001) were given. The time vs. height graph was plotted for each of the methods and step sizes and was compared with that of the analytical method. On the other hand, Springs, having multiple important roles in our community, should be studied and understood in order for it to be truly appreciated. This other part of the machine problem tackles the computation of the displacement over time of the two spring-masses along with its natural frequencies and mode shapes. Through the use of the 4th Order Runge-Kutta Method for Ordinary Differential Equations and the Shifted-Inverse Power Method, the group was able to accurately determine the characteristics of the coupled spring-mass system. The resulting graphed displacement-time values of the system resulted in a sinusoidal wave.

Part 1: Leaking Tank

I. Introduction

Leaking tank problems are one of the most common applications of ordinary differential equations in Civil Engineering. This part of the machine problem aims to determine the required time to drain a water-filled tank using four numerical methods- analytical, Euler's, midpoint, and 4th order Runge-Kutta wherein three different step sizes (0.1, 0.01, 0.001) were given. In addition, the top radius of the tank is 1 m, its bottom radius is 0.5 m, its height is 2 m, the side of the square hole is 0.03 m and the initial height of the water in the tank is 2 m. As a comparison to the analytical method, the time vs. height graph was plotted for each methods and step sizes which will be shown later on.

To use the four methods stated above, a differential equation must be set-up first. This leaking tank problem is specifically an outflow of water through a hole, which means that the volumetric flow rate, dV/dt, is important. Equation 1 shows the general formula of a volumetric flow rate.

$$\frac{dV}{dt} = \sum_{i=0}^{n} Q_{in} - \sum_{i=0}^{n} Q_{out} \tag{1}$$

Where Q_{in} is the volumetric flow rate of the entering fluid and Q_{out} is the volumetric flow rate of the exiting fluid. Since no fluid is entering the tank in this MP, Q_{in} = 0.

The given tank resembles a *truncated cone*, with the volume given by the equation:

$$V_{truncated\ cone} = \frac{\pi h}{3}(R^2 + Rr + r^2) \quad [1] \tag{2}$$

Where h is the height of the truncated cone, R is the bigger radius, and r is the smaller radius.

We will now use equation 1 to obtain the differential equation of the problem. Our goal is to write dV/dt with respect to dh/dt. We also take note that the big radius (R) depends on the height (h), so we also have to write R in terms of h by using similar triangles which is shown in figure 1.

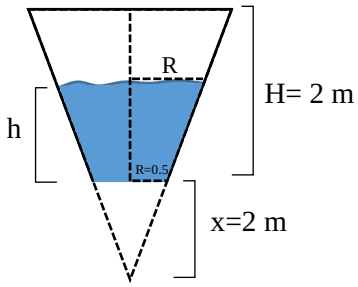


Figure 1: Relationship of big radius (R) with height (h)

By using the similar triangles in figure 1, we obtain the relationship of R in terms of h as:

$$R = \frac{h+2}{4} \tag{3}$$

Plugging in big radius R (Equation 3) in equation 2, and given small radius (r) is equal to 0.5 m, we get the volume of the truncated cone with respect to h and is simplified to:

$$V_{truncated\ cone} = \frac{\pi}{3}\left(\frac{h^3}{16} + 0.375h^2 + 0.75h\right) \tag{4}$$

The next step that we have to do to obtain *dV/dt* is to apply *partial derivatives*. We must know that $\left(\frac{dV}{dt}\right) = \left(\frac{\partial V}{\partial h}\right) * \left(\frac{dh}{dt}\right)$. Applying partial derivatives to equation 4, we have:

$$\left(\frac{dV}{dt}\right) = \frac{\pi}{3}\left[\frac{3h^2}{16}\left(\frac{dh}{dt}\right) + 0.75h\left(\frac{dh}{dt}\right) + 0.75\left(\frac{dh}{dt}\right)\right] \tag{5}$$

For the right side of equation 1, we express $Q_{out} = -vA$ as:

$$Q_{out} = -A_{hole} C_\alpha \sqrt{2gh} \tag{6}$$

Where the area of the hole, A_{hole} is given by $(0.03\ m)^2$, the coefficient of discharge C_α is equal to 1 for this MP, the acceleration due to gravity g is 9.81 m/s^2, and the variable h is the height.

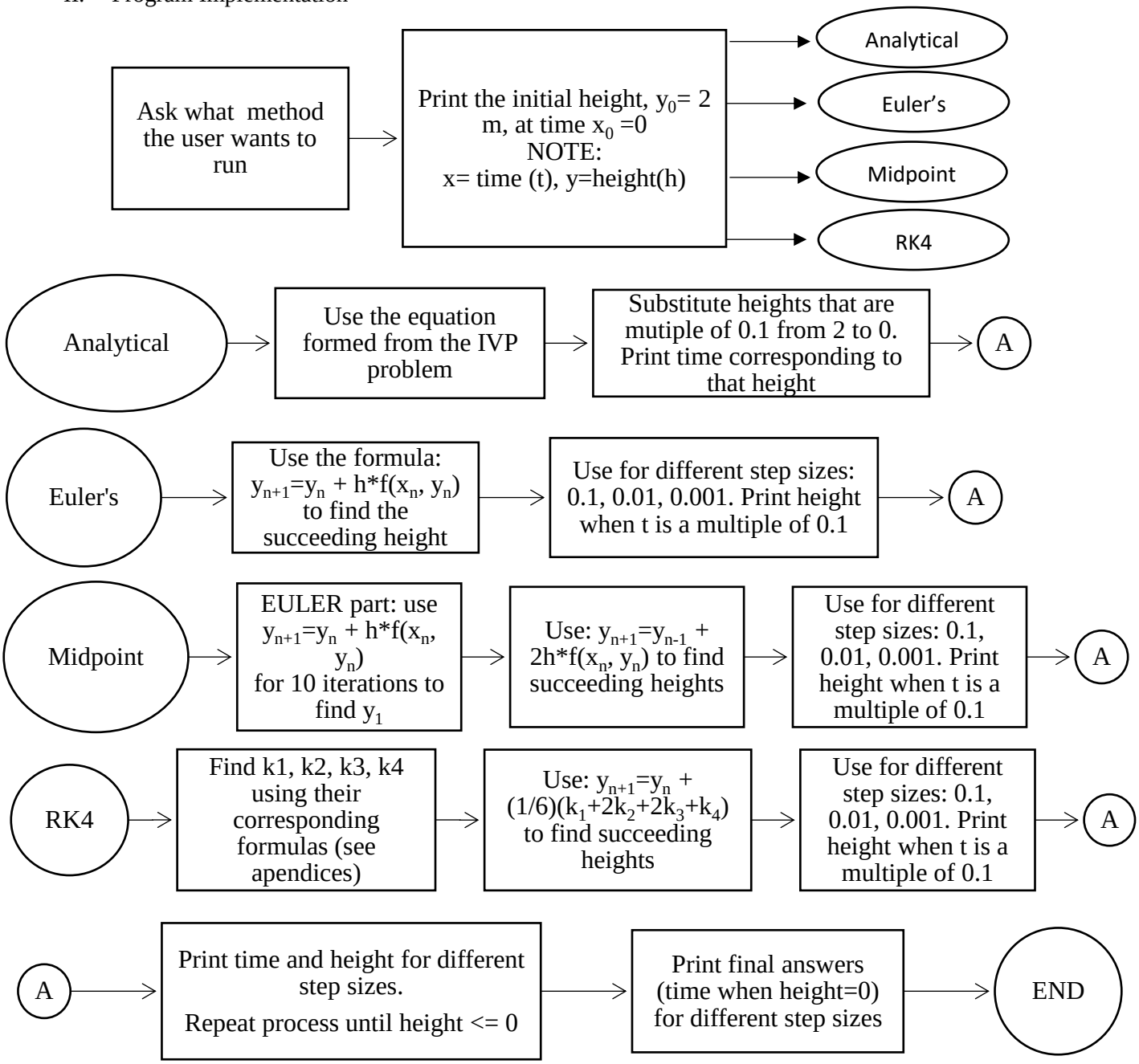
Substituting equations 5 and 6 to equation 1, we will find the simplified differential equation for this leaking tank problem given by:

$$\frac{dh}{dt} = \frac{-(0.009)\sqrt{19.62}}{(\pi/3)(0.1875h^{\frac{3}{2}} + 0.75h^{\frac{1}{2}} + 0.75h^{-\frac{1}{2}})} \quad (7)$$

Equation 7 will be used in Euler's, midpoint, and 4th order Runge-Kutta methods while the concept of initial-value problem will be used for the analytical method.

The expected outputs of the program are a table containing the time (in multiples of 0.1) and the height at any time t corresponding to the three different step sizes, and the final answer – the time it takes for the height to be 0. Further results and screenshots of the outputs of the different methods are shown in Results and Discussion.

II. Program Implementation



III. Results, Discussions and Conclusions

Before creating the program for the problem, the group first brainstormed on how will the program be user-friendly and more efficient. The first thought that they had in mind is that an introductory text showing the title of the problem, the mathematical model/ differential equation to be used, the initial values, and the other notes would give a good impression to the user. This is seen in lines 13 to 16. In addition, they thought that it would be more appealing to the user if it had organized texts in the screen by adding divisions and simple designs to it. An output is shown below.

```
==================================PROBLEM 1: LEAKING TANK==================================

After manually solving, the mathematical model was known to be:

 dh/dt= (-0.00380682921)/(0.1875h^1.5 + 0.75h^.5 + 0.75h^-0.5)   given h(0)=2

Please check the Executive report for the full solution of the mathematical model.

[NOTE]: Please open the file 'prob1data' to check the summary of calculated data for ALL methods.
```

Second, they thought that it would be good if the program will ask what method the user wants to run, so that it would be hassle free to the user. The group used a *do-while* loop and a *switch* structure to demonstrate this. The program will end only if the user enters 0, while the switch part gives the user an option from 1 to 5. The whole do-while loop extends from line 18 up to the very last line 407 while the switch structure is in between it. An output of this is seen below.

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

1 - Analytical Method   3 - Midpoint Method               5 - SUMMARY of all methods
2 - Euler's Method      4 - 4th Order Runge-Kutta Method
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

What method do you want to run? Pick a number:
```

Then, they thought that all data computed by the program would be more organized and less of a clutter if it is also printed into a CSV file. This is the part where the group encountered a problem. The data to be stored are composed of more than 10 000 rows but arrays can only store for up to 10 000 elements only. This CSV File handling part will be tackled and seen as we go through the different algorithms and screenshots taken later on.

The first method used to find the time it takes for the given water-filled tank to empty is by using analytical method which involves the concept of initial value problem. For the initial value problem, we simply use the concept of *separable ordinary differential equations* wherein we separate *dh* and *dt* in both sides of the equation 7, also making sure that all *h* terms are in the *dh* side. Then, we integrate the left and right side of equation 7 with respect to *h* and *t*, respectively. After integrating, we obtain the equation:

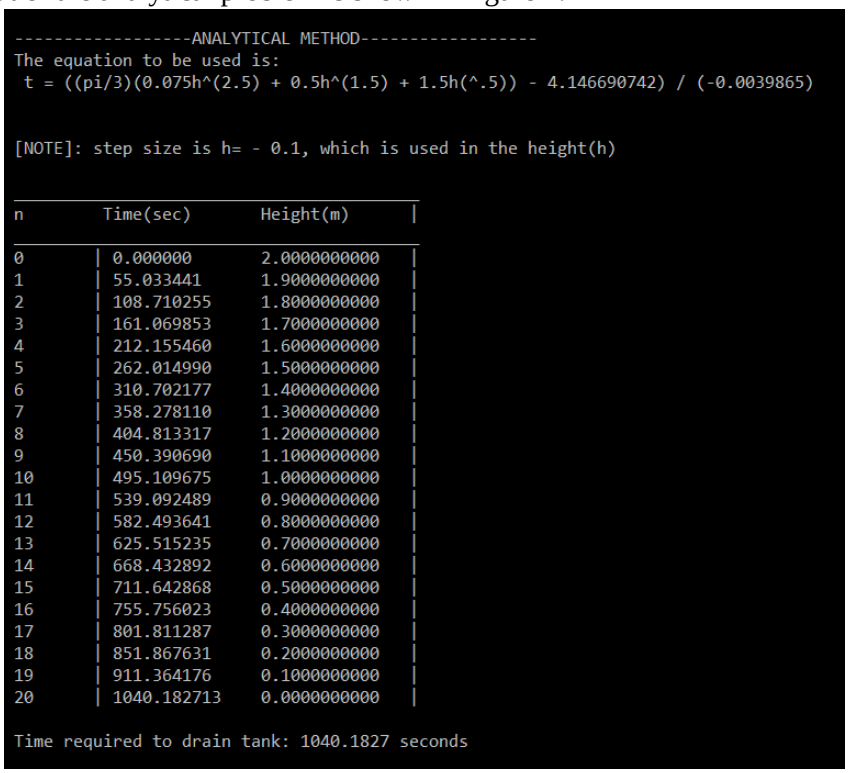
$$\frac{\pi}{3}\left(0.075h^{\frac{5}{2}} + 0.5h^{\frac{1}{2}} + 0.75h^{-\frac{1}{2}}\right) = -(0.009)\sqrt{19.62}t + C \qquad (8)$$

Solving the initial value problem, given h=2 when t=0, we obtain C=4.146690742 and the final equation of time (t) with respect to height (h) is given by:

$$t = \frac{\left(\frac{\pi}{3}\right)\left(0.075h^{\frac{5}{2}}+0.5h^{\frac{1}{2}}+0.75h^{-\frac{1}{2}}\right)-4.146690742}{-(0.009)\sqrt{19.62}} \qquad (9)$$

To solve the required time, we simply substitute h=0, we then find that by analytical method, the time it takes to empy the given tank is t = 1040.182723 seconds.

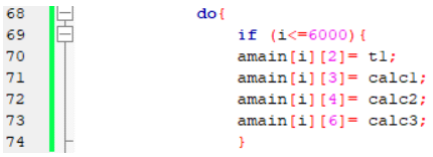
In the program, analytical method was demonstrated by showing the different time that corresponds to heights that are multiples of 0.1, starting from 2. The code for this is seen in lines 33 to 52. The program output of the analytical problem is shown in figure 2.



```
------------------ANALYTICAL METHOD------------------
The equation to be used is:
 t = ((pi/3)(0.075h^(2.5) + 0.5h^(1.5) + 1.5h(^.5)) - 4.146690742) / (-0.0039865)


[NOTE]: step size is h= - 0.1, which is used in the height(h)

_____________________________________________
n        Time(sec)       Height(m)      |
_____________________________________________
0       | 0.000000       2.0000000000   |
1       | 55.033441      1.9000000000   |
2       | 108.710255     1.8000000000   |
3       | 161.069853     1.7000000000   |
4       | 212.155460     1.6000000000   |
5       | 262.014990     1.5000000000   |
6       | 310.702177     1.4000000000   |
7       | 358.278110     1.3000000000   |
8       | 404.813317     1.2000000000   |
9       | 450.390690     1.1000000000   |
10      | 495.109675     1.0000000000   |
11      | 539.092489     0.9000000000   |
12      | 582.493641     0.8000000000   |
13      | 625.515235     0.7000000000   |
14      | 668.432892     0.6000000000   |
15      | 711.642868     0.5000000000   |
16      | 755.756023     0.4000000000   |
17      | 801.811287     0.3000000000   |
18      | 851.867631     0.2000000000   |
19      | 911.364176     0.1000000000   |
20      | 1040.182713    0.0000000000   |

Time required to drain tank: 1040.1827 seconds
```

Figure 2: Output of the Analytical Method

When the user inputs 2 in the prompt screen, the second method - Euler's method will run. Lines 54 to 66 of the code show the variable initializations and declarations, and the initial information for Euler's method. Then, equation 7 will now be used. Before heading to the main calculations we first create a code for storing data in a double array *amain*. This can be seen in lines 68 to 74.



```
68      do{
69          if (i<=6000){
70          amain[i][2]= t1;
71          amain[i][3]= calc1;
72          amain[i][4]= calc2;
73          amain[i][6]= calc3;
74          }
```

For the calculation part, the group divided it into three parts, each for the different step sizes, 0.1, 0.01, 0.001. The code for h=0.1 is seen in lines 76 to 86. An *if-else* statement was used such that if *h1* (height for h=0.1) is positive, the program will print the # of iterations, the time, and the corresponding height at that time. Else, if h1 is not positive, the column for h=0.1 will not print the corresponding height.

The code for the step sizes 0.01 and 0.001 are similar to that of the 0.1 except, the program will only print the height, calculated from Equation 10, corresponding to a time multiple of 0.1 until a given iteration.

$$y_{n+1} = y_n + h * f(x_n, y_n) \tag{10}$$

For 0.01, there are 10 iterations while for 0.001, there are 100. The whole program for the Euler's method is encapsulated in a *do while* statement wherein the stated codes for the three step sizes will just iterate while either of the three heights for the three step sizes are still greater than or equal to zero. When the reiteration stops, the program will print the final answer, for all step sizes used. Final output is seen in figure 3 (note: previous iterations weren't shown).

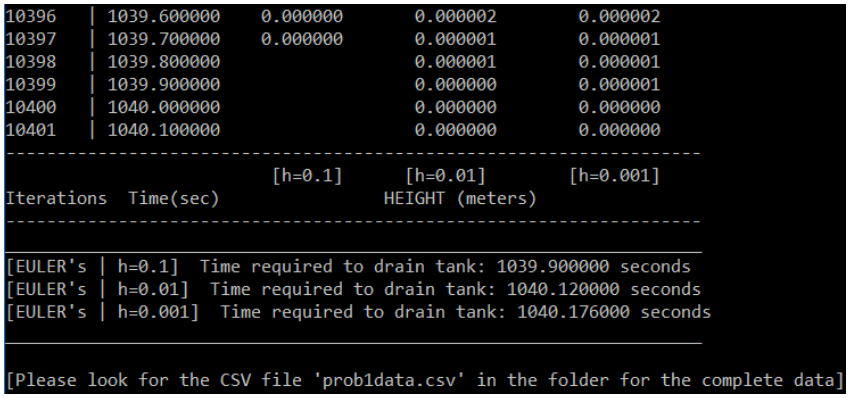


Figure 3: Output of Euler's Method

The third method used is the Midpoint method. Lines 126 to 132 shows the variable declarations, title, and other necessary information. For the Euler part, a *for loop* of 1000 iterations was used. Inside it is an *if* statement wherein, similar to the previously stated Euler method, the height corresponding to the three step sizes were printed. The group used 10 iterations for the three step sizes and is shown in figure 4.

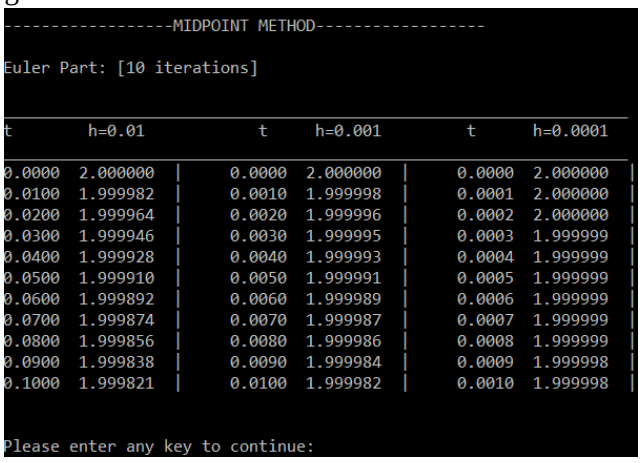


Figure 4: Output of the Euler's Part of Midpoint method

After printing the Euler part, we can see in Figure 4 that a prompt is shown, asking to enter any key to continue. The group used the syntax *getchar()* to code this prompt.

For the main midpoint method, a *for loop* is also used and similar to Euler's method, various height corresponding to different time were printed. The height is calculated from the formula:

$$y_{n+1} = y_{n-1} + 2h * f(x_n, y_n) \tag{11}$$

An output of the Midpoint method is shown in Figure 5.

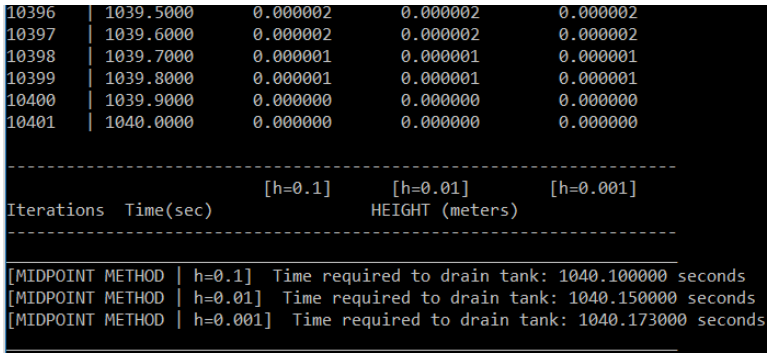


Figure 5: Output for Midpoint Method

Selecting number 4 in the main menu runs the 4th order Runge-Kutta Method. What made this method different than the previous methods was that it had *k1, k2, k3,* and *k4* that will be used to compute the next height using the formula:

$$y_{n+1} = y_n + h * \frac{1}{6}(k_1 + 2k_2 + 2k_3 + k_4) \quad (12)$$

Similar to the previous methods, the group used *for loops, if-else statements,* and *a do while loop* to code the program. All of these, including the variable declarations are shown in lines 235 to 313. An output of the 4th order Rk method is shown in figure 6.

```
1039.9  | 0.0000   0.0000   0.0000      | -0.00 -0.00 -0.00 -0.00  | -0.00 -0.00 -0.00 -0.00  | -0.00 -0.00 -0.00 -0.00
1040.0  | 0.0000   0.0000   0.0000      | -0.00 -0.00 -0.00 -0.00  | -0.00 -0.00 -0.00 -0.00  | -0.00 -0.00 -0.00 -0.00
1040.1  | 0.0000   0.0000   0.0000      | -0.00 -0.00 -0.00 -0.00  | -0.00 -0.00 -0.00 -0.00  | -0.00 -0.00 -0.00 -0.00
1040.2  | 0.0000   0.0000   0.0000      | -0.00 -0.00 -0.00 -0.00  | -0.00 -0.00 -0.00 -0.00  | -0.00 -0.00 -0.00 -0.00

-------------------------------------------------------------------------------------------------------------------------
        [h=0.1]  [h=0.01] [h=0.001]       k1    k2    k3    k4       k1    k2    k3    k4       k1    k2    k3    k4
time               HEIGHT                    k for [h=0.1]            k for [h=0.01]           k for [h=0.001]
-------------------------------------------------------------------------------------------------------------------------

[RK 4 | h=0.1]  Time required to drain tank: 1040.100000 seconds
[RK 4 | h=0.01]  Time required to drain tank: 1040.170000 seconds
[RK 4 | h=0.001]  Time required to drain tank: 1040.182000 seconds
```

Figure 6: Output for 4th Order Runge-Kutta Method

Finally, data from all of the methods are then printed into a CSV file named "prob1data.csv". The group used the concept of *File Handling* to successfully create the code for this. A *pointer* named *aPtr* and other syntaxes such as *fprintf, fopen, and fclose* were used to further maximize the potential of the program. These can be seen in lines 342 to 360.

The whole program doesn't end right there because there is still an option 5, the summary of all methods. The group decided to include this so that the user will not have hard time running all the methods all over again and to also provide a neat and user-friendly look. However, when selecting this option, a warning will be shown, saying that the user must run all methods at least once before continuing. This is shown in Figure 7.

```
---------------- SUMMARY OF ALL METHODS----------------

ANALYTICAL METHOD: 1040.182713

METHOD:          h=0.1          h=0.01          h=0.001
Euler's        | 1039.900       1040.120        1040.176
Midpoint       | 1040.100       1040.150        1040.173
RK4            | 1040.100       1040.170        1040.182
```

Figure 7: Summary of all time required for all methods

To further compare the different methods with the analytical method, a graph of time vs. height for the three methods were created. In addition, the graph for the analytical method is seen in figure 8.

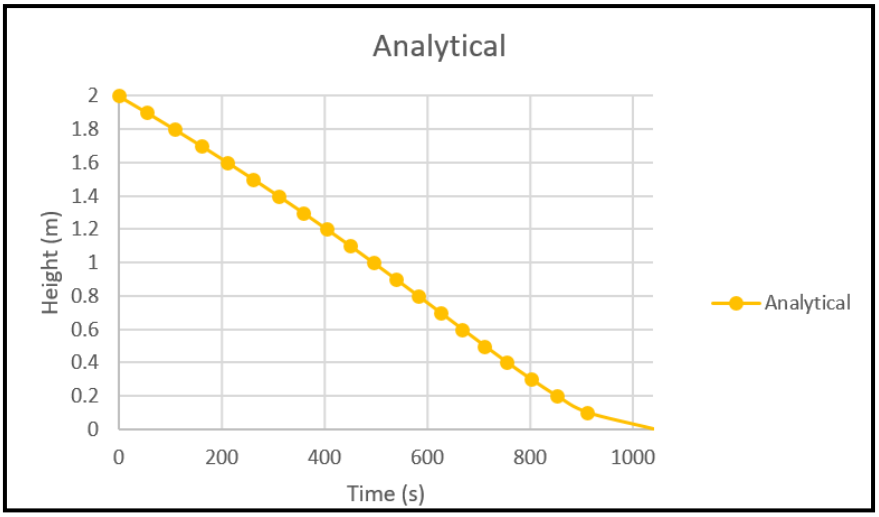


Figure 8: Time vs. height graph of the analytical method

After graphing, the group then realized that the graphs for all methods are not easy to compare from each other since the curves overlap. To fix this, the group decided to take a small portion of the graph from a small interval (from 1039.6 to 1040.2 seconds). Also, since the curves of step sizes 0.01 and 0.001 still coincide with the analytical method, Figures 10 and 11 only shows a part of the said curves to make them more visible.

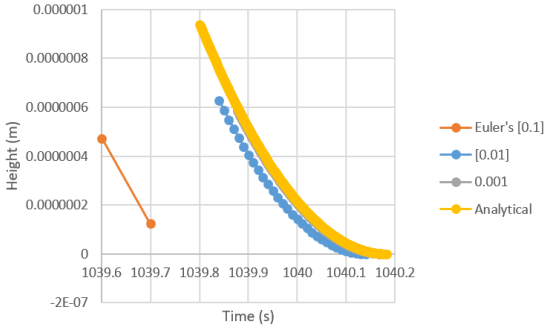


Figure 9: t vs. height graph for Euler's Method

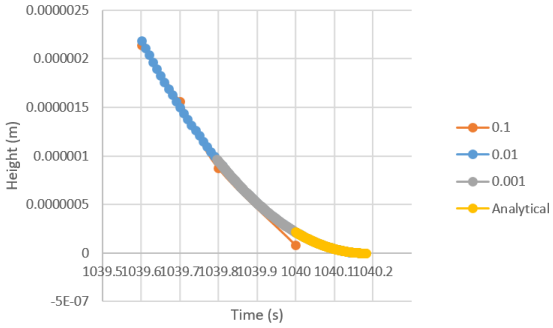


Figure 10: t vs. height graph for Midpoint Method

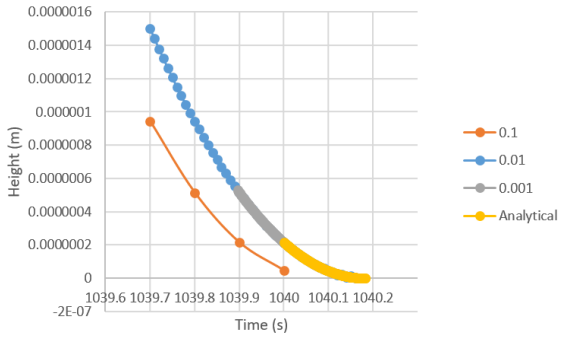


Figure 11: t vs. height graph for 4th Order Runge-Kutta Method

Almost all graphs show that different methods produce a different curve. The nearer the curve to that of the analytical method, the more accurate it is. In Figure 9, 10, and 11, the curve for the 0.1 step size is the farthest of all to the curve of the analytical method. Thus, we can conclude that as the step size gets smaller, the accuracy of getting the analytical answer gets higher.

IV. Conclusions and Recommendations

Different methods in obtaining the time required to empty a water-filled tank were programmed in this machine problem. Based from the results, it can now be verified that each method has a different process in obtaining time, and thus, different time required was obtained in each step sizes. By comparing the time versus height graphs of each methods and step sizes, we can conclude that as the step sizes get smaller, the accuracy becomes higher and the answer gets closer and closer to that of the analytical method. In addition, Euler's method was least accurate in obtaining the answer while the 4^{th} order Runge-Kutta method is the most accurate one. Indeed, it is more efficient to create a program that can calculate these many iterations so that tedious tasks and human errors will be prevented.

V. References

[1] Merz, Steven (May 9, 2017), *"How do you find the volume of a truncated cone shape?" May* 9, 2017, Retreived from https://www.quora.com/How-do-you-find-the-volume-of-a-truncated-cone-shape

VI. Appendices

List of Main Variables:

- . *int*
  - i, j, k – used in for loops, dummy variables
- *double*
  - *amain[10000][10000]- main array used to store all computed data*
  - *bmain[10000][10000]- backup array used to store all computed data*
  - *t, time, time1, time2, time3 – time variables for each methods and step sizes*

Part 2: Coupled Spring-Mass System

I. Introduction

Springs play a major role on most of our daily life. We feel it in a few of the chairs that we sit on and even in the ball pens that we use. On a major scale, large dampers are included in the construction of bridges so as to counteract the vibrations of the bridge. This is why it is important to theoretically determine the characteristics of a spring-mass system because it provides an avenue for mathematicians and engineers to compute for a system's most optimal specifications for real world objects, creating a safer and more convenient environment for communities.

For Problem 2, an undamped spring-mass system with an external force is presented to have two masses that have an acceleration and displacement with respect to the time. The problem is finding the changes in displacement over a certain amount of time. The problem is illustrated in diagram (1) below.

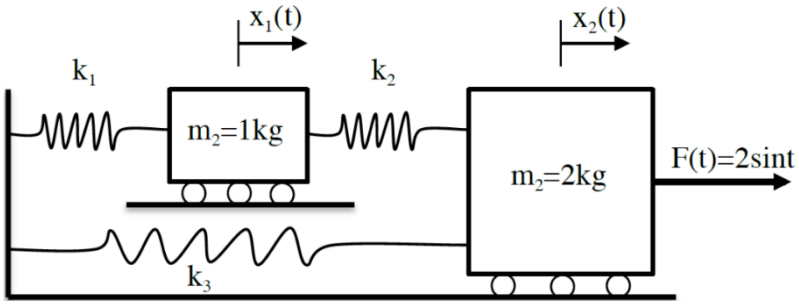


Figure 1. Spring-Mass System Diagram

For the first part of the problem, we are to make use of the given 2nd Order ordinary differential equation of the given diagram which is:

$$\begin{bmatrix} m_1 & 0 \\ 0 & m_2 \end{bmatrix} \begin{Bmatrix} x_1'' \\ x_2'' \end{Bmatrix} + \begin{bmatrix} k_1 + k_2 & -k_2 \\ -k_2 & k_2 + k_3 \end{bmatrix} \begin{Bmatrix} x_1 \\ x_2 \end{Bmatrix} = \begin{Bmatrix} 0 \\ 2 \end{Bmatrix} \sin t$$

Figure 2. 2nd Order ODE System of Equations

The method that will be used to solve for this equation is called 4th Order Runge-Kutta Method. This Method was named after famous Mathematicians, Karl Runge and Wilhem Kutta, that developed ways to numerically solve for differential equations. This could also be considered as the *improved* Euler Method. Unfortunately, this method is used mainly for 1st Order differential equations and we have to first substitute it with values that transform it into the 1st Order differential equation that we need it to be. On top of that, since it is considered as a system of two masses, the equations are to be solved in tandem with each other. This will be further illustrated in the Program Implementation part of the report.

The second part of the problem is to determine the natural frequencies and mode shapes of the system. This part could be considered through the concept of eigenvalues and eigenvectors having natural frequencies considered as the eigenvalue and its mode shapes as the eigenvectors of the given eigenvalue.

$$\boldsymbol{A} = \begin{bmatrix} m_1 & 0 \\ 0 & m_2 \end{bmatrix}^{-1} \begin{bmatrix} k_1 + k_2 & -k_2 \\ -k_2 & k_2 + k_3 \end{bmatrix}$$

Figure 3. A-matrix of Problem 2 part 2.

The objectives in solving problem is to utilize the methods suggested in order to determine the displacement of the two bodies at a certain time. In this case, only up to 5 seconds. The group must also successfully identify the natural frequencies and mode shapes of the given matrix through the program that will be presented. X_1" X_2" Y_1' Y_2'.

II. Program Implementation

Part 1: 4th Order Runge-Kutta Method

A. Convert 2nd Order DE to 1st Order DE

⇩

B. Solve for $f_{1\text{-}4}$ and $g_{1\text{-}4}$

⇩

C. Iterate through the use of the new values of x & y and progressing time (t + step size)

⇩

D. Repeat steps B & C until desired time is reached.

A. After isolating the $\begin{pmatrix} X1'' \\ X2'' \end{pmatrix}$ matrix to the left and the rest of the variables to the right, consider:

$$\begin{pmatrix} X1' \\ X2' \end{pmatrix} = \begin{pmatrix} Y1 \\ Y2 \end{pmatrix} = \begin{pmatrix} f1 \\ f2 \end{pmatrix}$$ - derive once

$$\begin{pmatrix} X1'' \\ X2'' \end{pmatrix} = \begin{pmatrix} Y1' \\ Y2' \end{pmatrix} = \begin{pmatrix} g1 \\ g2 \end{pmatrix}$$ - to be considered for RK4

B. With the initial values of x(0) = $\begin{pmatrix} 0 \\ 0 \end{pmatrix}$ and x'(0) = y(0) = $\begin{pmatrix} 0 \\ 0 \end{pmatrix}$ and @ t = 0, utilize f and g formulas of RK4.

$$x_{n+1} = x_n + \frac{1}{6}h \cdot (f_1 + 2f_2 + 2f_3 + f_4)$$

$$y_{n+1} = y_n + \frac{1}{6}h \cdot (g_1 + 2g_2 + 2g_3 + g_4)$$

Where

$$f_1 = f(t_n, x_n, y_n) \qquad g_1 = g(t_n, x_n, y_n)$$

$$f_2 = f\left(t_n + \tfrac{h}{2}, x_n + \tfrac{h}{2}f_1, y_n + \tfrac{h}{2}g_1\right) \qquad g_2 = g\left(t_n + \tfrac{h}{2}, x_n + \tfrac{h}{2}f_1, y_n + \tfrac{h}{2}g_1\right)$$

$$f_3 = f\left(t_n + \tfrac{h}{2}, x_n + \tfrac{h}{2}f_2, y_n + \tfrac{h}{2}g_2\right) \qquad g_3 = g\left(t_n + \tfrac{h}{2}, x_n + \tfrac{h}{2}f_2, y_n + \tfrac{h}{2}g_2\right)$$

$$f_4 = f(t_n + h, x_n + hf_3, y_n + hg_3) \qquad g_4 = g(t_n + h, x_n + hf_3, y_n + hg_3)$$

Part 2: Shifted-Inversed Power Method

A. Input given A-matrix, initial alpha value, initial column matrix

B. Subtract alpha-identity matrix from A-matrix.

A. The A-matrix inputted is the result of the given matrix from the problem:

$$\boldsymbol{A} = \begin{bmatrix} m_1 & 0 \\ 0 & m_2 \end{bmatrix}^{-1} \begin{bmatrix} k_1 + k_2 & -k_2 \\ -k_2 & k_2 + k_3 \end{bmatrix}$$

$$B.\ \begin{pmatrix} 3 & -2 \\ -1 & 1.5 \end{pmatrix} - \alpha \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$$

C. Inverse the (A-α*I*) matrix	C. $\begin{bmatrix} 3-\alpha & -2 \\ -1 & 1.5-\alpha \end{bmatrix}^{-1} = \begin{bmatrix} 1.5-\alpha & 2 \\ 1 & 3-\alpha \end{bmatrix}$
D. Multiply column matrix from (A-α*I*)$^{-1}$	D. $\begin{bmatrix} 1.5-\alpha & 2 \\ 1 & 3-\alpha \end{bmatrix} * \begin{pmatrix} X1 \\ X2 \end{pmatrix}$
E. Factor out the largest number from the resulting matrix: μ-value	
F. Multiply new column matrix to (A-α*I*)$^{-1}$	
G. Repeat D-F until tolerance is reached.	
H. After iteration, $\omega = \sqrt[2]{\frac{1}{\mu}+\alpha}$	$\phi = Final \begin{pmatrix} x1 \\ x2 \end{pmatrix}$

III. Results, Discussions and Conclusions

The main issue our group encountered in solving Problem 2 was comprehending the values and diagrams provided. We allotted 1-2 weeks just to correctly understand the problem before properly undertaking in creating an algorithm for the two parts.

The first part required a deep understanding of the 4th Order Runge-Kutta Method since the given equations was not only a 2nd Order DE but it was also systematically a 2x1 matrix which required the f and g formulas to be in matrix form. Once we got over the difficulty of how to view the equations, Microsoft Excel was used to compute for the values one by one.

$$\begin{pmatrix} f_{1n} \\ f_{2n} \end{pmatrix} = \begin{pmatrix} y_{1n} \\ y_{2n} \end{pmatrix} \qquad \begin{pmatrix} g_{1n} \\ g_{2n} \end{pmatrix} = \begin{pmatrix} y'_{1n} \\ y'_{2n} \end{pmatrix} = \begin{pmatrix} x''_{1n} \\ x''_{2n} \end{pmatrix} = \begin{bmatrix} -3 & 2 \\ 1 & -1.5 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} + \begin{bmatrix} 0 \\ \sin t \end{bmatrix}$$

We considered how each row of the matrix could be separated into two table of values to better see the differences in values.

ROW 1										
t	f1	f2	f3	f4	x1	g1	g2	g3	g4	y1
0.00000000					0.00000000					0.00000000
0.25000000	0.00000000	0.00000000	0.00000000	0.00000000	0.00000000	0.00000000	0.00000000	0.00000000	0.00779217	0.00032467
0.50000000	0.00032467	0.00097402	0.00192268	0.00536222	0.00047835	0.00519478	0.01278406	0.02015017	0.04146444	0.00501333
0.75000000	0.00501333	0.00988800	0.01335974	0.02476150	0.00365628	0.03899741	0.06677133	0.07899269	0.12163778	0.02385346
1.00000000	0.02385346	0.03887418	0.04556424	0.07045232	0.01462222	0.12016576	0.17368621	0.18639545	0.25023492	0.06929363
1.25000000	0.06929363	0.10059201	0.10998812	0.15271247	0.04142082	0.25038701	0.32555592	0.33367537	0.40951618	0.15172554
1.50000000	0.15172554	0.20316506	0.21353914	0.27512850	0.09393175	0.41151621	0.49450882	0.49361186	0.56454925	0.27473832
1.75000000	0.27473832	0.34575623	0.35448847	0.43112735	0.18169655	0.56814326	0.63800120	0.62555613	0.67052701	0.43164603
2.00000000	0.43164603	0.51602434	0.52019072	0.60278273	0.31114900	0.67502652	0.70835758	0.68454682	0.68329588	0.60431816
2.25000000	0.60431816	0.69028413	0.68737225	0.76240271	0.48290040	0.68772775	0.66443277	0.63233820	0.57063686	0.76481426
2.50000000	0.76481426	0.83655994	0.82514400	0.87674737	0.68977413	0.57396538	0.48263786	0.44773241	0.32159922	0.87966031
2.75000000	0.87966031	0.92003139	0.90024248	0.91308695	0.91616142	0.32296860	0.16465731	0.13370654	-0.04841793	0.91596358
3.00000000	0.91596358	0.90977886	0.88345141	0.84584381	1.13900592	-0.04947772	-0.26009732	-0.28047906	-0.49819542	0.84809583
3.25000000	0.84809583	0.78538980	0.75583786	0.66237732	1.33037794	-0.50164822	-0.73806375	-0.74287403	-0.96691910	0.66349404
3.50000000	0.66349404	0.54196699	0.51342883	0.36661611	1.46124884	-0.97221642	-1.20052173	-1.18751175	-1.38487475	0.36627912
3.75000000	0.36627912	0.19239582	0.16926190	-0.02031348	1.50580222	-1.39106644	-1.57613775	-1.54637040	-1.68635163	-0.02215565
4.00000000	-0.02215565	-0.23369171	-0.24770181	-0.46267113	1.44548498	-1.69228852	-1.80436930	-1.76206196	-1.82197288	-0.46578581
4.25000000	-0.46578581	-0.69410518	-0.69663886	-0.91541308	1.27203969	-1.82655495	-1.84682439	-1.79850908	-1.76778839	-0.91932790
4.50000000	-0.91932790	-1.14060310	-1.13109388	-1.33115739	0.98896139	-1.77020155	-1.69412779	-1.64731796	-1.52919399	-1.33525653
4.75000000	-1.33525653	-1.52639170	-1.50611421	-1.66738828	0.61114236	-1.52908136	-1.36686140	-1.32852699	-1.13891105	-1.67103858
5.00000000	-1.67103858	-1.81309354	-1.78484939	-1.89245889	0.16283472	-1.13643965	-0.91048649	-0.88568123	-0.64959843	-1.89513748

Figure 4. Table of Data Computed for Row 1 from System of Equations

After computing for the values of the displacements with Microsoft Excel, the formulating of the algorithm and the programming itself was less challenging since the general idea of the Runge-Kutta method in C was similar to the one we devised in Excel. The group focused on the usage of arrays and the do-while function, simplifying the process of the iteration.

```
42  do
43  {
44
45  printf("%d\t\t%f\t%f\t%f\t%f\t%f\t%f\t%f\t%f\t%f\t%f\t%f\t%f\n",n, t,f1[0],f2[0],
46  printf("\t\t\t\t%f\t%f\t%f\t%f\t%f\t%f\t%f\t%f\t%f\t%f\t%f\n\n",f1[1],f2[1], f3[1]
47
48  f1[0] = y[0];
49  g1[0] = gfxn1(x[0],x[1]);
50  f1[1] = y[1];
51  g1[1] = gfxn2(x[0],x[1],t);
52  f2[0] = (y[0] + (h*0.5*g1[0]));
53  g2[0] = gfxn1(x[0]+(h*0.5*f1[0]),x[1]+(h*0.5*f1[1]));
54  f2[1] = (y[1] + (h*0.5*g1[1]));
55  g2[1] = gfxn2(x[0]+(h*0.5*f1[0]),x[1]+(h*0.5*f1[1]),t+(h*0.5));
56  f3[0] = (y[0] + (h*0.5*g2[0]));
57  g3[0] = gfxn1(x[0]+(h*0.5*f2[0]),x[1]+(h*0.5*f2[1]));
58  f3[1] = (y[1] + (h*0.5*g2[1]));
59  g3[1] = gfxn2(x[0]+(h*0.5*f2[0]),x[1]+(h*0.5*f2[1]),t+(h*0.5));
60  f4[0] = (y[0] + (h*g3[0]));
61  g4[0] = gfxn1(x[0]+(h*f3[0]),x[1]+(h*f3[1]));
62  f4[1] = (y[1] + (h*g3[1]));
63  g4[1] = gfxn2(x[0]+(h*f3[0]),x[1]+(h*f3[1]),t+(h));
64
```

Figure 5. Do-while function, usage of arrays assigning values for each new iteration

The group also used functions that when called, uses values provided by the code to solve for the values of g per row:

```
float gfxn1(float x1, float x2)
{
    return(-3*x1+2*x2);
}

float gfxn2(float x1, float x2, float t)
{
    return(x1-1.5*x2+sin(t));
}
```

Figure 3. Functions used for solving g-values

The user-inputted values were only until what time the program will iterate the displacement and which step size is to be used. Due to the simplicity of the code, the output was equally less confusing and intuitive. The data in the output presented was already shown as a 2x1 matrix so it would be easier for the user to see the differences in displacement if the two bodies in motion.

```
"C:\Users\Jericho Mosquera\Desktop\Code Blocks\Problem 2 - RK4 Method.exe"
Enter until what time the values will be computed:5
Enter step size: 0.25


Given that initial at time = 0, x(0) = 0 and x'(0) = 0


Iteration       TIME            f1              f2              f3              f4              x-vector        g1
0               0.000000        0.000000        0.000000        0.000000        0.000000        0.000000        0.00
                                0.000000        0.000000        0.000000        0.000000        0.000000        0.00

1               0.250000        0.000000        0.000000        0.000000        0.000000        0.000000        0.00
                                0.000000        0.000000        0.015584        0.031169        0.002597        0.00

2               0.500000        0.000325        0.000974        0.001923        0.005362        0.000478        0.00
                                0.030844        0.061283        0.075423        0.118596        0.020216        0.24

3               0.750000        0.005013        0.009888        0.013360        0.024761        0.003656        0.03
                                0.118616        0.174813        0.185320        0.249543        0.065567        0.44

4               1.000000        0.023853        0.038874        0.045564        0.070452        0.014622        0.12
                                0.249862        0.323230        0.328485        0.404138        0.147127        0.58

5               1.250000        0.069294        0.100592        0.109988        0.152712        0.041421        0.25
                                0.404616        0.484041        0.483241        0.559120        0.267889        0.63
```

Figure 6. Cutout of Iterations Displayed

Finally, when the data was graphed the displacement showed a sinusoidal behavior explaining the concept that a spring is observed to exhibit a simple harmonic oscillation.

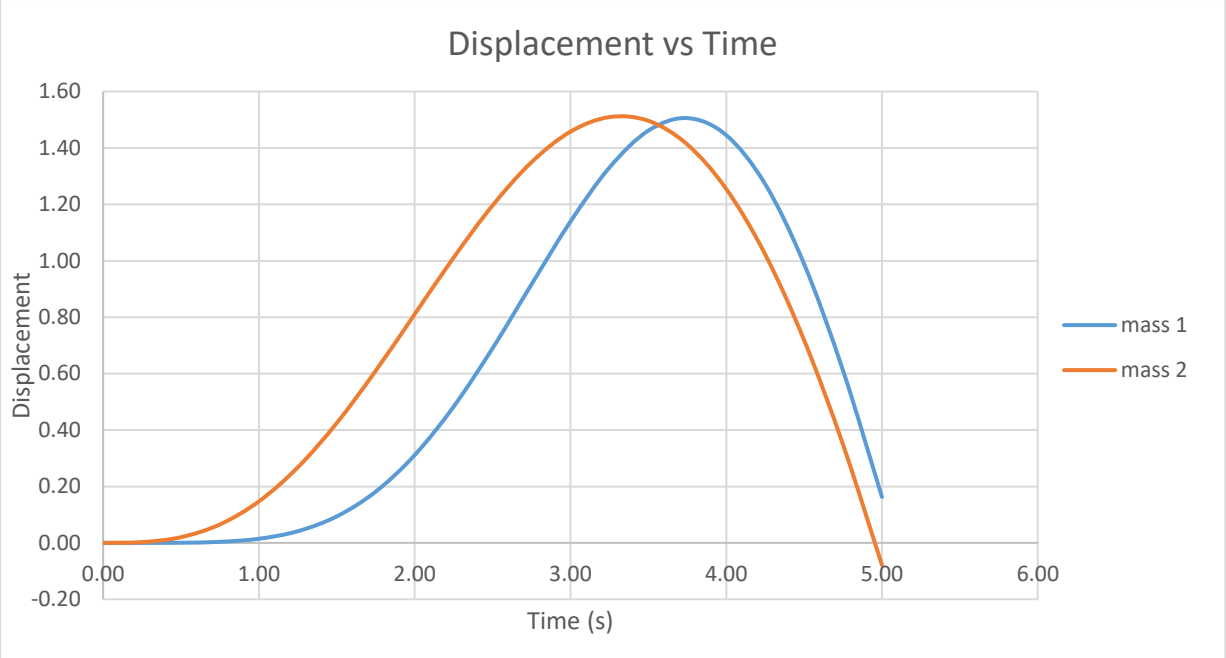


Figure 7. Displacement vs Time Graph of the Two Masses

The second part of the problem 2 focused on the concept of the shifted-inverse power method. The initial problem for the group was to understand the lesson of the said method. Once the initial challenge of understanding that after the alpha is subtracted to the given matrix and that resulting matrix was to be inversed, the group considered the method as a normal power method with pre-steps before the actual power method to be done. The main difference of the shifted-inverse power method, aside from the pre-steps, was that its resulting "eigenvalue" is to be inversed again then have alpha added to it before one could consider it as the problem's eigenvalue. The final step was to get the natural frequency which was the square root of the eigenvalue output.

```
211
212
213     printf("\n\n\n~~FINAL OUTPUT~~\n");
214     printf("Dominant mu-value = %f\n", mu);
215     printf("Dominant eigenvalue = %f\n", sqrt((1/mu)+alpha));
216     printf("mode shapes: (%f,%f)", z[0],z[1]);
217
```

Figure 8. Presentation of final output values

In the code, the values of the A-matrix, the alpha and the initial x-column matrix was asked by the program before proceeding to inverse the matrix and perform iterations to find the dominant eigenvalues and mode shapes. Since the program could only present the dominant eigenvalues, the user must input the values nearest to the theoretical lambda values. The group used the initial alpha values, 0 and 3 to arrive at the correct natural frequencies and mode shapes.

```
16              (1.000000,-0.425389)    1.175394        (1.000000,-0.425389)    3.850779        1.962340
17              (1.000000,-0.425391)    1.175389        (1.000000,-0.425391)    3.850782        1.962341

Iteration               X-vector        mu-value                Eigenvector     Lambda-value    omega-value



~~FINAL OUTPUT~~
Dominant mu-value = 1.175391
Dominant eigenvalue = 1.962341
mode shapes: (1.000000,-0.425390)
```

Figure 9. Sample output of program (α=3)

IV. References

[1] "Numerics for ODEs and PDEs." *Advanced Engineering Mathematics* , 10th ed., pp. 903–905.

V. Appendices

List of Main Variables:

- *int*
  - i, j, k, m, n – used in for loops, dummy variables
- *float*
  - *maxtime – user inputted time value for the iteration (Runge-Kutta part)*
  - *gfxn1&gfxn2 – functions called to output the equations for the g-values (Runge-Kutta part)*
  - *x[30] – array used to store either a column x-matrix or x-values in general (both parts)*
  - *mu – output eigenvalue (Shifted-Inverse Power Method part)*
  - *alpha – user inputted value to be subtracted to initial matrix (Shifted-Inverse Power Method part)*