Imperial College London

MATHEMATICS COURSEWORK

IMPERIAL COLLEGE LONDON

DEPARTMENT OF ELECTRICAL AND ELECTRONIC ENGINEERING

# Numerical Analysis of ODEs using Matlab

*Group: 25*

*Authors:*
Mwanakombo Hussein (email: mh4115@ic.ac.uk)
Aufar Laksana (email: apl115@ic.ac.uk)
Zihan Liu (email: zl6114@ic.ac.uk)
Calvin Chan (email: cc6815@ic.ac.uk)
Yann Herklotz (email: ymh15@ic.ac.uk)

Date: March 15, 2017

# Contents

# 1 Introduction

In this coursework, we focus on solving the differential equation related to circuit analysis, in particular, RL circuits and RLC circuit using numerical analysis techniques such as the Runge-Kutta method and Heun's method. Furthermore, we also simulate the 1-D heat equation and compare the difference of each method in Matlab.

In part one we encounter a high-pass filter (RL circuit) which takes an input signal and only passes the higher frequency components. The purpose of this exercise is to calculate the output signal using midpoint, Heun's and Ralston's method. We further compare the output we have to an exact solution to determine the error in each numerical method.

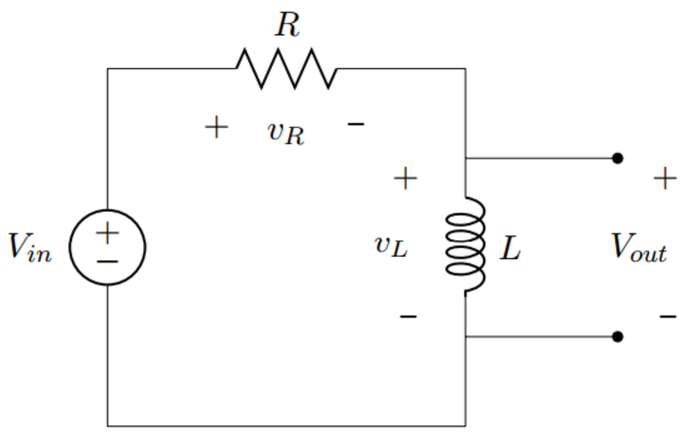

**Figure 1:** Series RL Circuit

In part two we encounter a harmonic oscillator(RLC circuit), namely a device that resonates to a sinusoidal input signal. RLC circuits are commonly used as a bandpass filter circuit. The purpose of this exercise is to calculate the output signal using Runge-Kutta 3/8 method, which is introduced in the original paper of Numerical Analysis of Runge-Kutta but not as commonly used as the classic Runge-Kutta 4th order method.

Then, we encounter the 1-D heat equation in part 3, which is an equation to calculate the thermal energy transfer from regions of higher temperature to regions of lower temperature in a period of time. In this case, we are not using the physical property of the heat/diffusion equation. Instead, we use the finite difference method with zero boundary condition which is outlined in lectures to solve and simulate the equation.

# 2 RL Circuit

## 2.1 RC Equation

### 1. Step signal with amplitude Vin = 3.5V

With the step signal as an input we observed the current of the inductor increasing from 0 to 7A gradually (Figure 2), since the current through the inductor cant change instantaneously. To preserve the fact that the inductor is unable to change instantaneously, the output voltage jumps to 3.5 volts creating no voltage difference across the RL components, thereby no current flow. For current to flow the voltage decreases exponentially, since the voltage across the inductor can change instantly (Figure 3). The rate of change (the slope) is greatest at the beginning, when the current is highest. The ratio R/L determines the steepness of the exponential response.

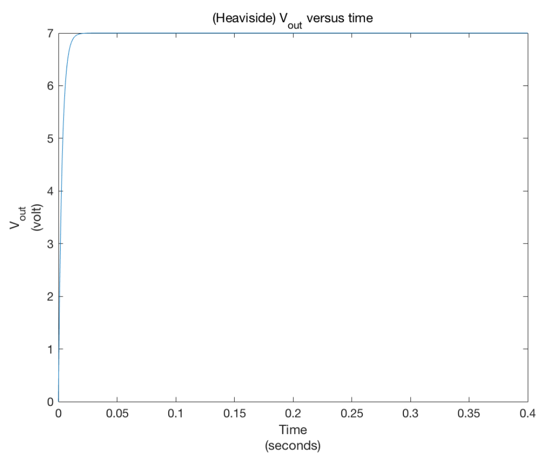

**Figure 2:** Heaviside $V_{out}$ against Time

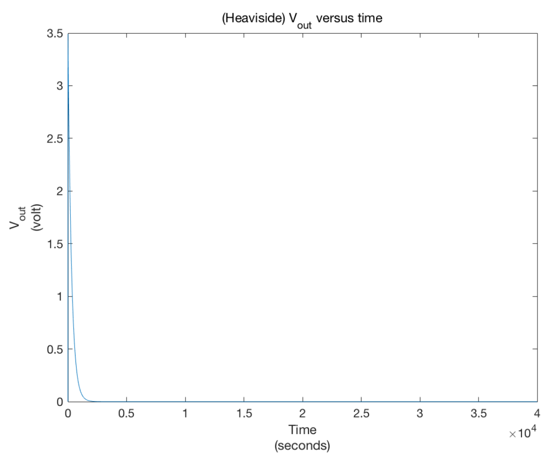

**Figure 3:** Heaviside $V_{out}$ against Time

Kirchoff's is applied:

$$V_{in}(t) = Ri_L(t) + L\frac{d}{dt}i_L(t)$$

When we rearrange the equation we get:

$$\frac{di_L(t)}{i_L(t) - \frac{V_{in}(t)}{R}} = -\frac{R}{L}dt$$

We integrate both sides:

$$\int_0^{i_L(t)} \frac{dx}{x - \frac{V_{in}(t)}{R}} = -\frac{R}{L}\int_0^t dy$$

result of integration:

$$ln\frac{i(t)-\frac{V_{in}(t)}{R}}{i_L(0)-\frac{V_{in}(t)}{R}} = -\frac{R}{L}t$$

I0 is the current at t(0) once we take inverse log we get:

$$i(t) = \frac{V_{in}(t)}{R} + (i_L(0) - \frac{V_{in}(t)}{R})e^{-\frac{R}{L}t}$$

Initially I0 is 0 therefore we get:

$$i(t) = \frac{V_{in}(t)}{R} - \frac{V_{in}(t)}{R}e^{-\frac{R}{L}t}$$

using the equation supplied the output is given by:

$$V_{out} = V_{in}(t) - RiL(t)$$

The step input results to the expected output with all three second-order Runge Kutta methods: Heun, Midpoint and Ralston.

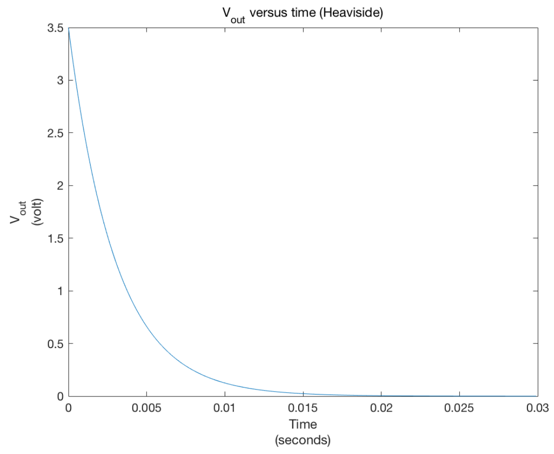

**Figure 4:** The Midpoint method Heaviside

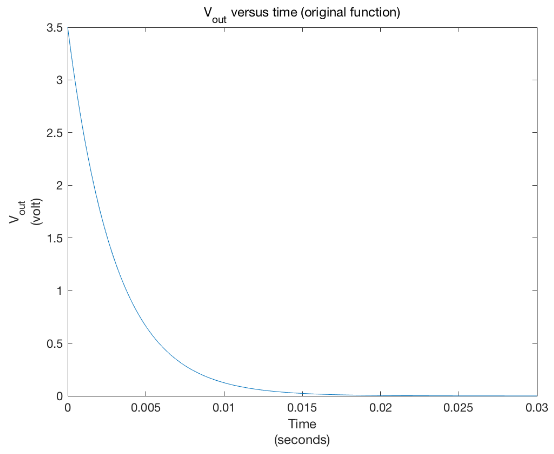

**Figure 5:** The Ralson method Heaviside

**2. Impulsive signal and decay**

$$V_{in} = \bar{V_{in}}(t)exp\left\{-\frac{t^2}{\tau}\right\}$$

$$V_{in} = \bar{V_{in}}(t)exp\left\{-\frac{t}{\tau}\right\}$$

With $\bar{V_{in}} = 3.5$V and $\tau = 150(\mu s)^2$ resp.

Using the same argument above, we see that with an input of an exponential function, we numerically expect $V_{out}$ to be an exponential function as well. This was observed using Heun, Midpoint and Ralston.

**3. Sine Wave with amplitude $V_{in}$ = 4V and different periods T = 150 $\mu$s, T = 15$\mu$s, T = 400$\mu$s, T = 1100$\mu$s.**

**Heun** We observe with Heun that with periods of 150$\mu s$ increasing to 1100$\mu s$, the expected output of a sinewave is seen. However when the period is much smaller, the input sinewave produces an incorrect output that resembles a triangle wave, as seen for the example of 15$\mu s$.

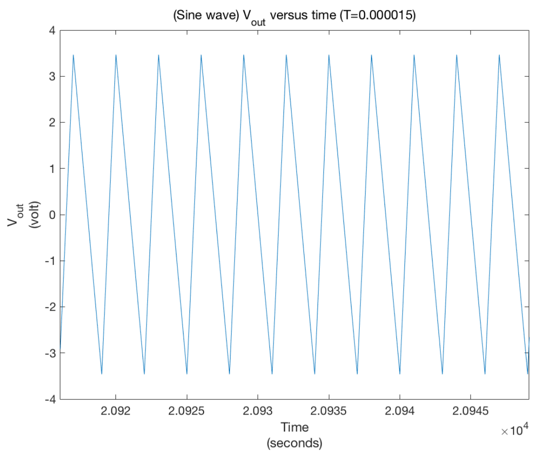

**Figure 6:** T = 150$\mu s$

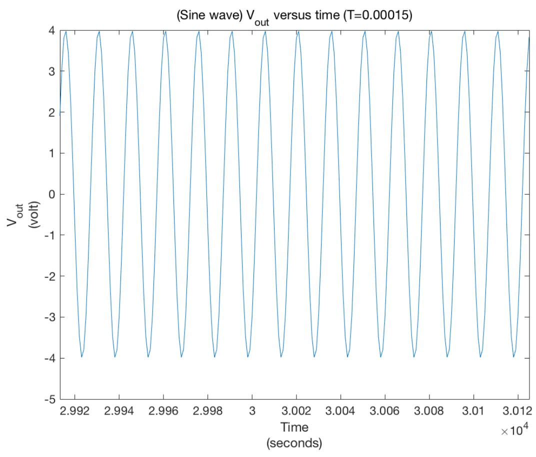

**Figure 7:** T = 15$\mu s$

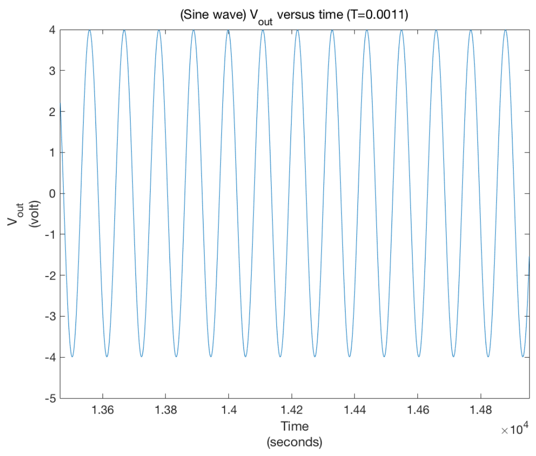

**Figure 8:** T = 1100$\mu s$

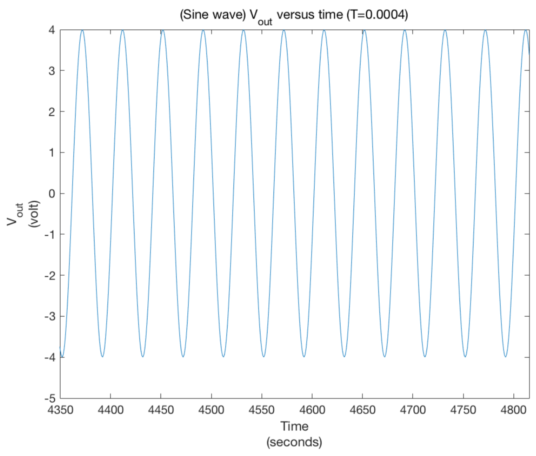

**Figure 9:** T = 400$\mu s$

**Midpoint** The opposite is observed with midpoint. Periods smaller than 1100$\mu$s produce an incorrect output resembling a triangle wave. It is observed that the midpoint method requires a period 1100$\mu$s and above to form a correct sinusoidal output.

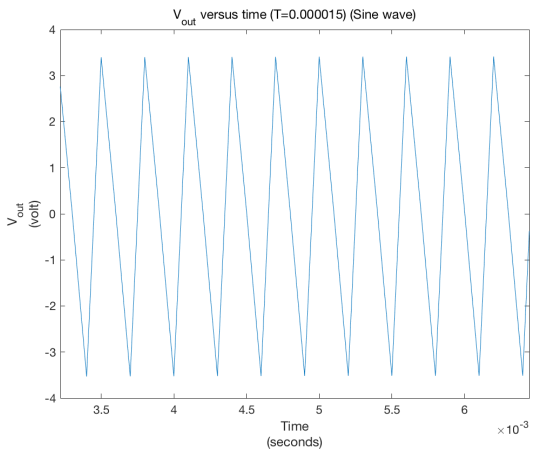

**Figure 10:** T = 150$\mu$s

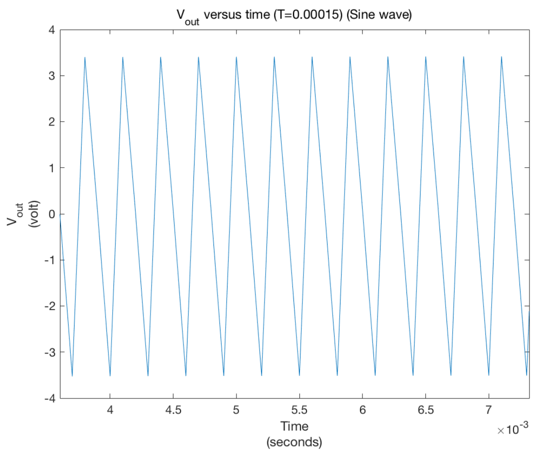

**Figure 11:** T = 15$\mu$s

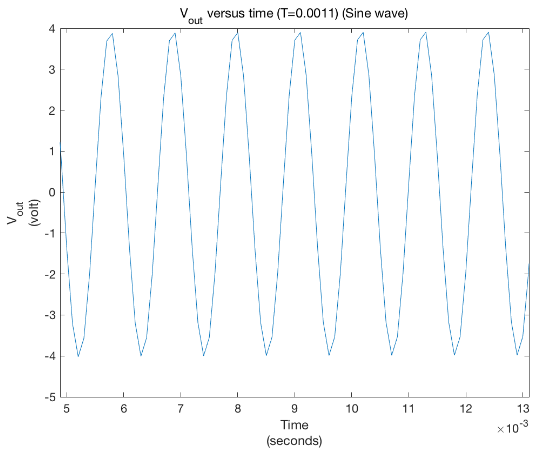

**Figure 12:** T = 1100$\mu$s

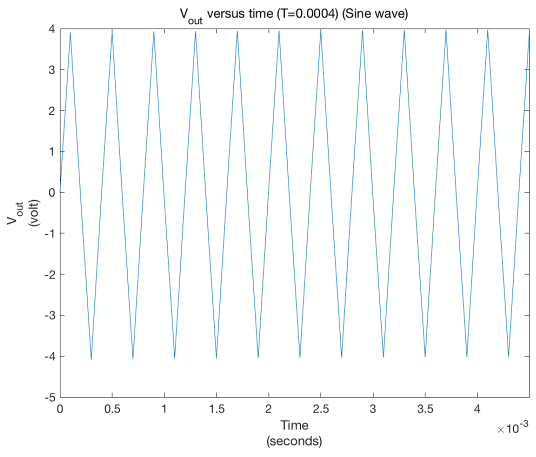

**Figure 13:** T = 400$\mu$s

**Ralston** From the graph we can observe a result very similar to the midpoint method.

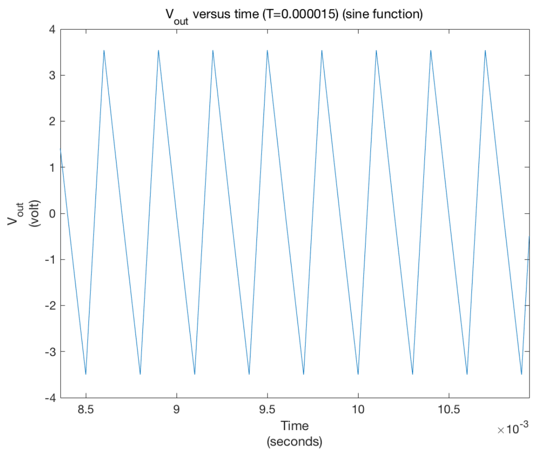

**Figure 14:** T = 150$\mu$s

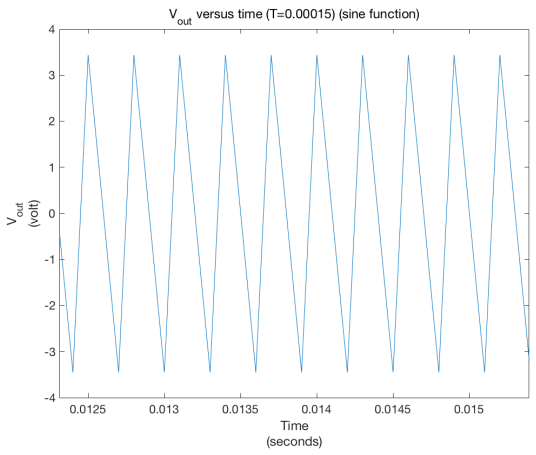

**Figure 15:** T = 15$\mu$s

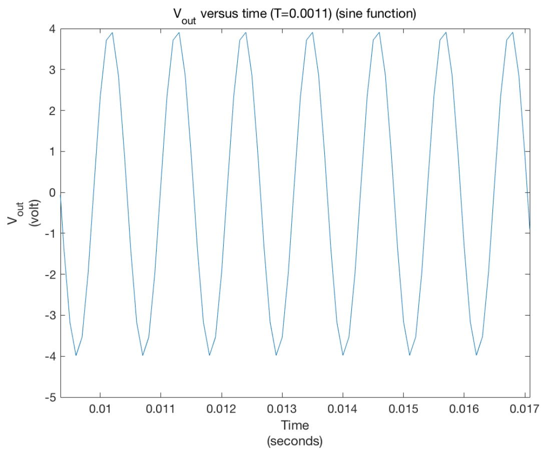

**Figure 16:** T = 1100$\mu$s

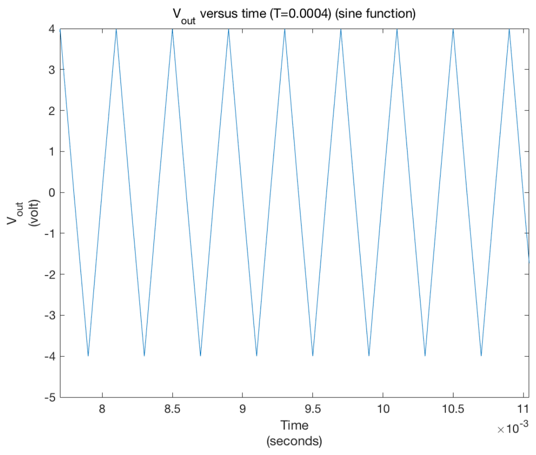

**Figure 17:** T = 400$\mu$s

**3. Square Wave with amplitude $V_{in}$ = 4V and different periods T = 150 $\mu$s, T = 15$\mu$s, T = 400$\mu$s, T = 1100$\mu$s.**

**Heun** We expect the RL circuit output to be a squarewave responding to a squarewave excitation. This is what is oberved with the Heun method. Heun is not perfect as we see that with periods shorter than 150$\mu s$ the output does not resemble a squarewave and is instead, distorted. As the period of the squarewave increases the output is corrected and resembles a squarewave at 1100$\mu s$.

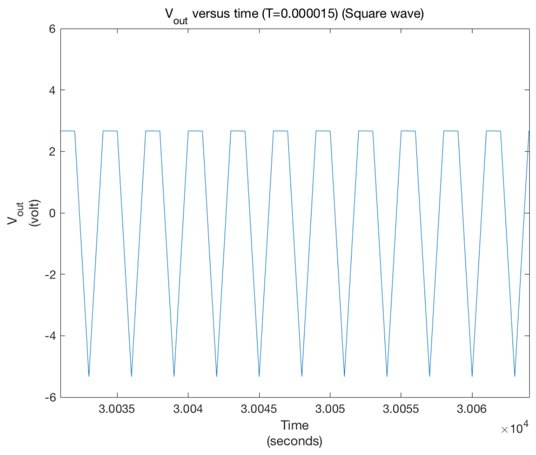

**Figure 18:** T = 150$\mu s$

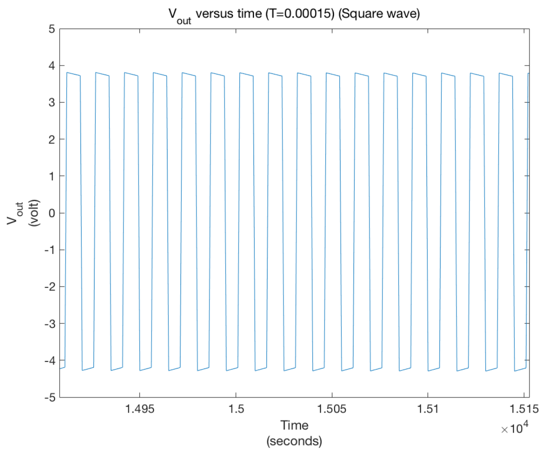

**Figure 19:** T = 15$\mu s$

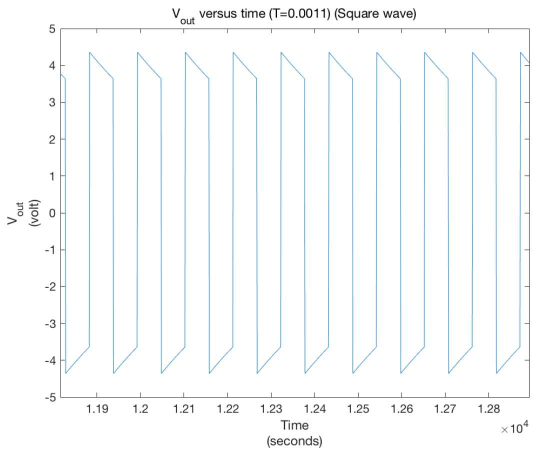

**Figure 20:** T = 1100$\mu s$

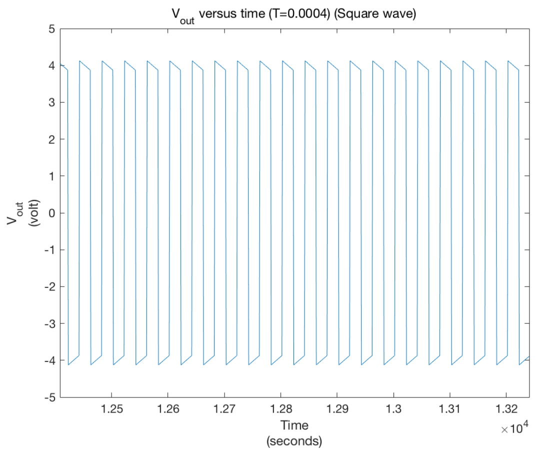

**Figure 21:** T = 400$\mu s$

**Midpoint** The Midpoint method responds worse than the Heun method since the output is distorted using any period shorter than 400$\mu$s.

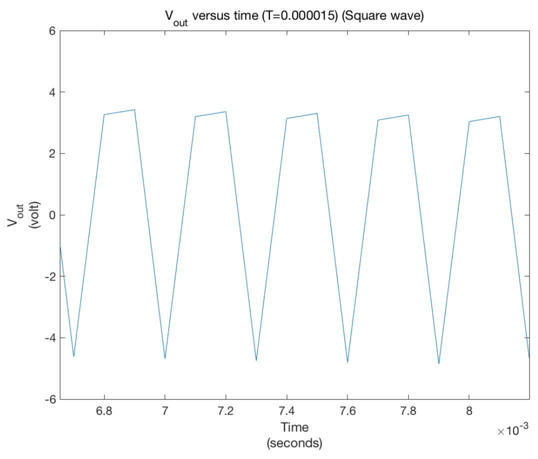

**Figure 22:** T = 150$\mu$s

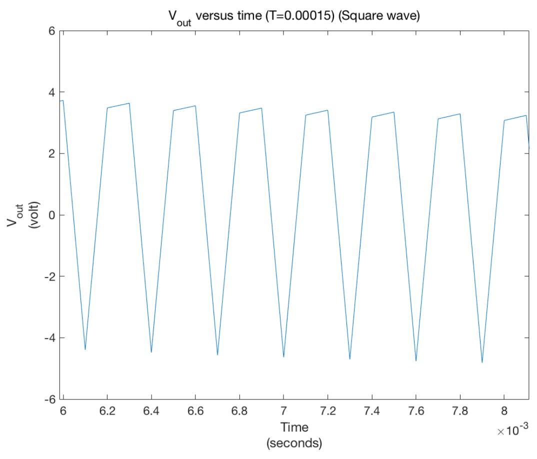

**Figure 23:** T = 15$\mu$s

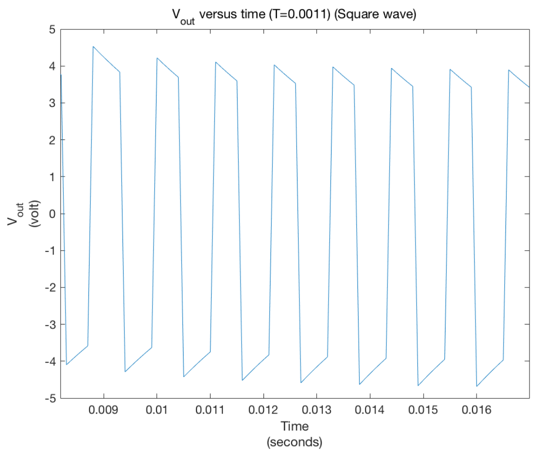

**Figure 24:** T = 1100$\mu$s

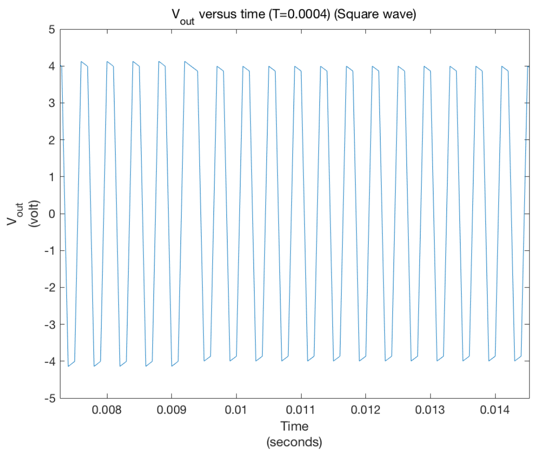

**Figure 25:** T = 400$\mu$s

**Ralston** The Ralston method responds worse than both the Heun and Midpoint method since the output is distorted using any period shorter than 1100$\mu$s.

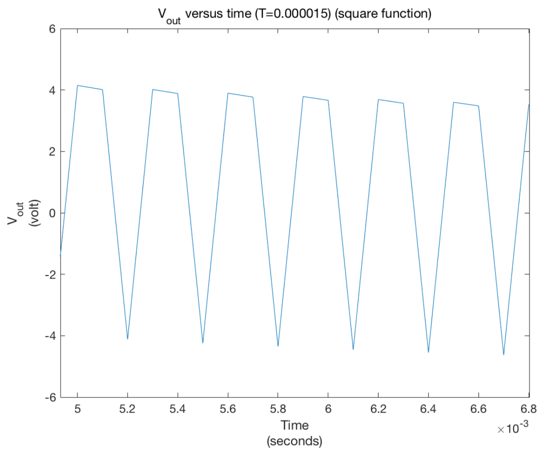

**Figure 26:** T = 150$\mu$s

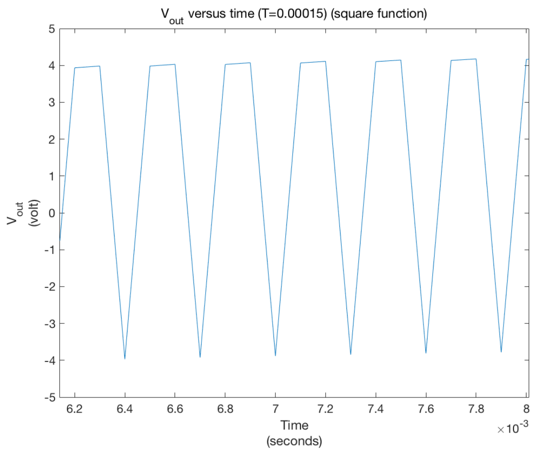

**Figure 27:** T = 15$\mu$s

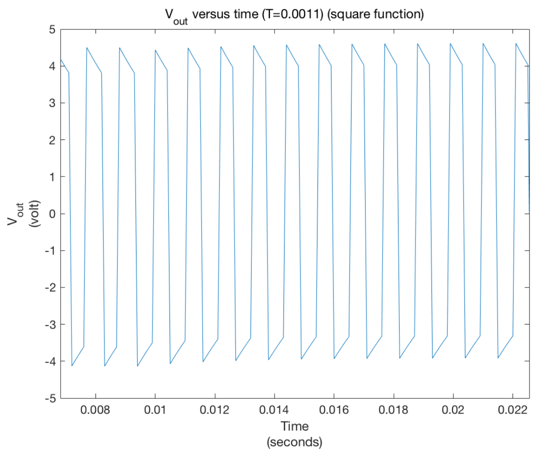

**Figure 28:** T = 1100$\mu$s

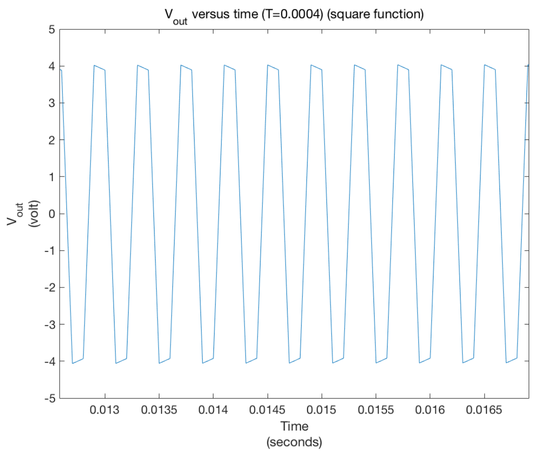

**Figure 29:** T = 400$\mu$s

**3. Sawtooth Wave with amplitude $V_{in}$ = 4V and different periods T = 150 $\mu$s, T = 15$\mu$s, T = 400$\mu$s, T = 1100$\mu$s.**

**Heun** The RL circuit response is meant to be a sawtooth wave responding to a sawtooth wave input. The Heun method gives a great response at periods 1100$\mu$s and larger. Any periods smaller than this produces distorted outputs, but can still be identified as sawtooth waves.

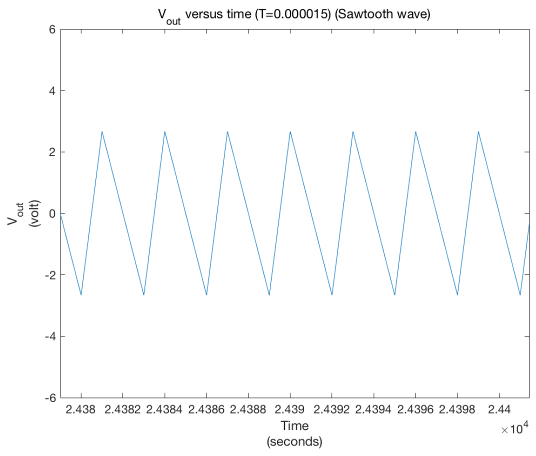

**Figure 30:** T = 150$\mu$s

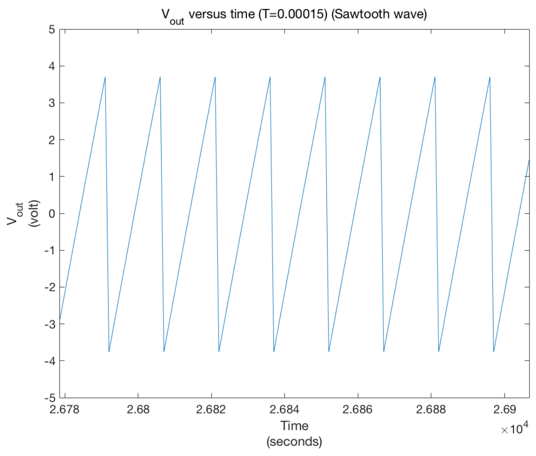

**Figure 31:** T = 15$\mu$s

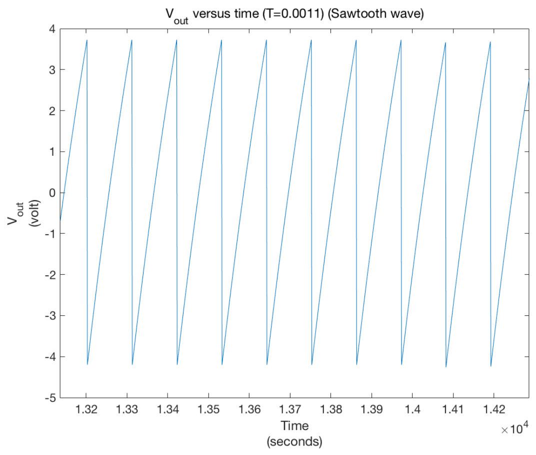

**Figure 32:** T = 1100$\mu$s

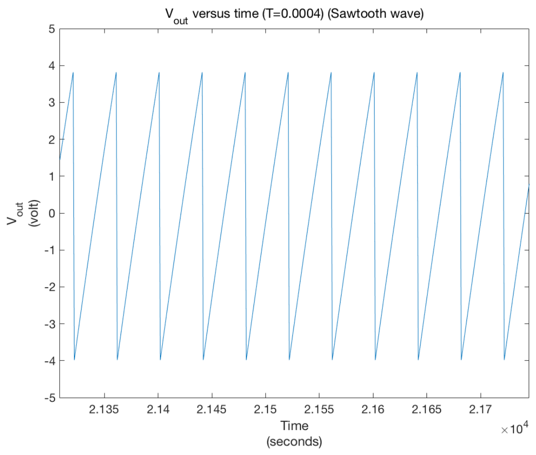

**Figure 33:** T = 400$\mu$s

**Midpoint** The maximum voltage output increases with the period increases, and it performs the best at 400$\mu$s.

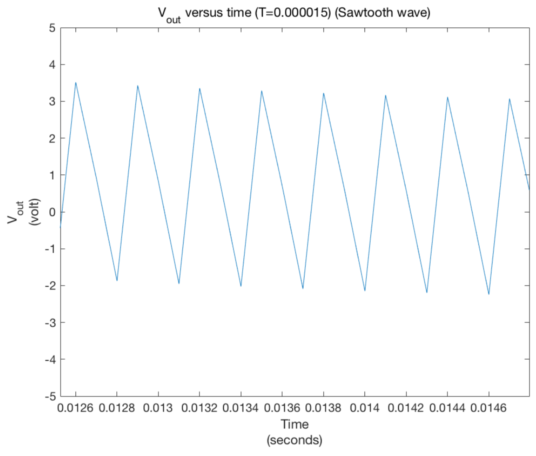

**Figure 34:** T = 150$\mu s$

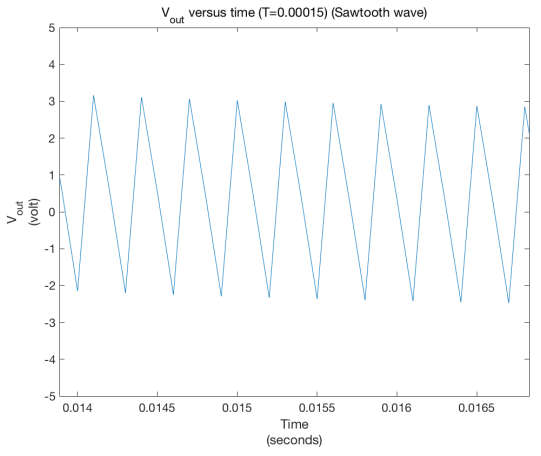

**Figure 35:** T = 15$\mu s$

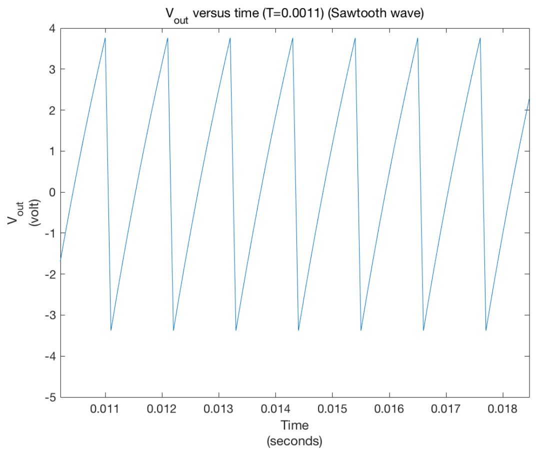

**Figure 36:** T = 1100$\mu s$

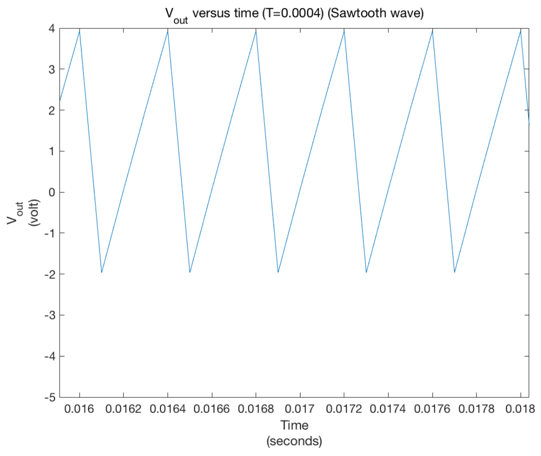

**Figure 37:** T = 400$\mu s$

**Ralston** The trend of Ralston plot is similar to the other methods.

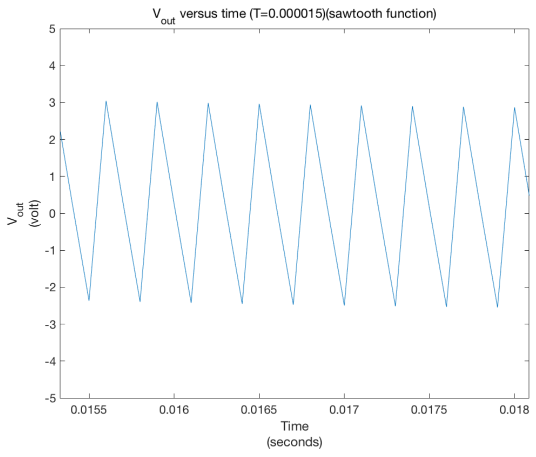

**Figure 38:** T = 150$\mu$s

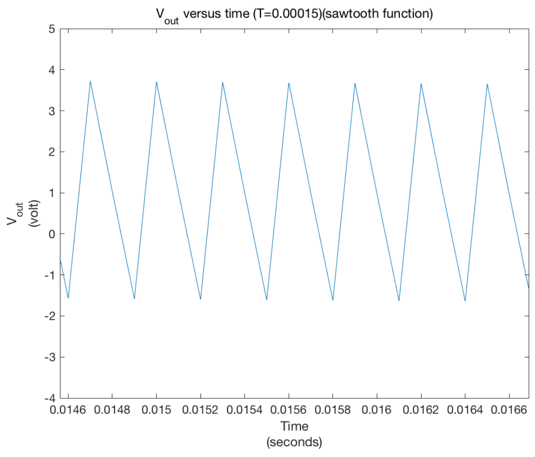

**Figure 39:** T = 15$\mu$s

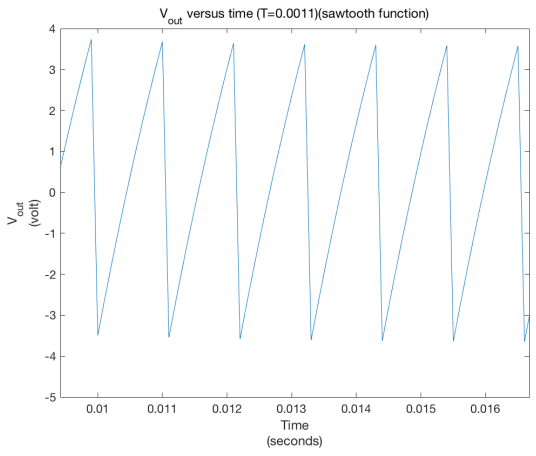

**Figure 40:** T = 1100$\mu$s

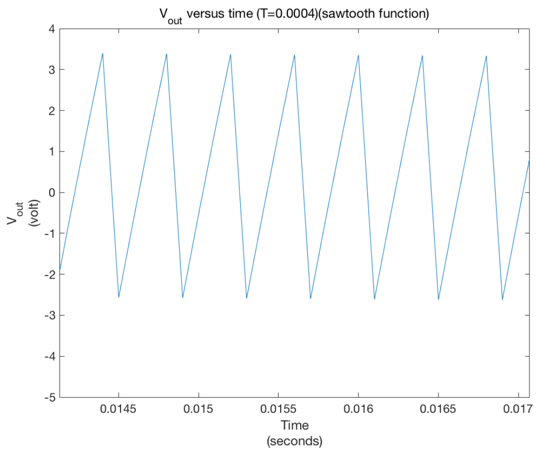

**Figure 41:** T = 400$\mu$s

## 2.2 Error Analysis

The exact solution was obtained using the reliable method of manually calculating the differential equation.

Exact solution derivation: The error was obtained as a function of t: Error is the term used to denote the amount by which an approximation fails to equal the exact solution. This is given by:
**Error** = **Exact**(Exact Solution) **Vout**(Numerical Solution).
Below are the graphs of the error function.

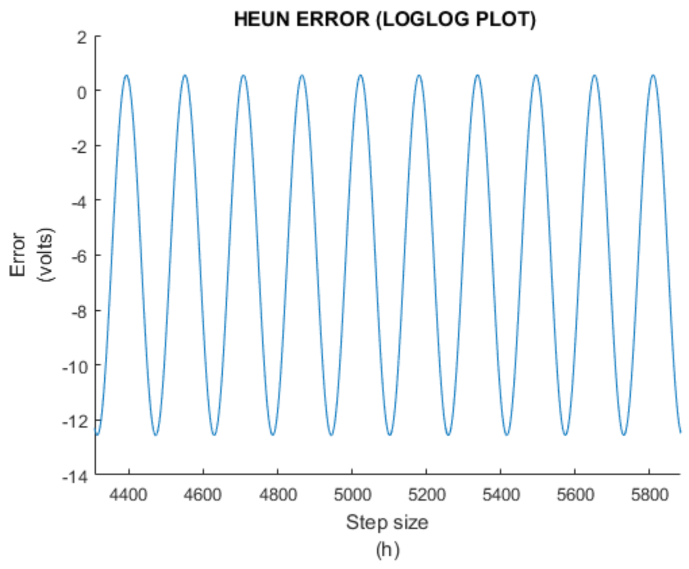

**Figure 42**

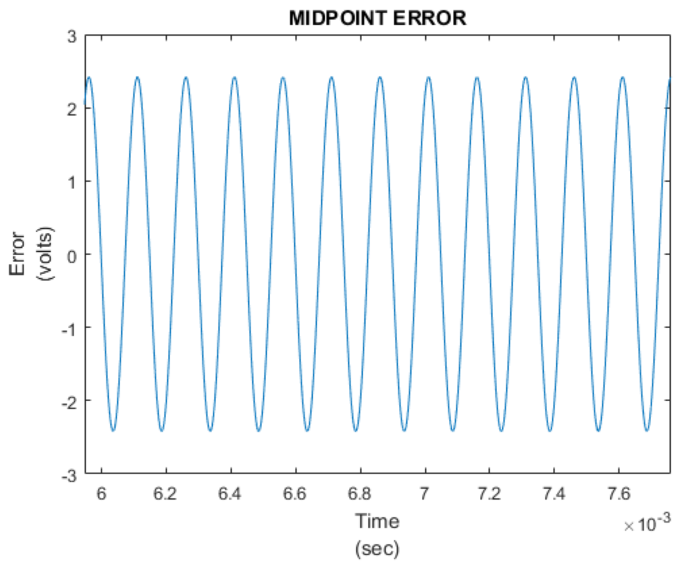

**Figure 43**

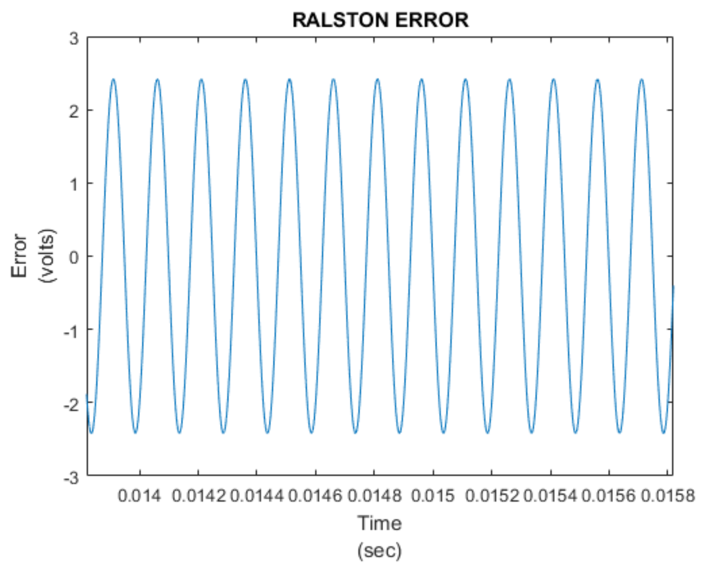

**Figure 44**

The results are realistic since the exact and numerical solution are both cosine waves, therefore the error between the two waves is expected to be a cosine wave as well.

To obtain an accurate log-log plot to show the order of the error by varying the step size h we took these steps:

We programmed a for loop that increases the value of the step size by a power of 2 each time. The for loop starts with the index at 16 and loops 10 times. This was done since the step size was updated using the equation $h = 2^{-index}$. After trial and error, we observed indexes too small led to a step size thats too large and thereby produces large errors as results. The index and the range were chosen carefully. The index starts at 16 and increases as this is the range whereby the error produced by each method was a stable sine graph.

The for loop produces decreasing h values from $2^-16$ to $2^-25$.

### Log-log Error Analysis

There are two types of truncation errors we can analyse with these methods.

**1. Local Error** This is the error within one step due to application of the numerical method.

**2. Propagation Error** This is the error due to previous local errors. Global Truncation Error = Local + Propagation

When the local truncation error is $O(h^{n+1})$then the global truncation error is $O(h^n)$. Midpoint and Heun both have a Local Error $(h^3)$ and Global Error $O(h^2)$. Second order Runga Kutta methods have a global truncation error of $O(h^2)$. According to the truncation error we have derived previously, the log-log graphs gradient should be theoretically at $h^2$.

Therefore, we expect the global error of error of the Heun, Midpoint and Ralston methods to be the same. We observe this with the Ralston and Midpoint method. We can clearly observe how close the errors of these two methods are in the below figure.

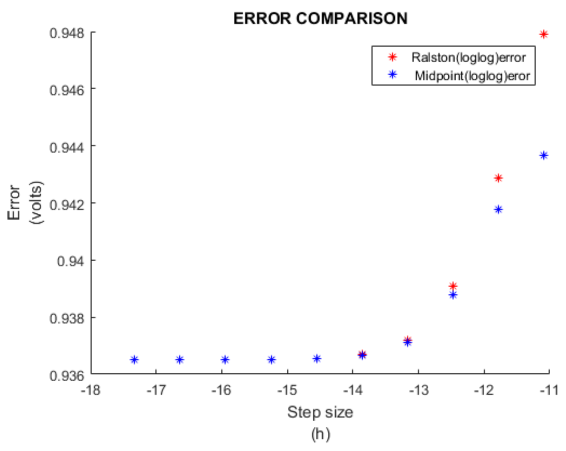

**Figure 45:** Ralston - Midpoint error comparison

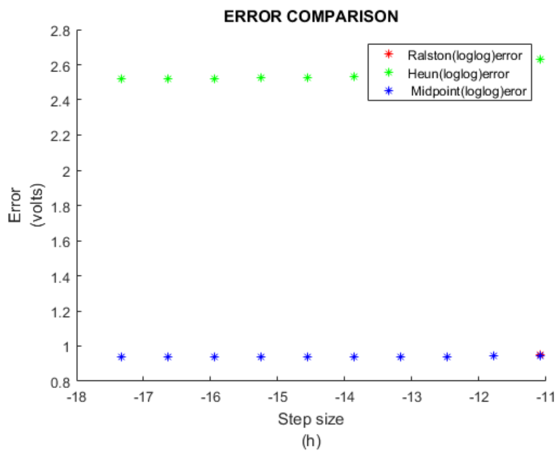

**Figure 46:** Heun - Midpoint - Ralston error comparison

These are the log-log plots that show the **order of the error to be $O(h^2)$ for all methods (Heun, Midpoint and Ralston)** since the shape of these graphs resemble a parabola on a non log-log plot. We observe for all the methods a straight line in the log-log plot when the step size is less than 10⁻6 which is what we as a group expected to see. This is based off our understanding of plotting non linear graphs onto a log-log plot to make it linear. We also observed that since our error is very small converting the plot to log-log using Matlabs' log-log function would produce a semilog graph(with log on the X axis) because of the small values on the y axis. We also observed exponential increases in the error when the step size is too large( in our case larger than 10⁻6) thereby causing our straight line log-log graph to no longer be linear as the step size is increased. This result is due to the fact that when step size is increased the approximation methods becomes more inaccurate thereby leading to a larger error that's no longer O(h).

It is obvious that the truncation error is small in our case because the exact mathematical formula we used is precise and accurate. Furthermore, it is clear that the error produced by the Heun method is much larger than both Midpoint and Ralston which have much smaller errors. However, in general, all the methods we have used perform linearly in the log-log graphs.

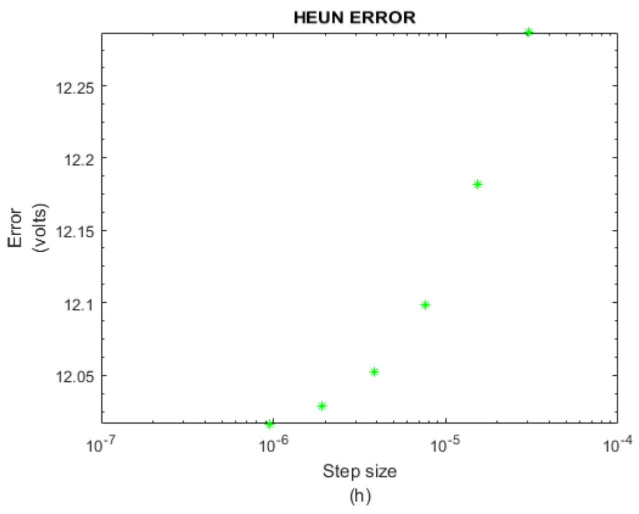

**Figure 47**

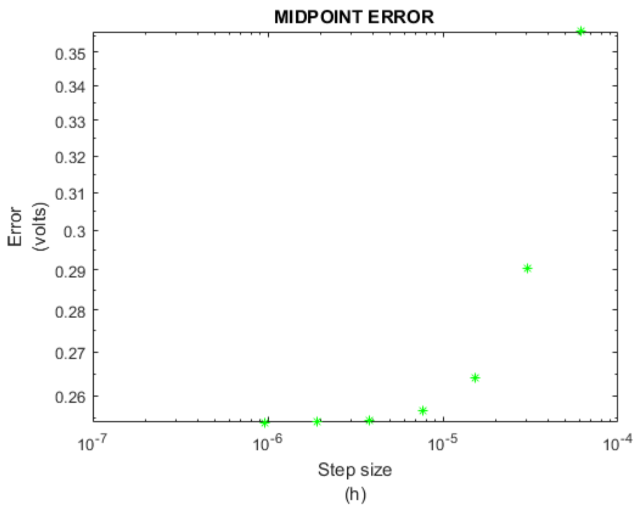

**Figure 48**

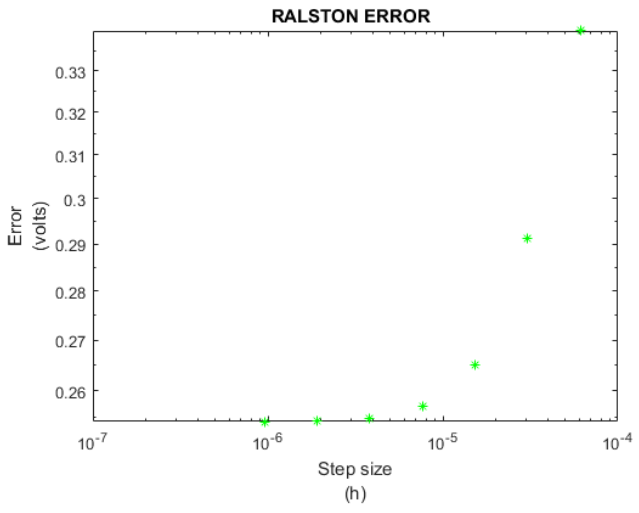

**Figure 49**

# 3 RLC Circuit

## 3.1 RLC Circuit Equation

An RLC circuit consists of a resistor, capacitor and an inductor. For the purpose of this example, we have a series RLC circuit, as shown in the figure below.

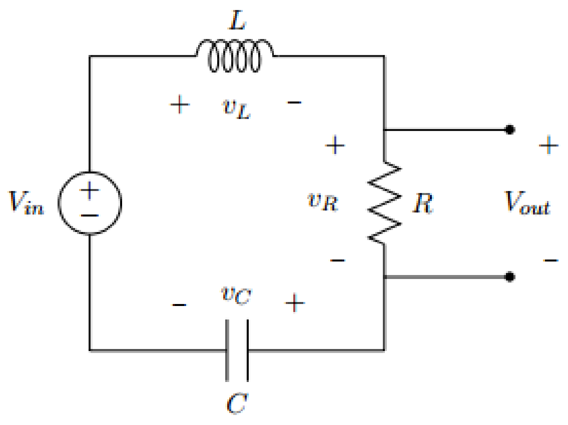

**Figure 50:** Series RLC Circuit

By applying Kirchoff's Voltage Law for each of the three components of the circuit, we can obtain the differential equation representing the circuit.

$$V_R + V_L + V_C = V_{in}(t)$$

$$L\frac{d}{dt}i_L(t) + Ri_L(t) + \frac{1}{C}\int_0^t i_L(\tau)\,d\tau = V_{in}(t)$$

$$L\frac{d^2}{dt^2}q_C(t) + R\frac{d}{dt}q_C(t) + \frac{1}{C}q_C(t) = V_{in}(t)$$

We assume that the capacitor is pre-charged at time t = 0, with $q_C(0)$ = 500nC. We also assume that no current flows through the inductor at time t = 0, so $i_L(0) = \frac{d}{dt}q_C(0)$ = 0A.

We were also given the values of resistances, capacitance and inductance of the components in the circuit:

$$R = 250\Omega,\ C = 3\mu F,\ L = 650mH$$

## 3.2 Runge-Kutta and Coupled Equations

### 3.2.1 Runge-Kutta

Runge-Kutta 4th order method is a numerical technique used to solve ordinary differential equations of the form $\frac{dy}{dx} = f(x,y)$, where $y(0) = y_0$. Therefore, we can only use it to solve first order ordinary differential equations.

### 3.2.2 Coupled Equations

In order to solve the RLC equation derived earlier, we need to rewrite the equation in a *coupled* form. We take the equation:

$$L\frac{d^2}{dt^2}q_C(t) + R\frac{d}{dt}q_C(t) + \frac{1}{C}q_C(t) = V_{in}(t)$$

By letting y $= \frac{d}{dt}q_C(t) = \dot{q}$ and $\dot{y} = \frac{d^2}{dt^2}q_C(t) = \ddot{q}$, we can do:

$$L\ddot{q} + R\dot{q} + \frac{1}{C}q = V_{in}$$

$$L\dot{y} + Ry + \frac{1}{C}q = V_{in}$$

And we end up with:

Equation 1:

$$y = \dot{q}$$

Equation 2:

$$\dot{y} = \frac{V_{in} - Ry - \frac{1}{C}q}{L}$$

## 3.3 Matlab Script

### 3.3.1 Runge-Kutta 3/8 Function

The Runge-Kutta 3/8 method is similar to the classical method, since it evaluates the integrand f(x,y) four times per step. In order to obtain the next step in the approximation, we must calculate:

$$y_{i+1} = y_i + \frac{1}{8}(k_1 + 3k_2 + 3k_3 + k_4)$$

Where:

$$k_1 = hf(x_i, y_i)$$

$$k_2 = hf(x_i + \frac{h}{3}, y_i + \frac{k_1}{3})$$

$$k_3 = hf(x_i + \frac{2h}{3}, y_i - \frac{k_1}{3} + k_2)$$

$$k_4 = hf(x_i + h, y_i + k_1 - k_2 + k_3)$$

and $x_i = x_0 + ih$

### 3.3.2 RLC Script

The RLC script was used to simulate the circuit, by setting up the values of the resistor, capacitor, inductor and the initial conditions. It was also used to determine the step-size h, as well as the number of iterations. In order to test the circuit, we used several different inputs, such as a step-function, an impulse with an exponential decay, a square wave and a sinusoidal wave.

## 3.4 Testing different inputs

### 3.4.1 Step-Signal

For this input, we have a step-function with an amplitude of 5V. When the capacitor gets charged, there is no further change in voltage, so no current passes and so the voltage across the resistor goes to zero.

From manual calculation, we can obtain the *Resonance Frequency*, $\omega_0 = \frac{1}{\sqrt{LC}} = 716 rad/s = 113 Hz$. We can also get the *Attenuation*, which is a measure of how fast the transient response will die away after the input is removed, $\alpha = \frac{R}{2L} = 192$. By comparing the attenuation value with the resonance frequency, we can determine that the system is *underdamped*, since $\omega_0 > \alpha$. This gives us a response of a decaying oscillation, of frequency $\omega_d = \sqrt{\omega_0^2 - \alpha^2} = 110 Hz$. This frequency can be seen in the figure below.

The system experiences a transient behavior when the input function is a step function, the transient happened because the voltage of a capacitor does not change instantaneously. And because of out combination of input resistance, capacitance the system have a nature of underdamping.

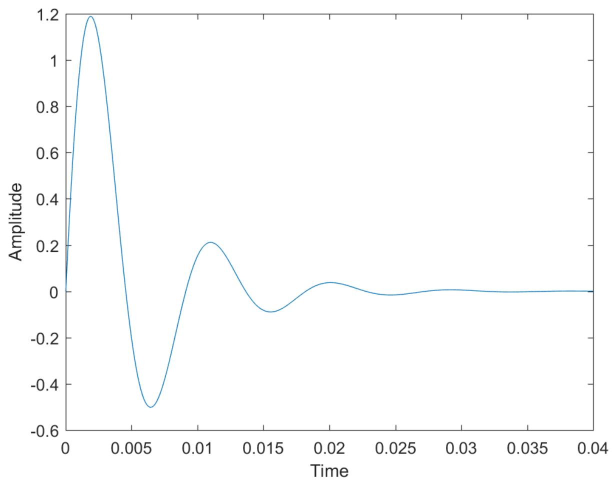

**Figure 51:** Output to a 5V Step-Function

### 3.4.2 Impulsive Signal with decay

Here we apply an impulse at the input which decays exponentially, described by the equation:

$$V_{in} = \bar{V_{in}} exp\left\{-\frac{t^2}{\tau}\right\}$$

Where $\tau = 3x10^{-6}$

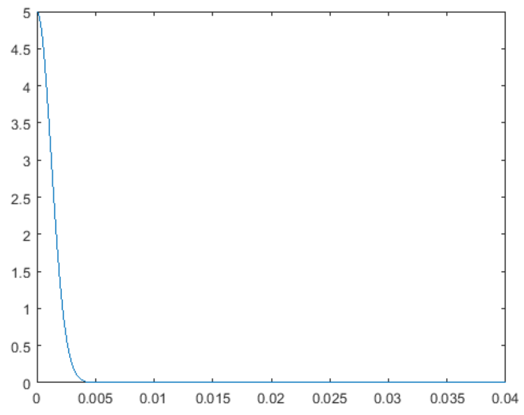

**Figure 52:** Impulsive Signal input

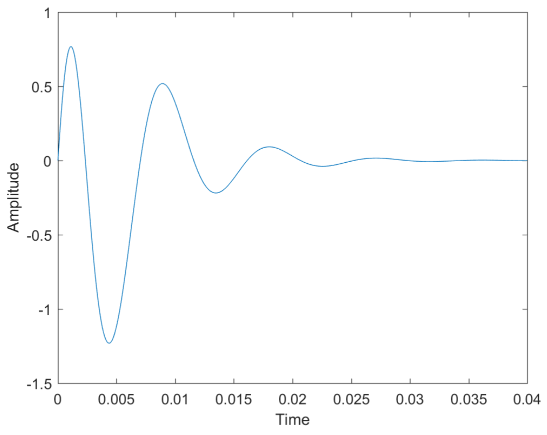

**Figure 53:** Output of RLC circuit

From the figures, we can see that the impulse decays to zero very quickly, in less than 0.0005 seconds. When the impulse is initially applied to the circuit, we can see the transient response in the output, where it begins to rise, to a maximum of 0.75V.

In comparison, when we applied the step-function to the input, the output reached 1.2V, even though initially, the impulse is of infinite amplitude, and the step-function had an amplitude of 5V. This is because the impulse decays to zero quickly, which prevents the RLC circuit from rising.

### 3.4.3 Square Wave

For this input, we have three square waves with constant amplitudes of 5V. Further change in voltage was observed due to the nature of the square wave and the constant discharge time of the capacitor. The behavior of the output also changes with the input square wave frequency.

$$V_{in} = \bar{V_{in}} * X_{square}(2\pi f)$$

Amplitude $\bar{V_{in}} = 5$ V

**f = 5Hz** Here we input the square wave with a frequency of 5 hz

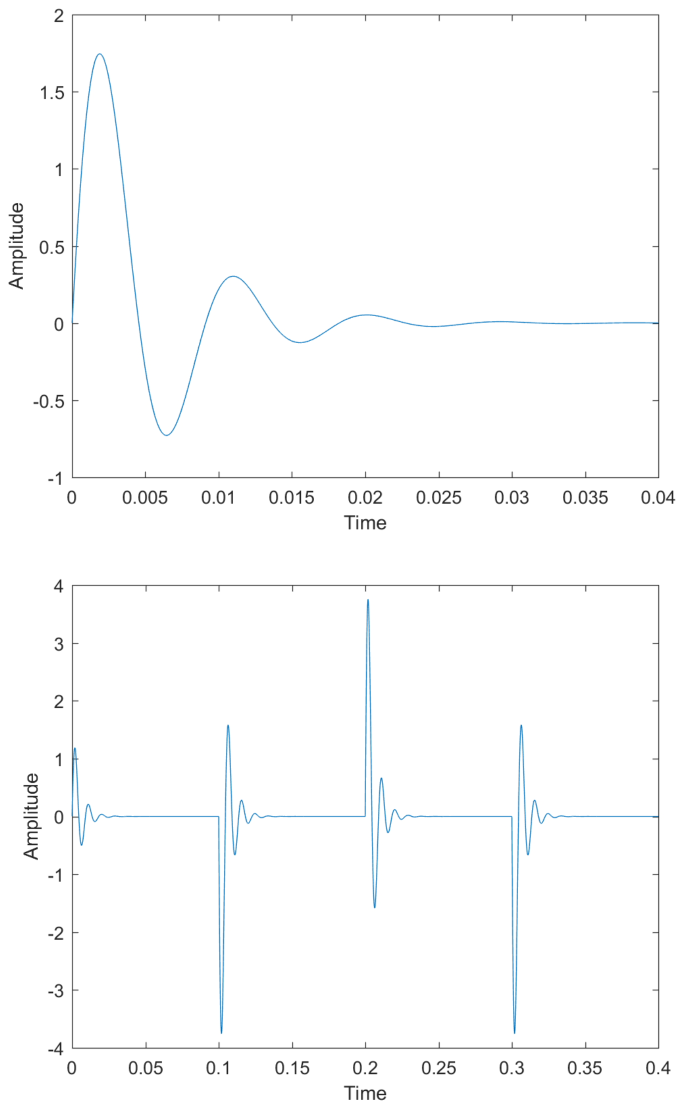

**Figure 54:** Output of a 5Hz Squarewave-Function

In the graphs above we can see that the square wave has a similar behavior as the step function when we set the finishing time to 0.04. This behavior is because a small section of the square wave acts like a step function. In order to observe the whole behavior of the function, we then increase the time frame to 0.4.

We can think of the square wave as a set of the step function in both positive and negative domain, and the transient happens when there is a sudden change in voltage. Since the input is of a low frequency of 5Hz, we see the transient part decay before the capacitor has recharged.

**f = 100Hz** We then apply a square wave of frequency 100Hz

From the figure, we can see that the underdamped input has a much shorter

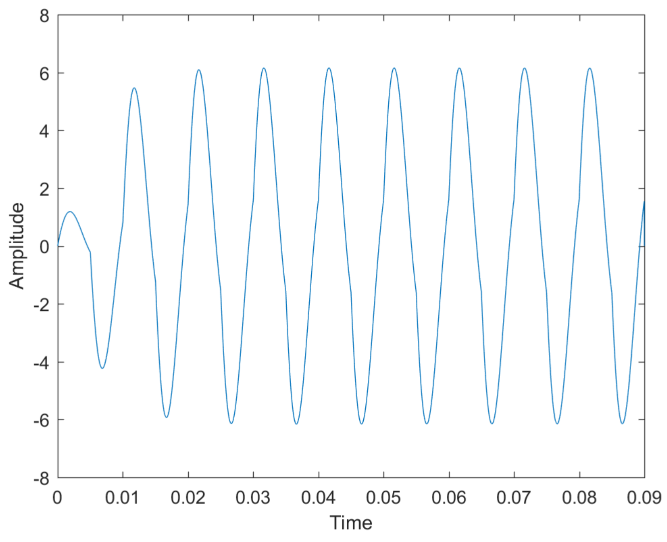

**Figure 55:** Output of a 100Hz Squarewave-Function

time of changing the behavior of the circuit than the last frequency.

**f = 500Hz** Finally we change the input to a square wave with a frequency of 500Hz

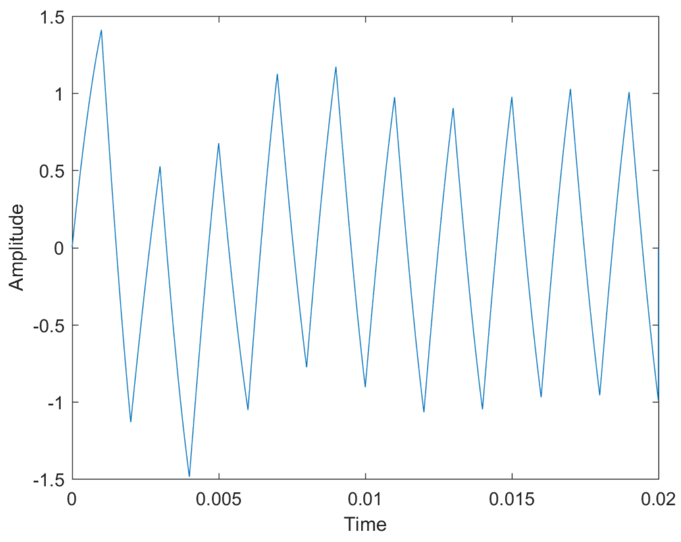

**Figure 56:** Output of a 500hz Squarewave-Function

Here we have an even higher frequency, thus the output is almost becoming a sawtooth function, the peak point of each sawtooth function actually shows the underdamped behavior of the circuit if you link them together.

### 3.4.4 Sine Wave

Here, we are testing the RLC output response to sinusoidal waves at the input, $\bar{V}_{in} = 5sin(2\pi f t)$. We applied sinusoids of different frequencies to the input, 5Hz, 100Hz and 500Hz and observed the output of the circuit.

**f = 5Hz** When we apply a 5Hz sine wave to the input, we get the following response:

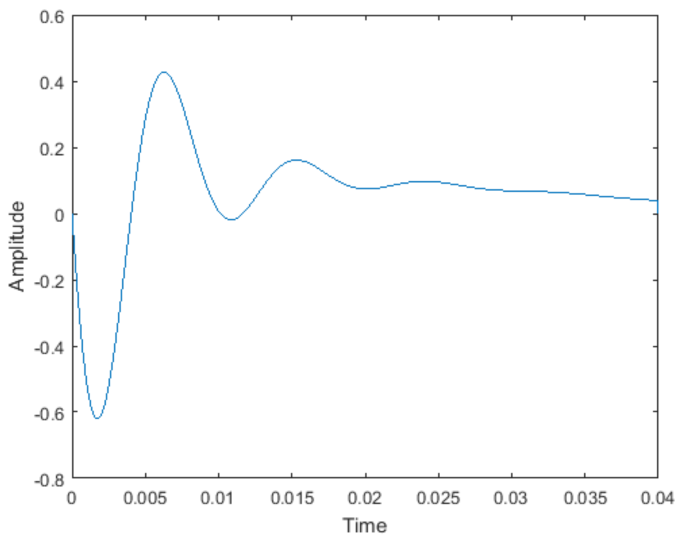

**Figure 57:** A small *df* value does not show the full response

We notice that the output has not reached a steady state by 0.04 seconds. If we increase the *df* value used, we get:

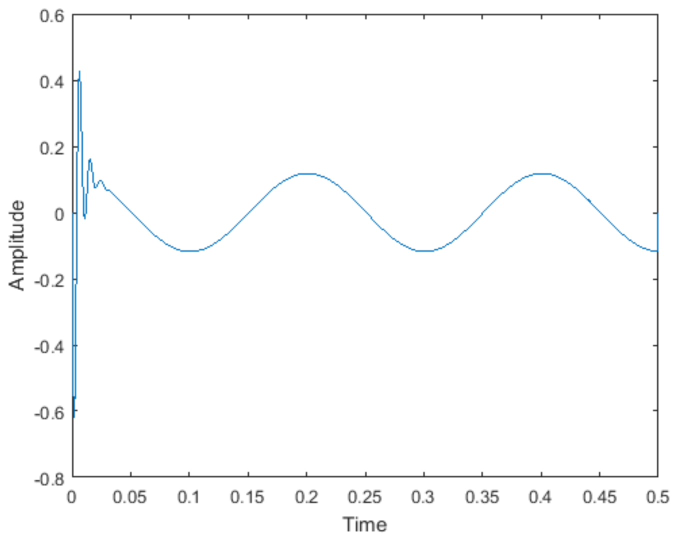

**Figure 58**

As we can see from the figure, there is a change in frequency of the output. Initially, the capacitor is charging up from the sinusoidal input, but since it is an under damped system, it eventually oscillates at 5Hz. However, we notice that the amplitude of the sinusoid is about 0.1V, and not the original 5V input.

**f = 100Hz** When we apply 100Hz to the input of the system,

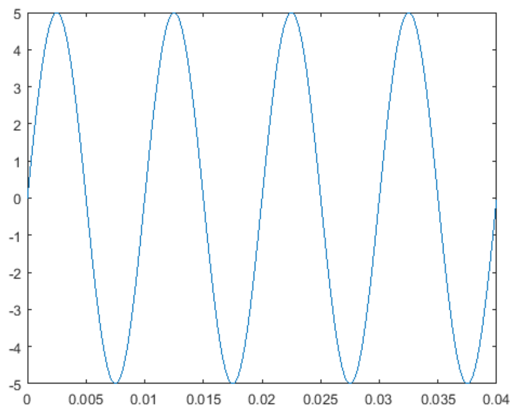

**Figure 59:** 100Hz Sine Wave

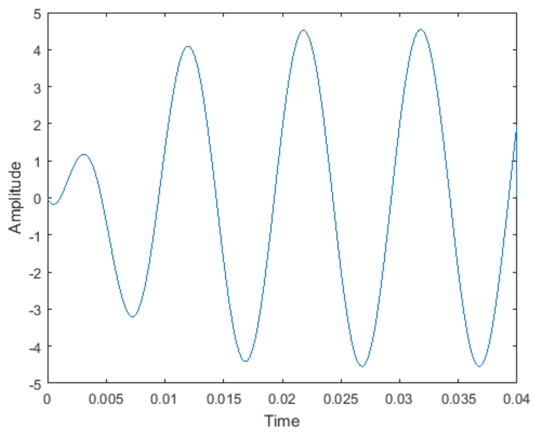

**Figure 60:** Output of RLC circuit

From the figure, we can see that in the output, there is a slight phase shift, as seen by the graph going below the x-axis at time t = 0. This is due to the capacitor introducing a phase shift of -90°, since it is the dominating component at low frequencies. Furthermore, we note that the square wave input becomes a sinusoidal output, this is due to the capacitor charging and discharging times.

**f = 500Hz** When we apply 500Hz to the input of the system, we note that the amplitude of the response is once again quite low. In comparison, at 100Hz, the amplitude of the response is close to the original 5V sinusoid applied at the input.

This is because the RLC circuit acts as a bandpass filter. We can calculate the centwe frequency $\omega_c = \frac{1}{\sqrt{LC}} = 113.97Hz$. We can also workout the bandwidth of the filter $\Delta\omega = \frac{R}{L} = 61.21Hz$, which means that frequencies between 52.76Hz and 175.18 will be passed, and frequencies outside that range will be suppressed, which is why the 5 Hz and 500Hz sine wave inputs have a low output amplitude.

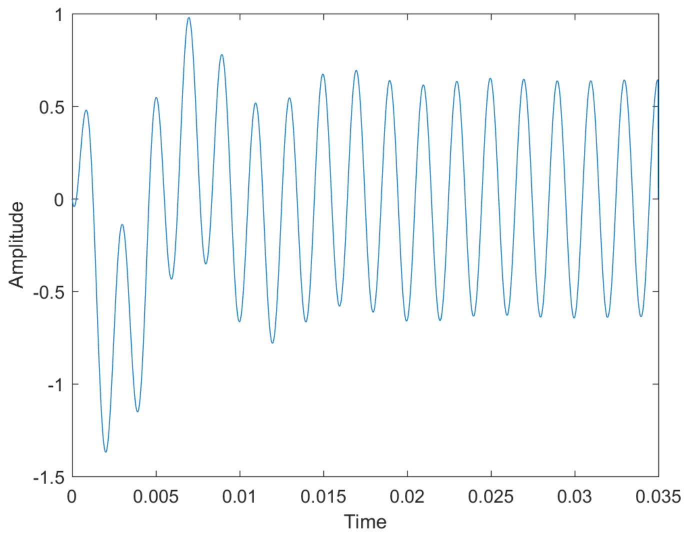

**Figure 61:** 500Hz sine wave

# 4 Finite Differences for PDE

## 4.1 1-D Heat Equation

The 1-D heat equation

$$\frac{\partial y}{\partial t} = \frac{\partial^2 y}{\partial x^2}, \quad 0 < x < 1, \quad t > 0$$

with zero boundary conditions $y(0,t) = y(1,t) = 0$ and initial conditions $y(x,0) = y_0(x)$ can be solved numerically using the finite difference method.

## 4.2 Method

The finite difference method consists of approximating a differential equation using difference equations. As the 1-D heat equation depends on $x$ and $t$ we can use the finite difference method to approximate the heat equation when $t = 0$ and then increment $t$ to get the heat equation as it changes over time.

## 4.3 Matlab Script

There were certain aspects of the finite difference method that had to be considered when implementing it in Matlab. We had to find the right values that would give the precision that we wanted and display the results properly. We used a $(N+1)\times(m+1)$ matrix to store all the results from the finite difference method. The matrix has 1 added to $N$ and $m$ because Matlab starts its index at 1 and we want to go from $0..N$ and from $0..m$ (see Appendix C.1).

### 4.3.1 Boundary Conditions

First of all, we had to find a way of adding the boundary conditions that we wanted to use for the finite differences method and still be able to change these easily, because we had to test different ones. We solved this by writing a Matlab function that enabled us to choose the function that we wanted to use as a boundary condition. We could then easily add more functions that could then be used as a boundary conditions and we could easily test these boundary conditions. The boundary conditions can also be expressed as $U_0^m$, $U_N^m$. This meant that we just had to set all the numbers at index 1 and index $N+1$ to be equal to the initial condition. This way we could also set the initial condition to be a different constant, or even an equation that changed with time.

For example, in our script (see Appendix C.1), to get a time varying boundary condition with function $\sin(2\pi x)$, we just assign the right function to the boundary variables as seen in the example code below.

```
res(1, :) = sin(2*pi*(0:1/m:1));
res(N+1, :) = sin(2*pi*(0:1/m:1));
```

### 4.3.2 Central Algorithm

The main part of the finite difference method is the central algorithm, as this is the algorithm that is derived from the approximations of the initial differential equation that we want to solve.

The central algorithm for the 1-D heat equation is calculated by using the following approximations

$$\frac{\partial^2 y(x,t)}{\partial x^2} \cong \frac{y(x+h,t) - 2y(x,t) + y(x-h,t)}{h^2}$$

Then let $x = x_j \;\rightarrow\; x \pm h = x_{j\pm 1}$

$$\begin{aligned}\frac{\partial^2 y(x_j,t_m)}{\partial x^2} &= \frac{y(x_{j+1},t_m) - 2y(x_j,t_m) + y(x_{j-1},t_m)}{h^2} \\ &= \frac{1}{h^2}\left(U_{j+1}^m - 2U_j^m + U_{j-1}^m\right) \qquad j = 1,2,3,\dots,N-1\end{aligned}$$

We then can do the same with the left hand side of the heat equation and calculate using the forward difference method

$$\frac{\partial y(x,t)}{\partial t} = \frac{y(x,t+k) - y(x,t)}{k}$$

$$\Rightarrow \frac{\partial y(x_j,t_m)}{\partial t} = \frac{y(x_j,t_{m+1}) - y(x_j,t_m)}{k}$$

$$= \frac{1}{k}\left(U_j^{m+1} + U_j^m\right) \qquad m = 0, 1, 2, ...$$

When equating both sides we get the following equation that lets us find the value for the next time period using previously calculated values.

$$U_j^{m+1} = \ vU_{j-1}^m + (1-2v)U_j^m + vU_{j+1}^m, \qquad v = \frac{k}{h^2}$$

We want to implement this algorithm for every value from $0..m$ that we set, and every value from $0..N$. This can be done by using two for loops that will iterate over the matrix, and for every $m$, to get $m+1$ we will pass the required values to the equation.

As we have a matrix with all the previous values that have been calculated, we can use it to get the values of $U_{j-1}^m$, $U_j^m$, $U_{j+1}^m$ directly from the previous column of the matrix.

### 4.3.3 Choosing Constants

We also had to choose a constant value for $h$, the step size, and $k$, which is the difference in time. We did this by setting $v = \frac{h}{k^2} = 0.25$ as $v \leqslant 0.5$ so that the finite difference method does not overshoot and give the wrong result, then set $h$ to the desired precision and calculated $k$ from that.

### 4.3.4 Plotting Results

We plotted the results on two different graphs. The first one was a 2D plot of the matrix, where every different line represents a different time. This gave a good idea of how the heat equation changed over time for constant boundary conditions. However, when making the boundary conditions a function that varies with time as well, it became unclear how the heat equation changed over time. That is why we also plotted the heat equation on a 3D plot. This made the result much more visible and one could instantly see how the function changed with time.

Another problem we faced was, that as we increased $m$, because we wanted the heat equation to run for longer so that we could see the complete effect, we started to plot too many functions on one graph and couldn't distinguish between the different lines anymore. We, therefore, added a variable called step, which skips the functions with that specific interval. This made it possible to decrease the value of $h$ by a lot, but still see the full effect of the heat equation over time.

## 4.4 Solving the Heat Equation

Using the script that we wrote in Matlab we were now able to test it for different initial conditions, while keeping the boundary conditions constantly at 0.

### 4.4.1 Tent Function

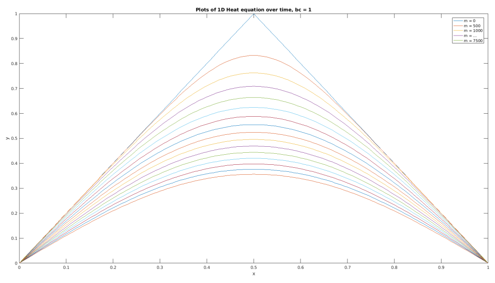

**Figure 62:** Heat equation with tent function as initial condition, using $N = 150,\ m = 7500,\ step = 500$.

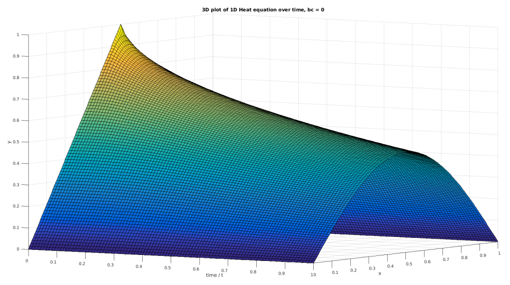

**Figure 63:** Heat equation with tent function as initial condition, using $N = 150,\ m = 5000,\ step = 50$.

As we can observe in Figure 62 and Figure 63, the tent function becomes parabolic over time and also decreases in amplitude over time. This is visualized in Figure 63 as we can clearly see how the amplitude decreases over time. This is expected from the 1-D heat equation, as over time the heat will disperse through the ends of the 1-D line.

### 4.4.2 Sinusoidal Function

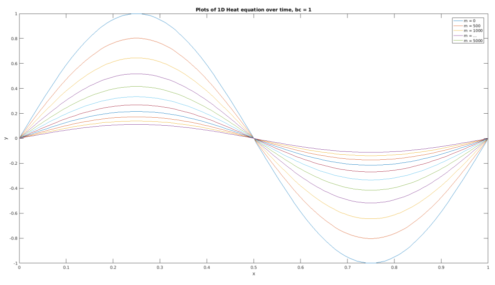

**Figure 64:** Heat equation with $y_0(x) = \sin(2\pi x)$ as initial condition, using $N = 150,\ m = 5000,\ step = 500$.

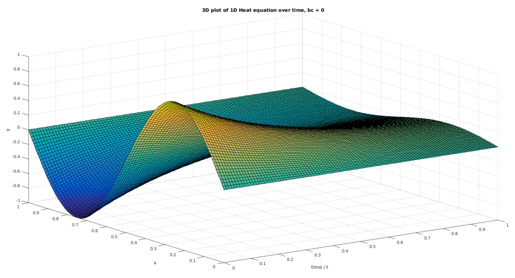

**Figure 65:** Heat equation with $y_0(x) = \sin(2\pi x)$ as initial condition, using $N = 150,\ m = 5000,\ step = 50$.

Over time, because the sine function has a maximum and a minimum, which are symmetrical to each other, it means that they will cancel each other out according to the heat equation, which can be observed in Figure 64.

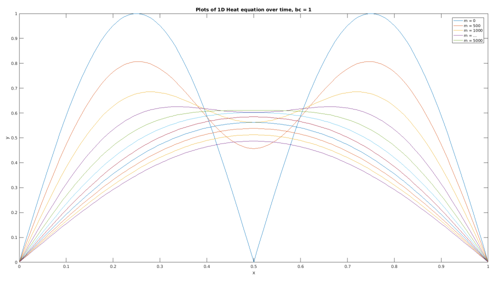

**Figure 66:** Heat equation with $y_0(x) = |\sin(2\pi x)|$ as initial condition, using $N = 150,\ m = 5000,\ step = 500$.

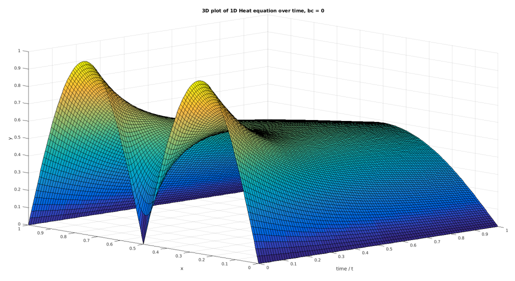

**Figure 67:** Heat equation with $y_0(x) = |\sin(2\pi x)|$ as initial condition, using $N = 150,\ m = 5000,\ step = 50$.

For $y_0(x) = |\sin(2\pi x)|$ as initial condition, the function now has two maximum points instead of a maximum and a minimum. Instead of canceling out over time, it instead produces a similar result to the tent function and produces a parabolic looking function over time. This is because, as the boundary condition are constantly 0, the heat will converge towards the center which is shown by the figures above.

### 4.4.3 Inverse Hyperbolic Sine

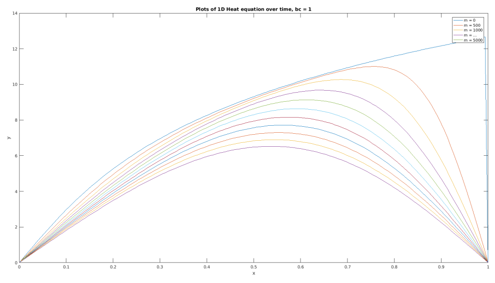

**Figure 68:** Heat equation with $y_0(x) = 5\sinh^{-1}(2\pi x)$ as initial condition, using $N = 150,\ m = 5000,\ step = 500$.

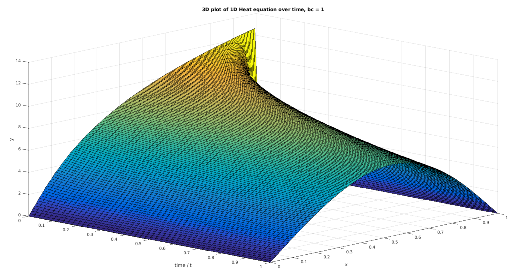

**Figure 69:** Heat equation with $y_0(x) = 5\sinh^{-1}(2\pi x)$ as initial condition, using $N = 150,\ m = 5000,\ step = 50$.

As an additional initial condition, we chose to cover $y_0(x) = 5\sinh^{-1}(2\pi x)$. We thought it would be interesting to see how the heat equation would react if the initial condition did not match up with one of the extremes of the function itself, such as in this case. As we expect with the heat equation, the heat converges towards the center over time to form a symmetrical parabolic function when the initial conditions are constantly 0.

### 4.4.4 Exponential

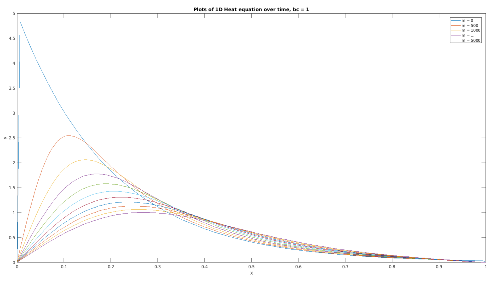

**Figure 70:** Heat equation with $y_0(x) = 5e^{-5x}$ as initial condition, using $N = 150,\ m = 5000,\ step = 500$.

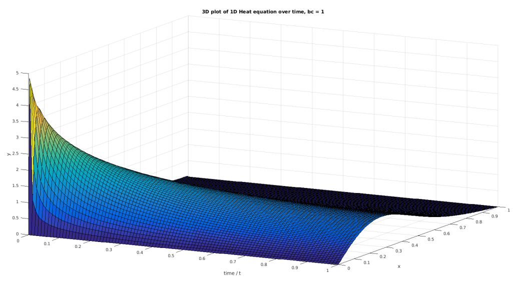

**Figure 71:** Heat equation with $y_0(x) = 5e^{-5x}$ as initial condition, using $N = 150,\ m = 5000,\ step = 50$.

We also chose to test the exponential function $y_0(x) = 5e^{-5x}$ as this function had both $x = 0,\ x = 1$ were not equal to the boundary conditions of 0, even though as $x$ increases it will slowly converge towards 0. As the boundary condition is 0, the heat equation will ensure that the peak moves towards the center and the boundaries will smooth out where $x = 0,\ x = 1$.

# 5 Bonus

In this section we continued to experiment on the 1-D heat equation and solved it with different boundary conditions and different initial conditions to see what results we can get.

## 5.1 Initial condition that does not match one or both boundary conditions

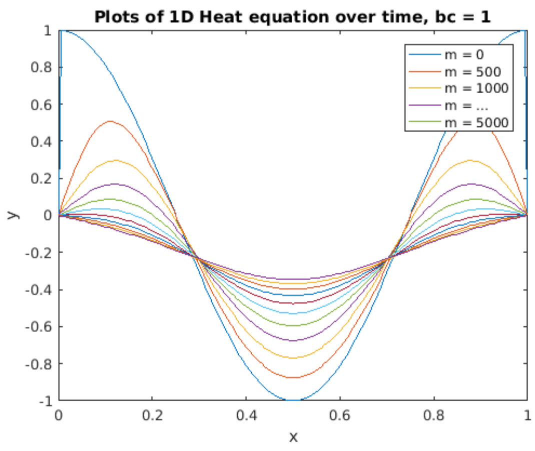

**Figure 72:** Heat equation with $y_0(x) = \cos(2\pi x)$ as initial condition, using $N = 150,\ m = 5000,\ step = 500$.

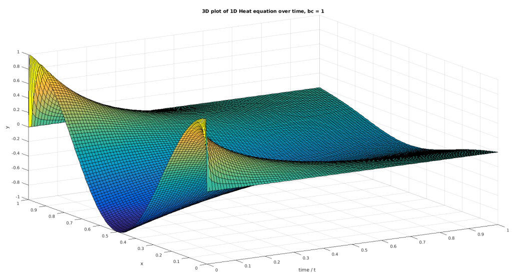

**Figure 73:** Heat equation with $y_0(x) = \cos(2\pi x)$ as initial condition, using $N = 150,\ m = 5000,\ step = 50$.

We used $y_0(x) = \cos(2\pi x)$ to see how the heat equation will run over time when the initial condition does not match both boundary conditions. As we can see, when $m = 0$, the function jumps from 0 to 1 because that is where the cosine function would normally start. However, as time goes by, the function converges towards being a straight line at $y(x) = 0$

## 5.2 Further extend by including constant, non-zero boundary conditions.

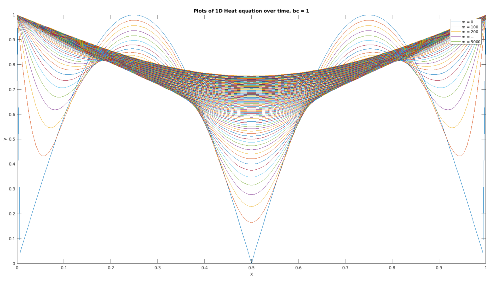

**Figure 74:** Heat equation with $y_0(x) = |\sin(2\pi x)|$ as initial condition, using $N = 150$, $m = 5000$, $step = 500$ and boundary conditions = 1.

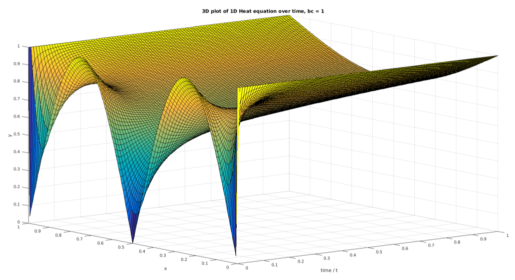

**Figure 75:** Heat equation with $y_0(x) = |\sin(2\pi x)|$ as initial condition, using $N = 150$, $m = 5000$, $step = 50$ and boundary conditions = 1.

We also tested how the heat equation would develop over time if the initial conditions did not match both boundary conditions and the boundary conditions were not constantly at 0. We set the boundary conditions to 1, and as we can see in Figure 74 and in Figure 75, instead of converging to $y = 0$, the function now converges to $y = 1$ as we let the heat equation run over time.

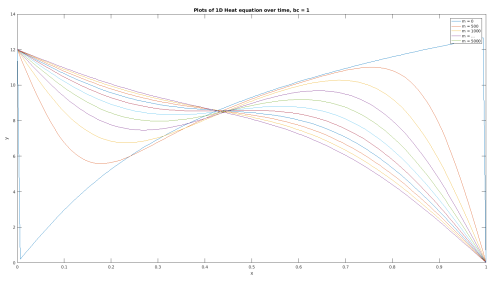

**Figure 76:** Heat equation with $y_0(x) = 5\sinh^{-1}(2\pi x)$ as initial condition, using $N = 150$, $m = 5000$, $step = 500$ and boundary conditions = 12 and 0.

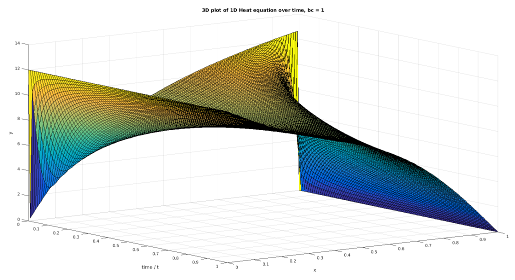

**Figure 77:** Heat equation with $y_0(x) = 5\sinh^{-1}(2\pi x)$ as initial condition, using $N = 150$, $m = 5000$, $step = 50$ and boundary conditions = 12 and 0.

For this part we also wanted to see what would happen if the boundary conditions both didn't match the initial condition and were different themselves. We predicted that using an initial condition of 12 and 0 would eventually produce a straight line with the equation $y = -\frac{x}{12}$. As we can observe above in Figures 76 and 77, the line is approaching that function as time increase to infinity.

## 5.3 Further extend to include time-varying boundary conditions.

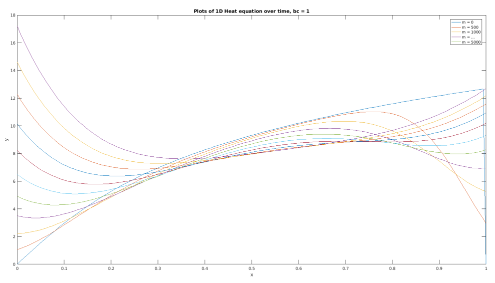

**Figure 78:** Heat equation with $y_0(x) = 5\sinh^{-1}(2\pi x)$ as initial condition, using $N = 150$, $m = 5000$, $step = 50$, where $x = 0$ boundary condition $= 10e^x$ and $x = 1$ boundary condition $= 5\sinh^{-1}(2\pi x)$

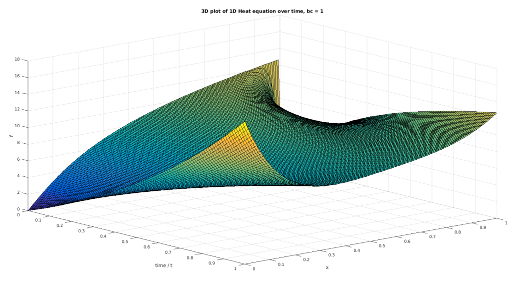

**Figure 79:** Heat equation with $y_0(x) = 5\sinh^{-1}(2\pi x)$ as initial condition, using $N = 150$, $m = 5000$, $step = 50$, where $x = 0$ boundary condition $= 10e^x$ and $x = 1$ boundary condition $= 5\sinh^{-1}(2\pi x)$

Using time-varying boundary conditions produced much more interesting plots that, however, still followed the pattern that we observed until now. For the first plot seen in Figures 78 and 79 we observed that, even though the initial boundary conditions did not match up with initial condition of $y_0(x) = 5\sinh^{-1}(2\pi x)$ it would slowly converge towards them, even though they varied with time. We used two different boundary conditions for these plots, at $x = 0$ the boundary condition was $10e^x$ and when $x = 1$ the boundary condition was $5\sinh^{-1}(2\pi x)$. These two boundary conditions are very hard to distinguish in the 2-D plot, however, one can easily see them on the left and right hand side of the 3-D plot.

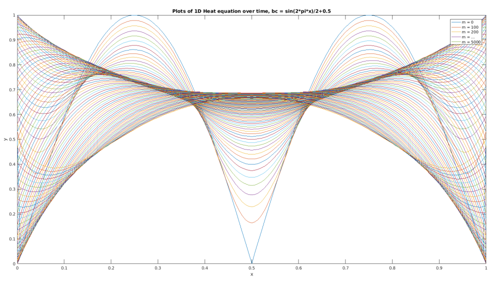

**Figure 80:** Heat equation with the inverse hyperbolic sine function as initial condition, using $N = 150$, $m = 5000$, $step = 500$.

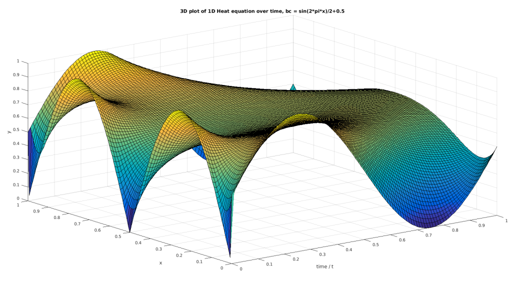

**Figure 81:** Heat equation with the inverse hyperbolic sine function as initial condition, using $N = 150$, $m = 5000$, $step = 50$.

We also plotted $y_0(x) = |\sin(2\pi x)|$ using boundary conditions of $\frac{\sin(2\pi x)+1}{2}$ to show that as time increases, even though the boundary conditions are varying at a constant frequency, the middle, where $x = 0.5$ averages out and tends to $y = 0.5$ as time goes on. This is interesting because the boundary conditions are forcing the extremities of the function to vary, but the center stays constant.

# A Appendix: RL Circuit

## A.1 Heun Method

### A.1.1 heun.m

```
function [x,y] = heun(func, xa, ya, h)

%caluclate next x by incrementing by the stepsize
x = xa + h;

gradient1 = feval(func, xa, ya); %calculate the gradient at t
ypredictor=ya+h*gradient1;        %calculate predictor for the next valu
gradient2=feval(func, x, ypredictor); %calculate the gradient at t + h
y = ya + h/2*(gradient1 + gradient2);

end
```

### A.1.2 heun_script.m

```
function heun_script (tf) %tf is the end time

%initailise the circuits
R = 0.5;
L = 0.0015;
h = 0.0001; %step size


%initailise the container

N = round(tf/h); %number of iterations
t = zeros(1, N);
Vout = zeros(1, N);
current = zeros(1,N);

%input voltage
% step function of 3.5 volt
Vin = @(t)3.5*heaviside(t);

%the initial condition
t(1) = 0;
current(1) = 0;


%the equation
func = @(t,current) (Vin(t)-R*(current))/L; %Function input for differe
```

```
%Huen
for j = 1 : N-1
    [t(j + 1),current(j + 1)] = heun(func, t(j), current(j), h);
    Vout(j + 1) = Vin(t(j)) - R*current(j); %Create Vout array from Iout
end

%plot

figure(1);
plot(Vout);
xlabel({'Time', '(seconds)'});
ylabel({'V_{out}', '(volt)'});
title('(Heaviside)␣V_{out}␣versus␣time');

figure(8);
plot(t, current);
xlabel({'Time', '(seconds)'});
ylabel({'V_{out}', '(volt)'});
title('V_{out}␣versus␣time␣(Heaviside)');

%-----------------------------------------------------------------------

%initailise the circuits information at the top

%input voltage
tau = 0.000150;
A = 3.5;

Vin = @(t) A * exp(-t.^2/tau);

%the initial condition
t(1) = 0;
current(1) = 0;


%the equation
func = @(t,current) (Vin(t)-R*(current))/L; %Function input for differe


%Huen
for j = 1 : N-1
    [t(j + 1),current(j + 1)] = heun(func, t(j), current(j), h);
    Vout(j + 1) = Vin(t(j)) - R*current(j); %Create Vout array from Iout
end
```

```
%plot

figure(2);
plot(Vout);
xlabel({'Time', '(seconds)'});
ylabel({'V_{out}', '(volt)'});
title('V_{out}␣versus␣time␣(Exponential␣square␣function␣#1)');

%----------------------------------------------------------------------

%initailise the container information at the top

%input voltage
tau = 0.000150;
A = 3.5;

Vin = @(t) A * exp(-t/tau);

%the initial condition
t(1) = 0;
current(1) = 0;


%the equation
func = @(t,current) (Vin(t)-R*(current))/L; %Function input for differe


%Huen
for j = 1 : N-1
    [t(j + 1),current(j + 1)] = heun(func, t(j), current(j), h);
    Vout(j + 1) = Vin(t(j)) - R*current(j); %Create Vout array from Iout
end

%plot

figure(3);
plot(Vout);
xlabel({'Time', '(seconds)'});
ylabel({'V_{out}', '(volt)'});
title('V_{out}␣versus␣time␣(Exponential␣#2)');


% %--------------------------------------------------------------------
```

```
%initailise the circuits information at the top


N = round(tf/h); %number of iterations
 t = zeros(1, N);
 Vout = zeros(1, N);
 current = zeros(1,N);

%input voltage
% step function of 5 volt
% T= 0.00015, 0.000015, 0.0004, 0.0011
T = 0.0011;
 Vin = @(t)4*sin(2*pi*t/T);

%the initial condition
 t(1) = 0;
 current(1) = 0;


%the equation
 func =  @(t,current) (Vin(t)-R*(current))/L; %Function input for differe


%Huen
 for j = 1 : N-1
     [t(j + 1),current(j + 1)] = heun(func, t(j), current(j), h);
     Vout(j + 1) = Vin(t(j)) - R*current(j); %Create Vout array from Iout
end

%plot

 figure(4);
 plot(Vout);
 xlabel({'Time', '(seconds)'});
 ylabel({'V_{out}', '(volt)'});
 title('V_{out}␣versus␣time␣(T=0.0011)(Sine␣wave)');

%-------------------------------------------------------------------------

%initailise the circuits information at the top

%input voltage
% step function of 5 volt
A = 4;
```

```
T = 0.0011;

Vin = @(t) A * square(2*pi*t/T);

%the initial condition
t(1) = 0;
current(1) = 0;


%the equation
func = @(t,current) (Vin(t)-R*(current))/L; %Function input for differe


%Huen
for j = 1 : N-1
    [t(j + 1),current(j + 1)] = heun(func, t(j), current(j), h);
    Vout(j + 1) = Vin(t(j)) - R*current(j); %Create Vout array from Iout
end

%plot

figure(5);
plot(Vout);
xlabel({'Time', '(seconds)'});
ylabel({'V_{out}', '(volt)'});
title('V_{out}␣versus␣time␣(T=0.0011)␣(Square␣wave)');

%-------------------------------------------------------------------------
%initailise the circuits information at the top

%input voltage
% step function of 5 volt
A = 4;
T = 0.0011;

Vin = @(t) A * sawtooth(2*pi*t/T);

%the initial condition
t(1) = 0;
current(1) = 0;


%the equation
func = @(t,current) (Vin(t)-R*(current))/L; %Function input for differe
```

```
%Huen
for j = 1 : N-1
    [t(j + 1),current(j + 1)] = heun(func, t(j), current(j), h);
    Vout(j + 1) = Vin(t(j)) - R*current(j); %Create Vout array from Iout
end

%plot

figure(6);
plot(Vout);
xlabel({'Time', '(seconds)'});
ylabel({'V_{out}', '(volt)'});
title('V_{out}␣versus␣time␣(T=0.0011)␣(Sawtooth␣wave)');

end
```

## A.2 Midpoint Method

### A.2.1 midpoint.m

```
function [ x, y ] = midpoint( f, t0, tfinal, y0, h)

%calculate number of steps
N = round((tfinal - t0) / h);
%initialise the array
ya = zeros(1,N);
ta = zeros(1,N);
ya(1) = y0;
ta(1) = t0;

for i = 1 : N - 1
    ta(i+1) = ta(i) + h;
    halfstep = ta(i) + 1 * h / 2;
    gradient1 = f(ta(i), ya(i));
    ypredict = ya(i) + 0.5 * h * gradient1;
    gradient2 = f(halfstep, ypredict);
    ya(i+1) = ya(i) + h * gradient2;
end
x = ta;
y = ya;
```

### A.2.2 midpoint_script.m

```
clear;
ts = 0;      % set initial value of x_0
is = 0;
h = 0.0001;  % set step-size
tf = 0.03;      % stop here

R = 0.5;
L = 0.0015;

vin = @(t) 3.5;
func = @(t, iout) (vin(t) - iout*R) / L;      % define func
[t, iout ] = midpoint(func, ts, tf, is, h);

vout = vin(t) - iout * R;
figure(1);
plot(t,vout);
xlabel({'Time', '(seconds)'});
ylabel({'V_{out}', '(volt)'});
title('V_{out}␣versus␣time␣(Heaviside)');
%-------------------------------------------------------------------
h = 0.0001;
tf = 0.03;
figure;
A = 3.5;
tau = 0.00015;

vin = @(t) A * exp(-t.^2/tau);
func = @(t, iout) (vin(t) - iout*R) / L;      % define func
[t, iout ] = midpoint(func, ts, tf, is, h);

vout = vin(t) - iout * R;
figure(2);
plot(t,vout);
xlabel({'Time', '(seconds)'});
ylabel({'V_{out}', '(volt)'});
title('V_{out}␣versus␣time␣(Exponential␣square␣function␣#1)');

%-------------------------------------------------------------------
h = 0.0001;
tf = 0.03;
figure;
A = 3.5;
tau = 0.00015;
```

```
vin = @(t) A * exp(-t/tau);
func = @(t, iout) (vin(t) - iout*R) / L;      % define func
[t, iout ] = midpoint(func, ts, tf, is, h);

vout = vin(t) - iout * R;
figure(3);
plot(t,vout);
xlabel({'Time', '(seconds)'});
ylabel({'V_{out}', '(volt)'});
title('V_{out}␣versus␣time␣(Exponential␣function␣#2)');

%-------------------------------------------------------------------
h = 0.0001;
tf = 0.03;
figure;
A = 4;
T = 0.0011;

vin = @(t) A * sin(2*pi*t/T);
func = @(t, iout) (vin(t) - iout*R) / L;      % define func
[t, iout ] = midpoint(func, ts, tf, is, h);

vout = vin(t) - iout * R;
figure(4);
plot(t,vout);
xlabel({'Time', '(seconds)'});
ylabel({'V_{out}', '(volt)'});
title('V_{out}␣versus␣time␣(T=0.0011)␣(Sine␣wave)');
%-------------------------------------------------------------------
h = 0.0001;
tf = 0.03;
A = 4;
T = 0.0011;

vin = @(t) A * square(2*pi*t/T);
func = @(t, iout) (vin(t) - iout*R) / L;      % define func
[t, iout ] = midpoint(func, ts, tf, is, h);

vout = vin(t) - iout * R;
figure(5);
plot(t,vout);
xlabel({'Time', '(seconds)'});
ylabel({'V_{out}', '(volt)'});
title('V_{out}␣versus␣time␣(T=0.0011)␣(Square␣wave)');
%-------------------------------------------------------------------
```

```
h = 0.0001;
tf = 0.03;
A = 4;
T = 0.0011;

vin = @(t) A * sawtooth(2*pi*t/T);
func = @(t, iout) (vin(t) - iout*R) / L;      % define func
[t, iout ] = midpoint(func, ts, tf, is, h);

vout = vin(t) - iout * R;
figure(6);
plot(t,vout);
xlabel({'Time', '(seconds)'});
ylabel({'V_{out}', '(volt)'});
title('V_{out}␣versus␣time␣(T=0.0011)␣(Sawtooth␣wave)');
```

## A.3 Ralston Method

### A.3.1 ralston.m

```
function [xa,ya] = ralston(func, t0, tf,i0 , h)
%calculate number of steps
N = round((tf) / h);
%initialise the array
xa = zeros(1,N);
ya = zeros(1,N);
xa(1)=t0;
ya(1)=i0;

%param
a1=1/3;
a2=2/3;
p1=3/4;
q11=3/4;

for i=1: N - 1
    xtemp=xa(i);
    ytemp=ya(i);

k1=func(xtemp,ytemp);
k2=func(xtemp+p1*h,ytemp+q11*k1*h);

ya(i+1)=ytemp+(a1*k1+a2*k2)*h;
xa(i+1)=xtemp+h;
end
```

### A.3.2 ralston_script.m

```
clear;
ts = 0;      % set initial value of x_0
is = 0;
h = 0.0001;  % set step-size
tf = 0.03;      % stop here
R = 0.5;
L = 0.0015;

% vin = @(t) 3.5;
% func = @(t, iout) (vin(t) - iout*R) / L;      % define func
% [t, iout ] = ralston(func, ts, tf, is, h);
%
% vout = vin(t) - iout * R;
% plot(t,vout);
% xlabel({'Time', '(seconds)'});
% ylabel({'V_{out}', '(volt)'});
% title('V_{out} versus time (original function)');
%%------------------------------------------------------------
% h = 0.0001;
% tf = 0.03;
% figure;
% A = 3.5;
% tau = 0.00015;
%
% vin = @(t) A * exp(-t.^2/tau);
% func = @(t, iout) (vin(t) - iout*R) / L;      % define func
% [t, iout ] = ralston(func, ts, tf, is, h);
%
% vout = vin(t) - iout * R;
% plot(t,vout);
% xlabel({'Time', '(seconds)'});
% ylabel({'V_{out}', '(volt)'});
% title('V_{out} versus time (exponential square funtion)');
%%------------------------------------------------------------
% h = 0.0001;
% tf = 0.03;
% figure;
% A = 3.5;
% tau = 0.00015;
%
% vin = @(t) A * exp(-t/tau);
% func = @(t, iout) (vin(t) - iout*R) / L;      % define func
% [t, iout ] = ralston(func, ts, tf, is, h);
%
```

```
% vout = vin(t) - iout * R;
% plot(t,vout);
% xlabel({'Time', '(seconds)'});
% ylabel({'V_{out}', '(volt)'});
% title('V_{out} versus time (exponential function)');

%-----------------------------------------------------------------
h = 0.0001;
tf = 0.03;
figure;
A = 4;
T = 0.000015;

vin = @(t) A * sin(2*pi*t/T);
func = @(t, iout) (vin(t) - iout*R) / L;      % define func
[t, iout ] = ralston(func, ts, tf, is, h);

vout = vin(t) - iout * R;
plot(t,vout);
xlabel({'Time', '(seconds)'});
ylabel({'V_{out}', '(volt)'});
title('V_{out}␣versus␣time␣(T=0.000015)␣(sine␣function)');
%%% --------------------------------------------------------------
% h = 0.0001;
% tf = 0.03;
% figure;
% A = 4;
% T = 0.0011;
%
% vin = @(t) A * square(2*pi*t/T);
% func = @(t, iout) (vin(t) - iout*R) / L;      % define func
% [t, iout ] = ralston(func, ts, tf, is, h);
%
% vout = vin(t) - iout * R;
% plot(t,vout);
% xlabel({'Time', '(seconds)'});
% ylabel({'V_{out}', '(volt)'});
% title('V_{out} versus time (T=0.0011) (square function)');
%%----------------------------------------------------------------
% h = 0.0001;
% tf = 0.03;
% figure;
% A = 4;
% T = 0.000015;
%
```

```
% vin = @(t) A * sawtooth(2*pi*t/T);
% func = @(t, iout) (vin(t) - iout*R) / L;      % define func
% [t, iout ] = ralston(func, ts, tf, is, h);
%
% vout = vin(t) - iout * R;
% plot(t,vout);
% xlabel({'Time', '(seconds)'});
% ylabel({'V_{out}', '(volt)'});
% title('V_{out} versus time (T=0.0011)(sawtooth function)');
```

## A.4 Error Analysis

### A.4.1 error_script.m

```
clear;
ts = 0;      % set initial value of x_0
is = 0;
tf = 0.03;      % stop here
R = 0.5;
L = 0.0015;

A = 6;
T = 0.00015;

vin = @(t) A * cos(2*pi*t/T);
func = @(t, iout) (vin(t) - iout*R) / L;      % define func

figure(1);

for ind=16:25
        % This increments the time step
    %by 2^-n every time where n = 16, 17, .... 25
    %we started from 16 as we observed
    %the results with larger step sizes
    %gave irregular data

    h1=2^(-ind); % set stepsize

    %obtaining the exact solution with favorite method
    [t, i_Exact] = exact_solution(R,L,T,tf,h1);
    exact = vin(t) - R*i_Exact;

    %call appropriate method
    %[t, iout] = heun_methodforerror(func, ts, tf, is, h1);
    %[t, iout] = ralston(func, ts, tf, is, h1);
    [t, iout] = midpoint(func, ts, tf, is, h1);
    vout = vin(t) - R*iout;

    %error as a function of t
    error1 = exact - vout;
    error_final = max(abs(error1));

    %this plots a log lgo plot to show the order of the error
    plot(log(h1),log(error_final), '*g'); % log/log plot stepsize vs err
    xlabel({'Step size', '(h)'});
    ylabel({'Error', '(volts)' });
    title('RALSTON ERROR (LOGLOG PLOT) ');
    hold on;
end

%this gives us the error function for the above method
figure(2);
    plot(error1);
    xlabel({'Time', '(sec)'});
    ylabel({'Error', '(volts)'});
    title('RALSTON ERROR');
```

### A.4.2 exact_solution.m

```
%this is our exact solution using our favorite method into a function
function [ t, i_Exact ] = exact_solution(R,L,T,tfinal, h)

%initial arrays
N = round((tfinal) / h);
t = zeros(1,N);
i_Exact = zeros(1,N);

%for loop calculating next exact current
for i = 1 : N - 1
    t(i+1) = t(i) + h;
    i_Exact(i+1) = (6/L)*((2*pi)/T * sin((2*pi)/T*t(i)) + (R/L)*cos((2*p
end
```

# B Appendix: RLC Circuit

## B.1 RK4Second.m

```
function [xnext, ynext] = RK4second(t, x, y, h, f1, f2)
    %source from http://www.mymathlib.com/diffeq/runge-kutta/runge_kutta
    ky1 = f2(t, x, y);
    ky2 = f2(t, x + h/3, y + h*ky1/3);
    ky3 = f2(t, x + 2*h/3, y - h*ky1/3 + h*ky2);
    ky4 = f2(t, x + h, y + h*ky1 - h*ky2 + h*ky3);
    %yi+1 = yi + 1/8 ( k1 + 3 k2 + 3 k3 + k4 )
    %xi = x0 + i h
    xnext = x + h*f1(t, x, y);
    ynext = y + h/8*(ky1 + 3*ky2 + 3*ky3 + ky4);
end
```

## B.2 RLC_script.m

```
function RLC_Script (tf)

%initialise the circuits
R = 250;
L = 650*10^(-3);
C = 3*10^(-6);
h = 0.00001; %step size

%initialise the container

N = round(tf/h); %number of iterations
qc = zeros(1, N); %x
qc_dash = zeros(1, N); %y
t = zeros(1, N);
Vout = zeros(1, N);

%input voltage
%%step function of 5 volt
%Vin = @(t)5*heaviside(t);

%%Impulse with Exponential Decay
Tau = 3*(10^-6);
Vin = @(t)5*exp(-(t.^2)/Tau);

%%Square Wave (5Hz, 100Hz, 500Hz)
%Vin = @(t)5*square(2*pi*5*t);
```

```
%%Sine Wave (5Hz, 100Hz, 500Hz)
%Vin = @(t)5*sin(2*pi*5*t);

%the coupled equation
func1 = @(t, qc, qc_dash)qc_dash;
func2 = @(t, qc, qc_dash)(Vin(t) - qc/C - R*qc_dash)/L;


%the initial condition
qc_dash(1) = 0;
qc(1) = 500*(10.^-9);
t(1) = 0;

%Runge Kutta
for i = 1 : N - 1
    t(i + 1) = t(i) + h;
    [qc(i + 1), qc_dash(i + 1)] = RK4second(t(i), qc(i), qc_dash(i), h,
    Vout(i) = R*qc_dash(i);
end

%Plot the input function
plot(t, Vin(t));

%Plot the output of the system
figure
plot(t, Vout);
xlabel('Time');
ylabel('Amplitude');
end
```

# C Appendix: Finite Differences for PDE

## C.1 finite_script.m

```
% This script implements the finite difference method to solve the heat
% equation

clear;

steps = 100;

% Set the number of samples to take
N = 150;

% Set the total time to run
m = 5000;

% Declare final size of matrix for speed
res = zeros(N+1, m+1);

% Set v
v = 0.25;

% Get h and k
h = 1/N;
k = h^2 * v;

% Set Initial condition
res(:, 1) = get_function(N, h, 3)';

% Set boundary conditions
res(1, :) = zeros(1, m+1);
res(N+1, :) = zeros(1, m+1);

% Calculate M+1 and plot it continuously
for c = 1:m
    for i = 2:N
        res(i, c+1) = v * res(i-1, c) + (1-2*v) * res(i, c) + v * res(i+
    end
end

% Plot 2D
figure;
Z = zeros(N+1, m/steps+1);
count = 1;
```

```
for i = 1:m+1
    if rem(i, steps) == 1
        plot(0:h:1, res(:, i));
        Z(:, count) = res(:, i);
        hold on;
        count = count + 1;
    end
end
hold off;

xlabel('x');
ylabel('y');
title('Plots␣of␣1D␣Heat␣equation␣over␣time,␣bc␣=␣1');
legend('m␣=␣0', 'm␣=␣100', 'm␣=␣200', 'm␣=␣...', 'm␣=␣5000');

% Plot 3D
figure;
x = (0:m/steps) / (m/steps);
y = (0:N) / N;
[X,Y] = meshgrid(x, y);
surf(X, Y, Z);

xlabel('time␣/␣t');
ylabel('x');
zlabel('y');
title('3D␣plot␣of␣1D␣Heat␣equation␣over␣time,␣bc␣=␣1');
```

## C.2 get_function.m

```
function y = get_function(N, h, f)

y = 0:N;

if f == 1
    count = 1;
    for i = 0:h:1
        if i < 0.5
            y(count) = 2*i;
        else
            y(count) = 2-2*i;
        end
        count = count + 1;
    end
elseif f == 2
    y = sin(2*pi*(0:h:1));
elseif f == 3
    y = abs(sin(2*pi*(0:h:1)));
elseif f == 4
    %add func
    y = 5*asinh(2*pi*(0:h:1));
elseif f == 5
    %add
    y = 5*exp(-5*(0:h:1)); % exponential
end

end
```