Numerical Analysis of ODEs using Matlab Coursework 2018

Group 20

Al-Aqqad, Zed CID: 01202742
Pietreanu, Andrei CID: 01200252
Tang, Tze Hei CID: 01221240
Vijayaraghavan, Suparnan CID: 01180674
Lawrence, Michael CID: 01181925
Rallis, Vasilios CID: 1229190

Contents

1. RL Circuit

Exercise 1

In the first exercise we observed the properties of a RC circuit which acts like as a low pass filter cutting all the frequencies above 1.592kHz.

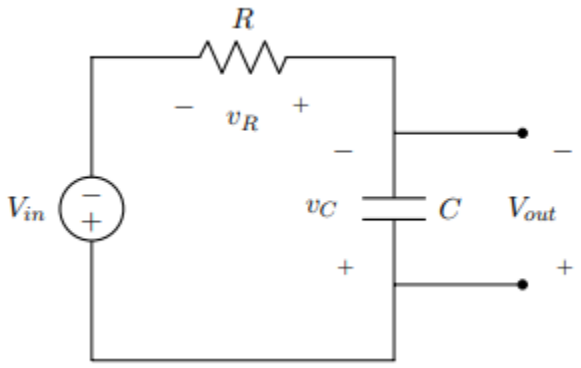

Figure 1 RC circuit

The RC circuit above can be described by the following equation:

$$v_C(t) + v_R(t) = V_{in}(t)$$

$$\frac{1}{C}\int_0^t i_C(t) + Ri_C(t) = V_{in}(t)$$

$$\frac{1}{C}q_C(t) + R_{\dot{q}C}(t) = V_{in}(t)$$

Where $R = 1k\Omega;\ C = 100nF;\ q_C(0) = 500nC;\ input = V_{in};\ V_{out} = \frac{1}{C}q_C(t)$

Second-order Runge-Kutta methods have 2 components: The Predictor Equation and the Corrector Equation, given by:

$y_{i+1}^p = y_i + hf(x_i, y_i)$ - Predictor equation (Euler's Method)

$y_{i+1} = y_i + h(\frac{f(x_i,y_i)+f(x_{i+1},y_{i+1}^p)}{2})$ - Corrector equation

, where h is the step interval.

It can be further simplified by implementing an increment function that can be defined as ϕ, where:

$$\phi = a_1k_1 + a_2k_2 + \cdots a_nk_n$$

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

$k_2 = f(x_i + p_1h, y_i + q_{11}k_1h)$Etc.

So the second-order Runge-Kutta methods can be written as:

$$y_{i+1} = y_i + h\phi(x_i, y_i, h)$$

The constants $a_1, a_2, p_1 \; and \; q_{11}$make the difference between Heun's, midpoint and Ralston's methods.

$$Heun's\ Method: a_1 = 0.5, a_2 = 0.5, p_1 = 1, q_1 = 1$$

$$Midpoint\ Method: a_1 = 0, a_2 = 1, p_1 = 0.5, q_1 = 0.5$$

$$Ralston's\ Method: a_1 = \frac{1}{3}, a_2 = \frac{2}{3}, p_1 = \frac{3}{4}, q_1 = \frac{3}{4}$$

We modeled the behavior of the RC-circuit using three second-order Runge-Kutta methods: Heun's method, the midpoint method and Ralston's method. We had to write 2 scripts for this part:

- **RK2.m** in which we defined a function that implements the three methods. Due to their similar forms we were able to easily switch between the three just by changing the constant
- **RK2-script.m** is a script that contains calls to the function **RK2.m**, for the following cases:
  1. Step signal with amplitude $\bar{V}_{in} = 2.5V$
  2. Impulse signal and decay

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

$$V\_in = \overline{V_{in}} exp\left\{-\frac{t}{\tau}\right\}$$

  Where $\bar{V}_{in} = 2.5V \; and \; \tau = 100(\mu s)^2, \tau = 100\mu s$ respectively.

  3. Sine, square and sawtooth waves with amplitude $Vin = 5\ V$ and the periods: $T = \{100\ \mu s, 10\ \mu s, 500\ \mu s, 1000\ \mu s\}$

The following code was used to generate graphs for each of the cases mentioned above:

RK2.m:

```
function [x,y] = RK2(dy,x0,y0,h,xf)
N = round((xf - x0) / h); %nr of steps: (interval size)/(step size)
x = x0:h:N*h;
y = y0;
%%Heun
a1 = 0.5; a2 = 0.5; p1 = 1; q1 = 1;
%%Midpoint
```

```
%a1 = 0; a2 = 1; p1 = 0.5; q1 = 0.5;

%%Ralston

%a1 = 1/3; a2 = 2/3; p1 = 3/4; q1 = 3/4;

k1 = dy;

k2 = @(x,y) dy(x + p1*h, y + q1*k1(x,y)*h);

fi = @(x,y) a1*k1(x,y) + a2*k2(x,y);

for i=1:N

    y(i+1) = y(i) + h*fi(x(i),y(i));

end
```

RK2-script.m:

```
%Choose step size, initial condition, final value
x0 = 0;
y0 = 500e-9;
h = 1e-6;
xf = 1e-3;

%Choose Vin
Vin = @(x) 2.5;

%Choose circuit constants
R = 1e+3;
C = 100e-9;

dy = @(x,y) Vin(x)/R - y/(C*R);

%Let ye = qc(t)
[x,ye] = RK2(dy,x0,y0,h,xf);

%Generate the arrays for Vin(t) and Vout(t)
vout = ye/C;
vin = arrayfun(@(x) Vin(x), x);

%Plot Vin(t) and Vout(t)
plot(x, vout, 'r');
hold on;
plot(x, vin, '--b');

xlabel('Time(s)');
ylabel('Voltage(V) Vin(t) in blue/Vout(t) in red');

ylim([min([min(vin) min(ye/C)]) max([max(vin) max(ye/C)])]);
xlim([0 xf]);
```

1. Step signal with amplitude 2.5V

As the input signal is a DC voltage the capacitor blocks the current and the voltage across it is equal to the one supplied by the source (Vout = 2.5V).

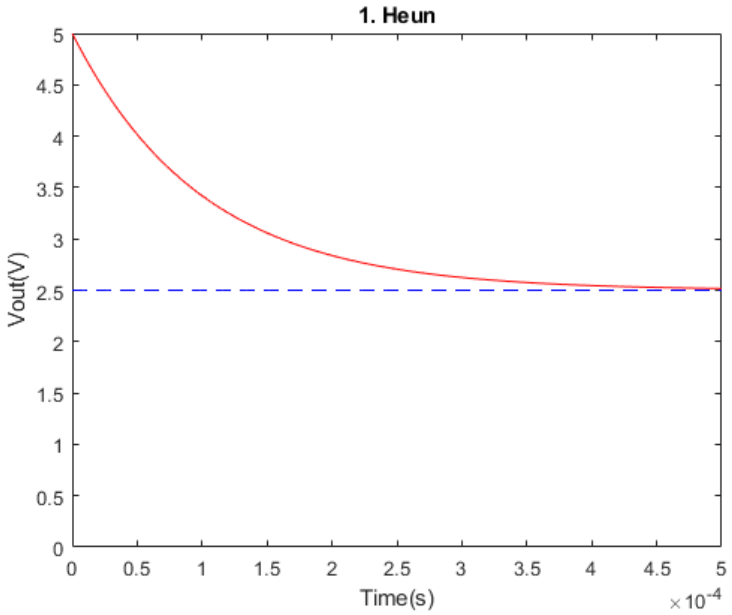

Figure 2) 1. Heun Input=2.5V

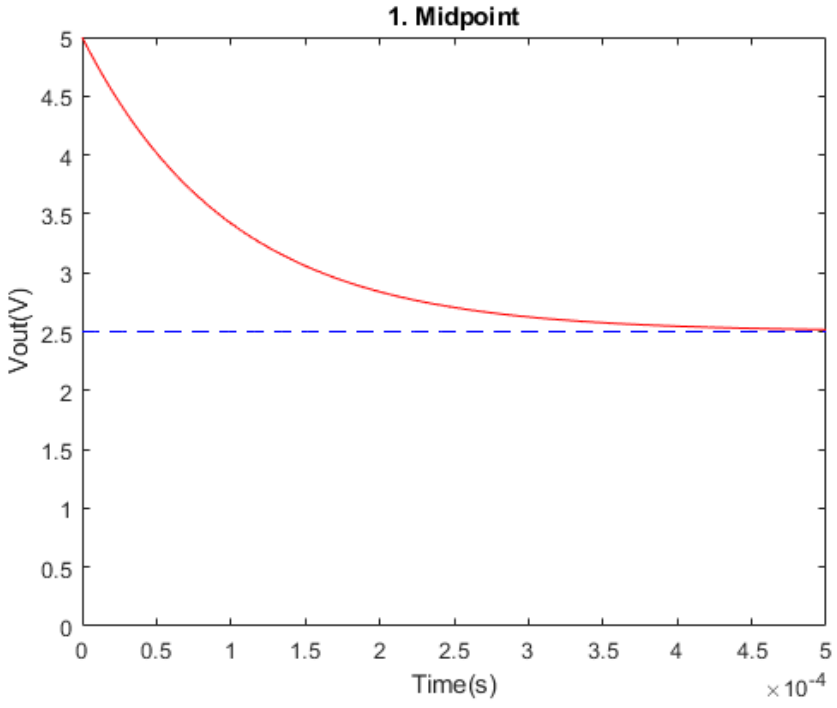

Figure 3) 1. Midpoint Input=2.5V

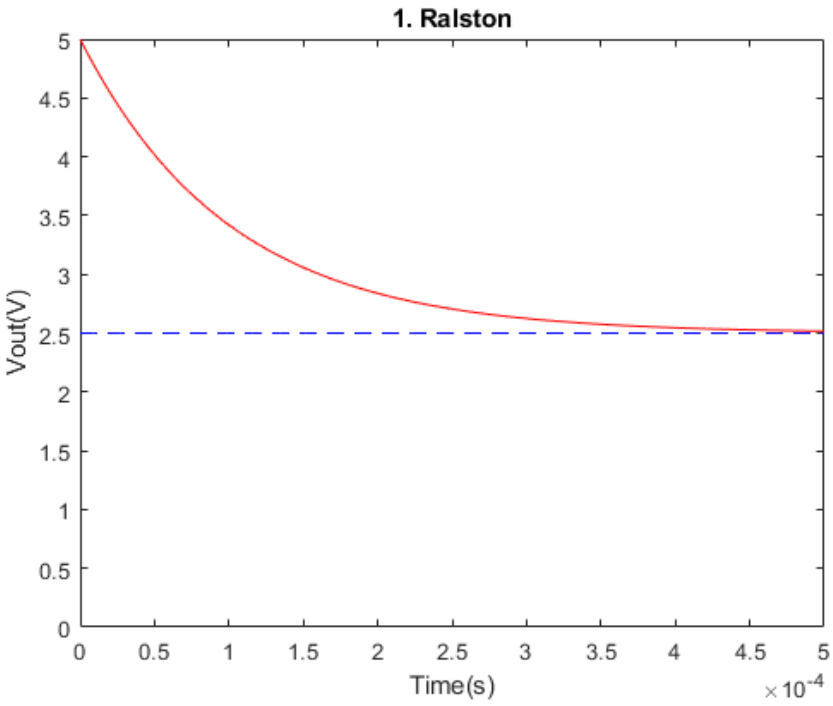

Figure 4) 1. Ralston Input=2.5V

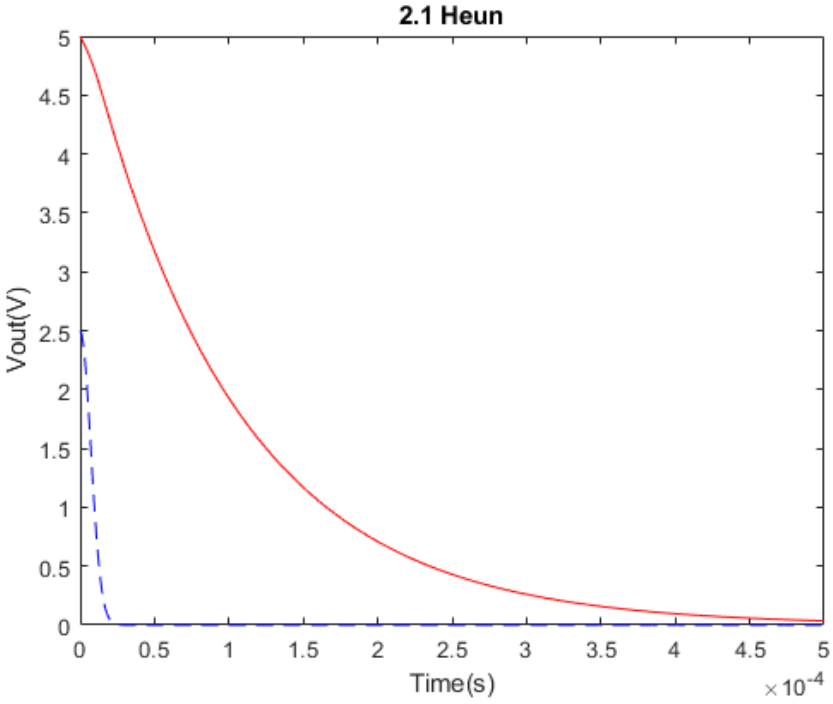

Figure 5) 2.1 Heun

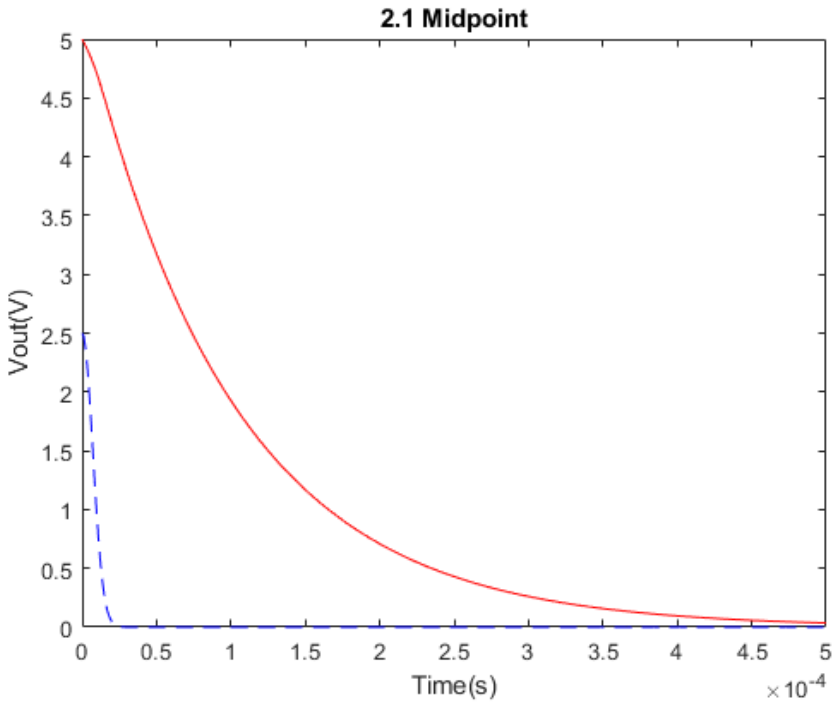

Figure 6) 2.1 Midpoint

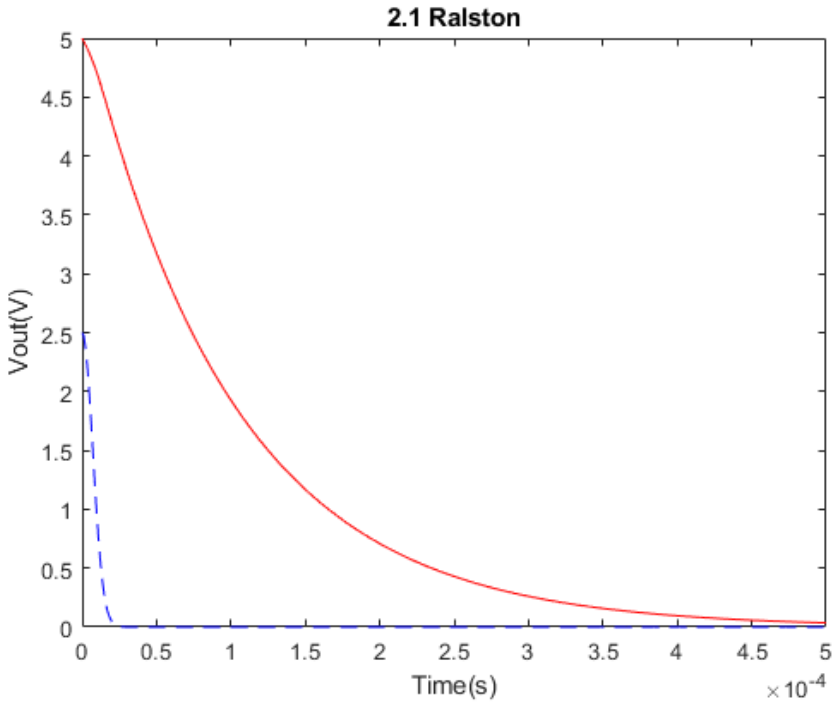

Figure 7) 2.1 Ralston

2.2

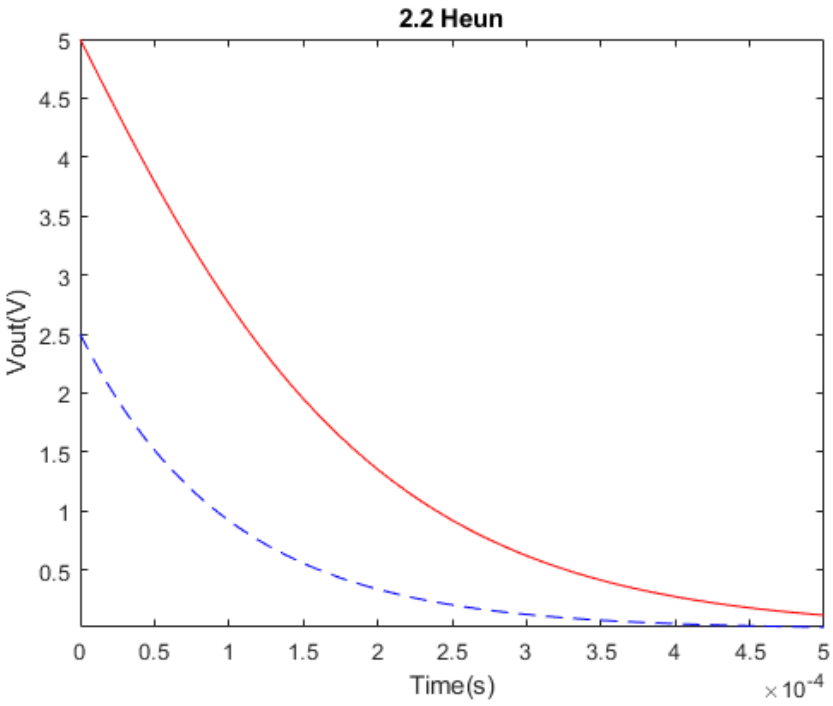

Figure 8) 2.2 Heun

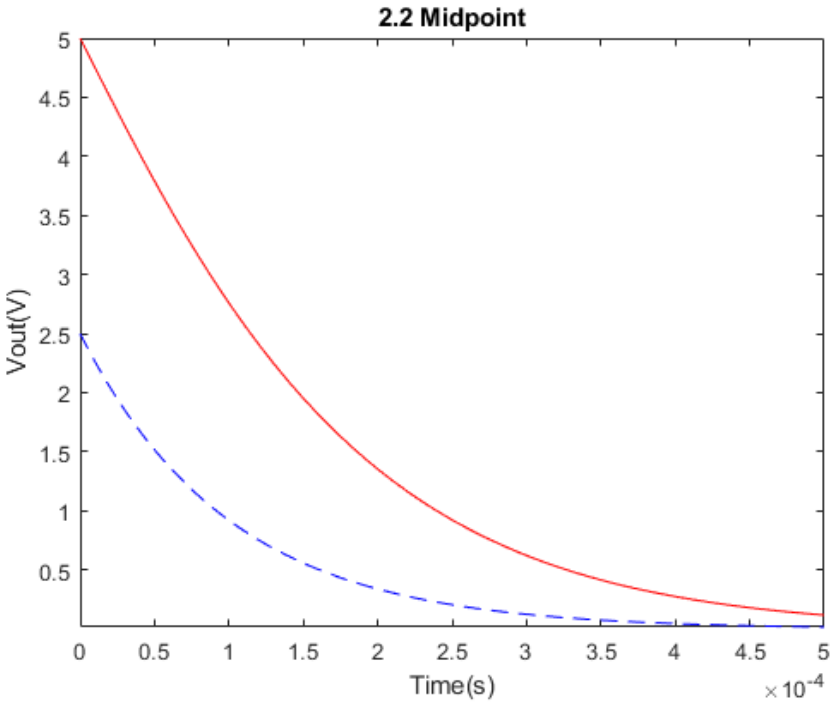

Figure 9) 2.2 Midpoint

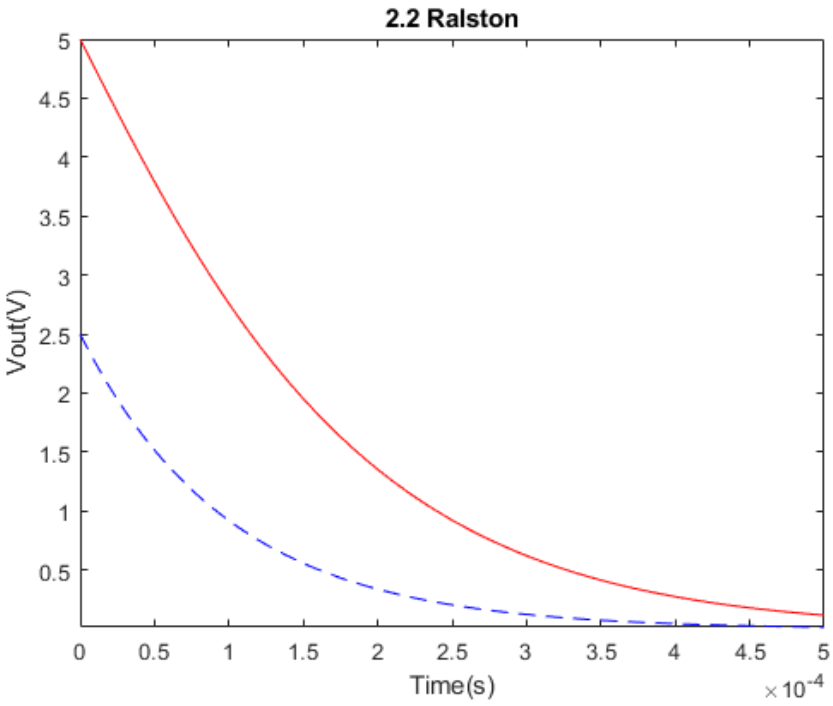

Figure 10) 2.2 Ralston

After comparing the outputs obtained by applying the two inputs we can state that the first set of graphs is "more abrupt" as the signal settles down to zero faster due to the presence of the squared term(t^{-2}).

3.1.1 Sine $T = 100\ \mu s$

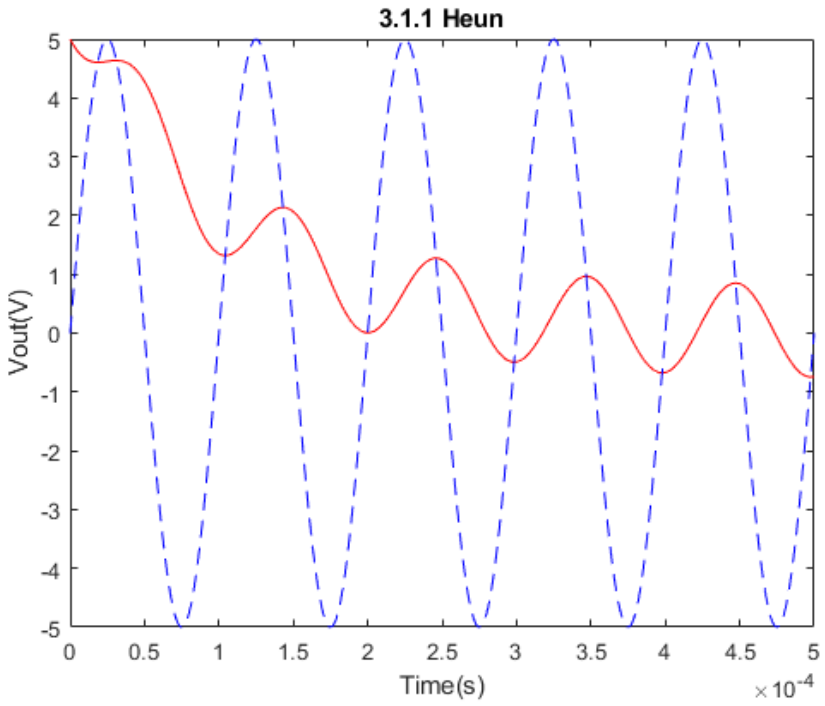

Figure 11) 3.1.1 Heun, Sine wave, $T = 10^{-4}s$

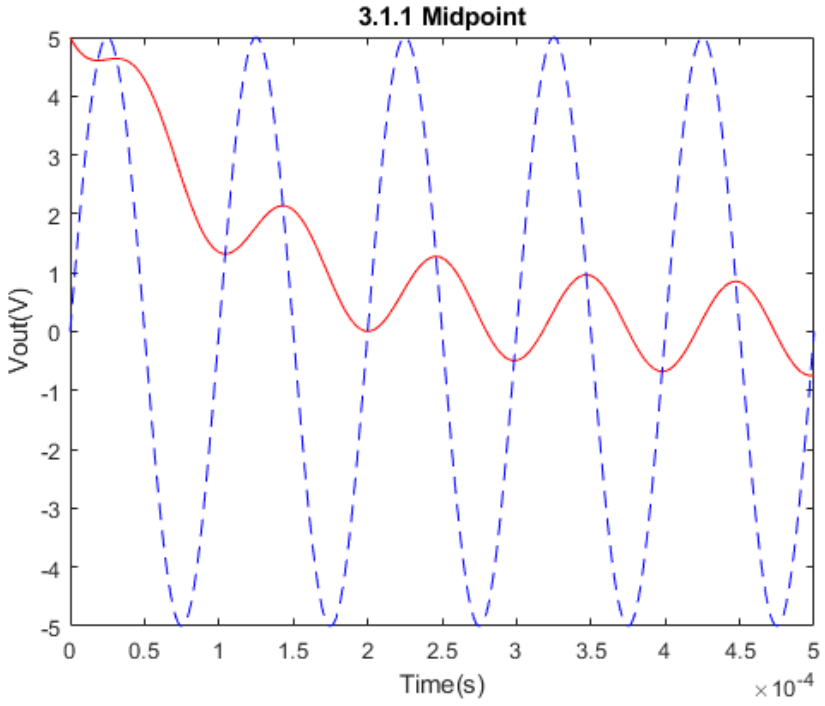

Figure 12) 3.1.1 Midpoint, Sine wave, $T = 10^{-4}s$

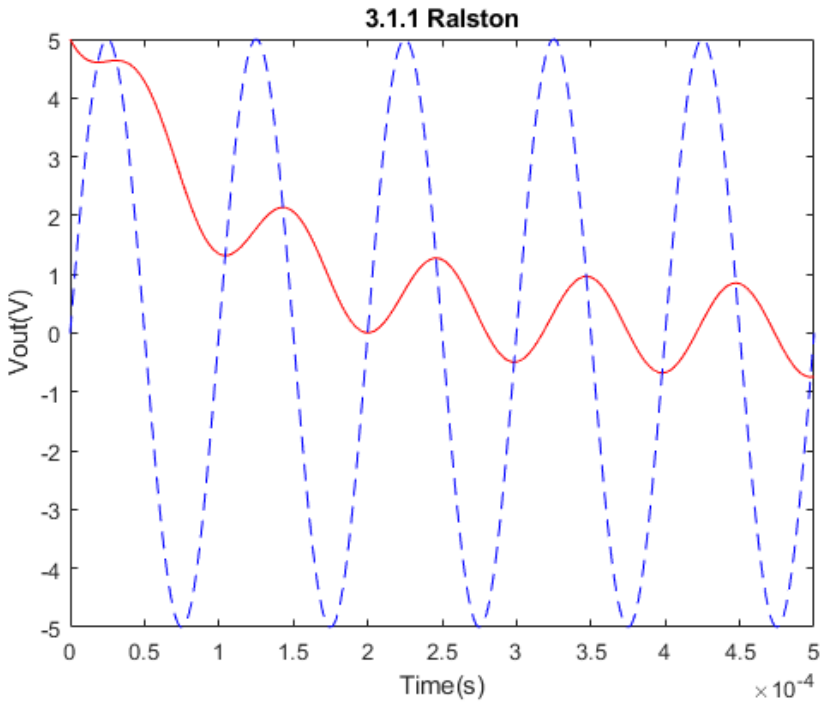

Figure 13) 3.1.1 Ralston, Sine wave, $T = 10^{-4}s$

3.1.2 Square $T = 100\ \mu s$

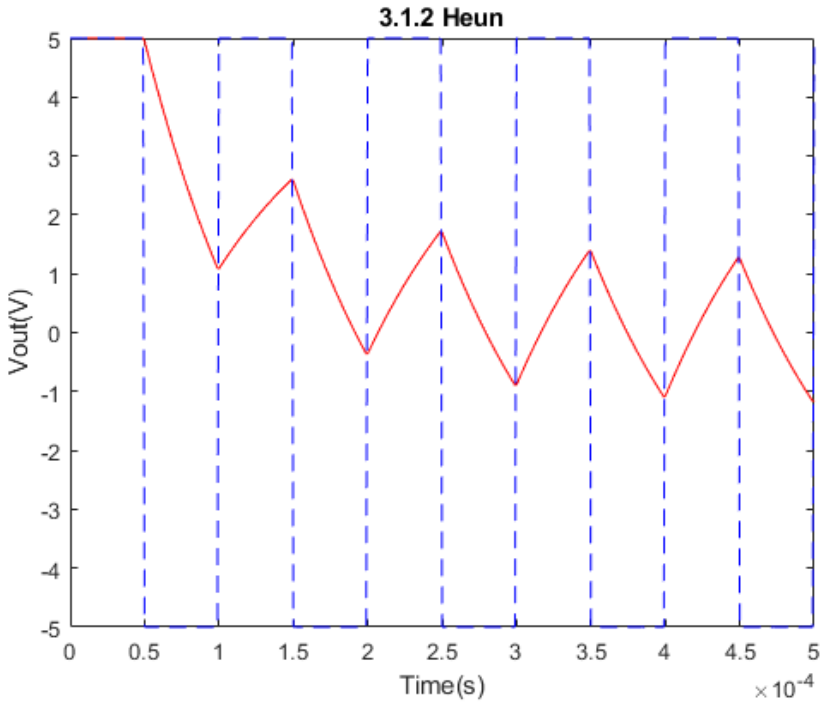

Figure 14) 3.1.2 Heun, Square wave, $T = 10^{-4}s$

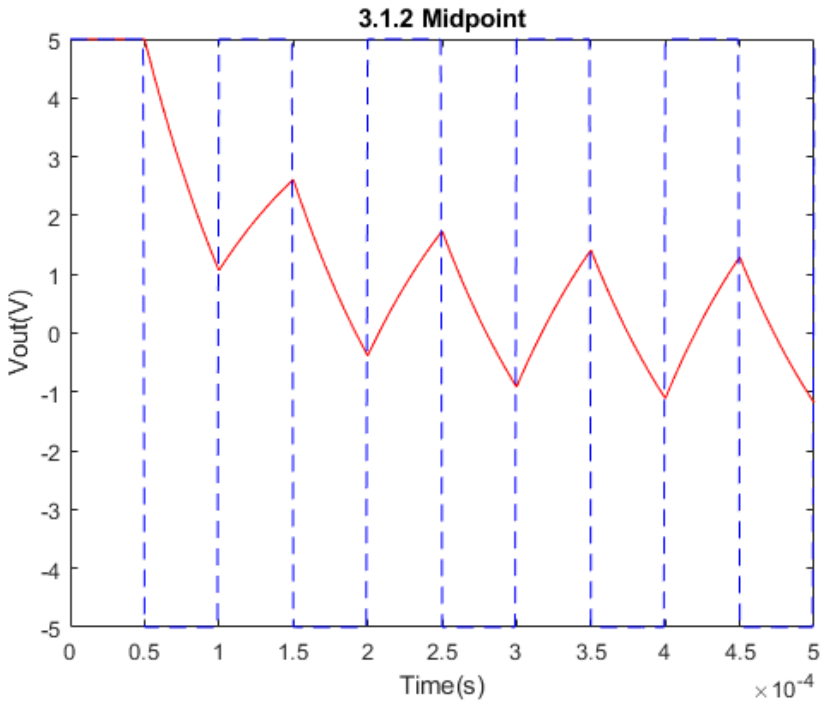

Figure 15) 3.1.2 Midpoint, Square wave, $T = 10^{-4}s$

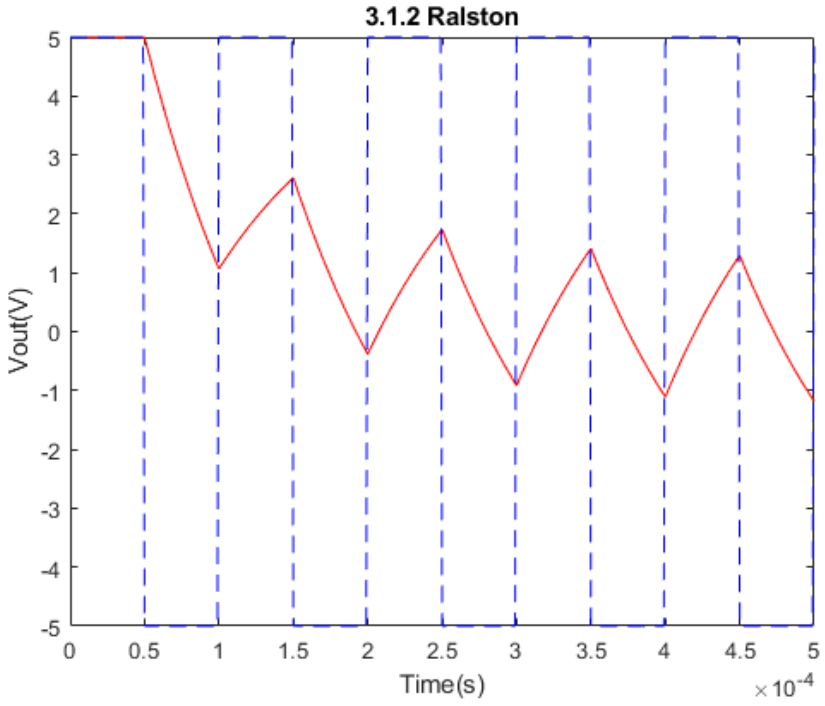

Figure 16) 3.1.2 Ralston, Square wave, $T = 10^{-4}s$

3.1.3 Sawtooth $T = 100\ \mu s$

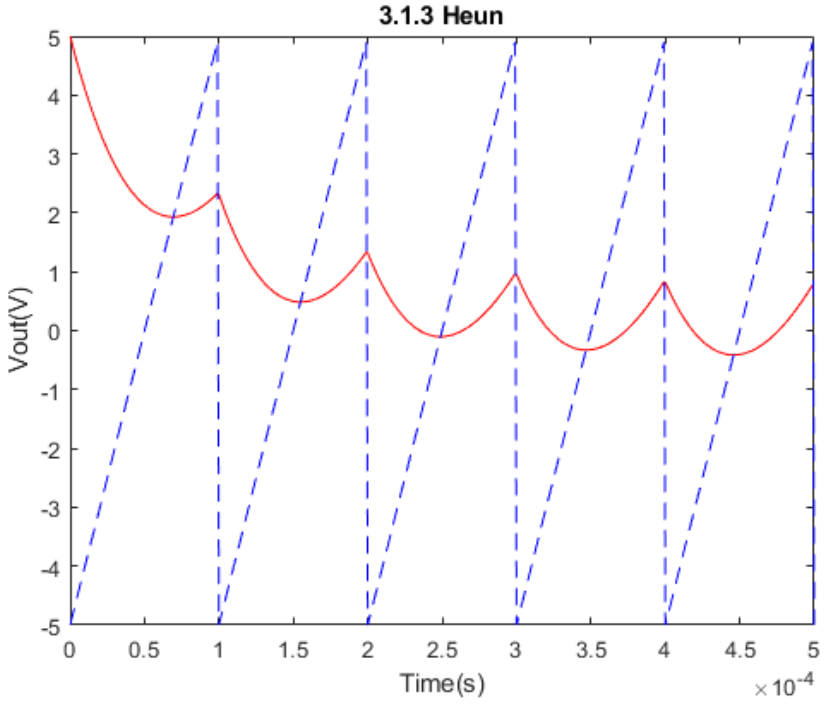

Figure 17) 3.1.3 Heun, Sawtooth wave, $T = 10^{-4}s$

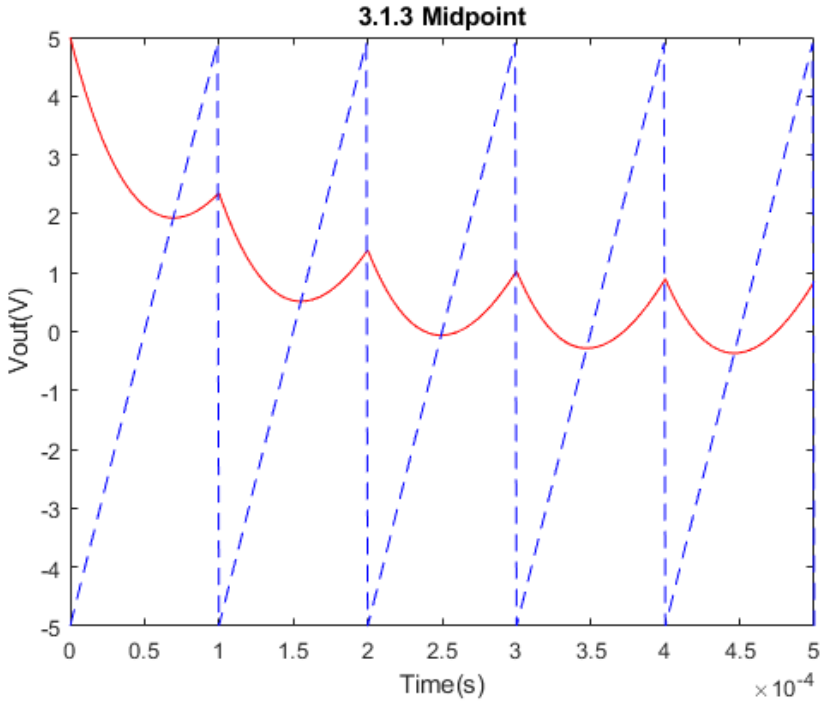

Figure 18) 3.1.3 Midpoint, Sawtooth wave, $T = 10^{-4}s$

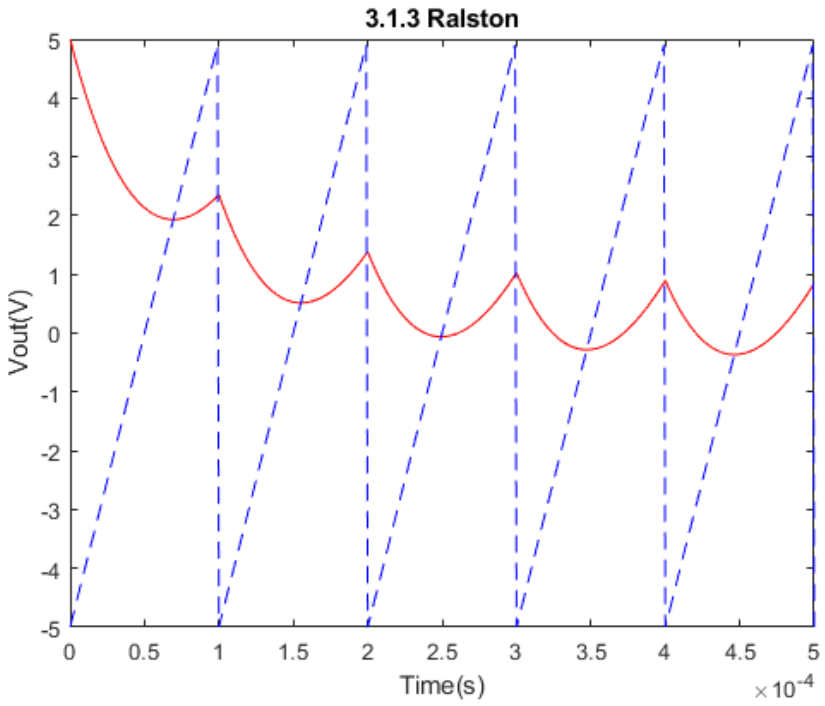

Figure 19) 3.1.3 Ralston, Sawtooth wave, $T = 10^{-4}s$

3.2.1 Sine $T = 10\ \mu s$

Due to the high frequency we plotted both input and output signals with continuous lines.

$$T = 10\ \mu s$$

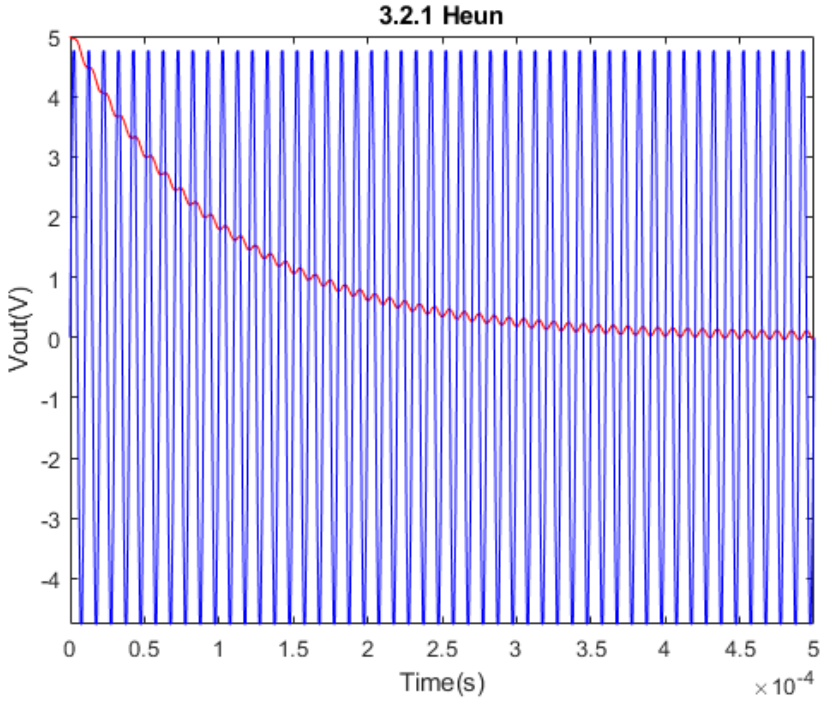

Figure 20) 3.2.1 Heun, Sine wave, $T = 10^{-5}s$

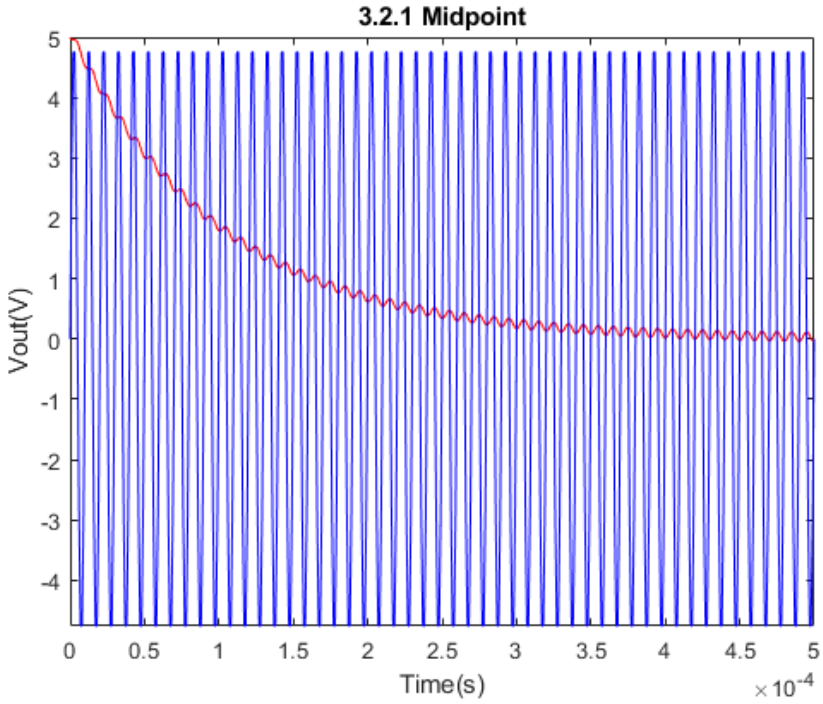

Figure 21) 3.2.1 Midpoint, Sine wave, $T = 10^{-5}s$

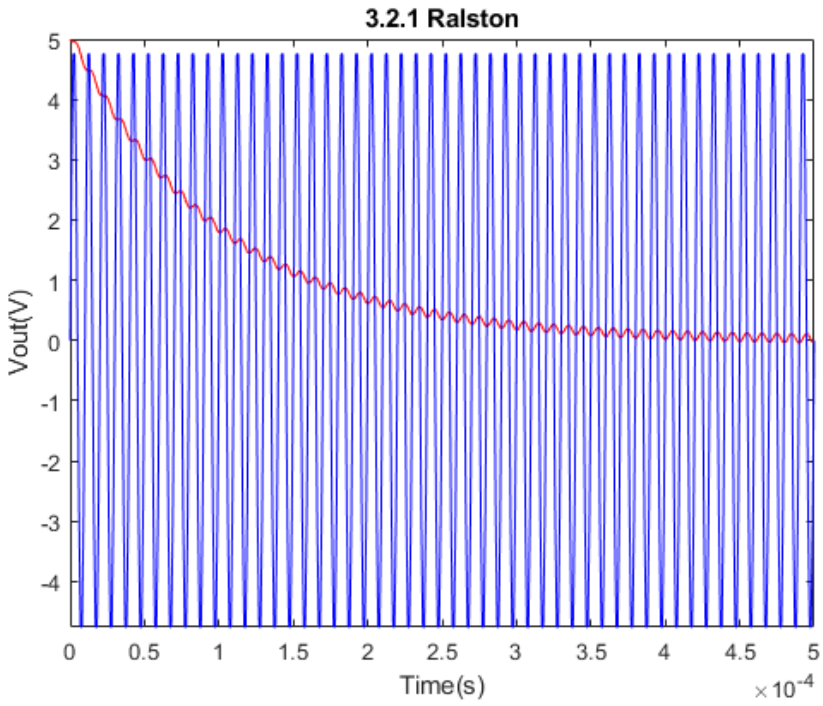

Figure 22) 3.2.1 Ralston, Sine wave, $T = 10^{-5}s$

3.2.2 Square $T = 10\ \mu s$

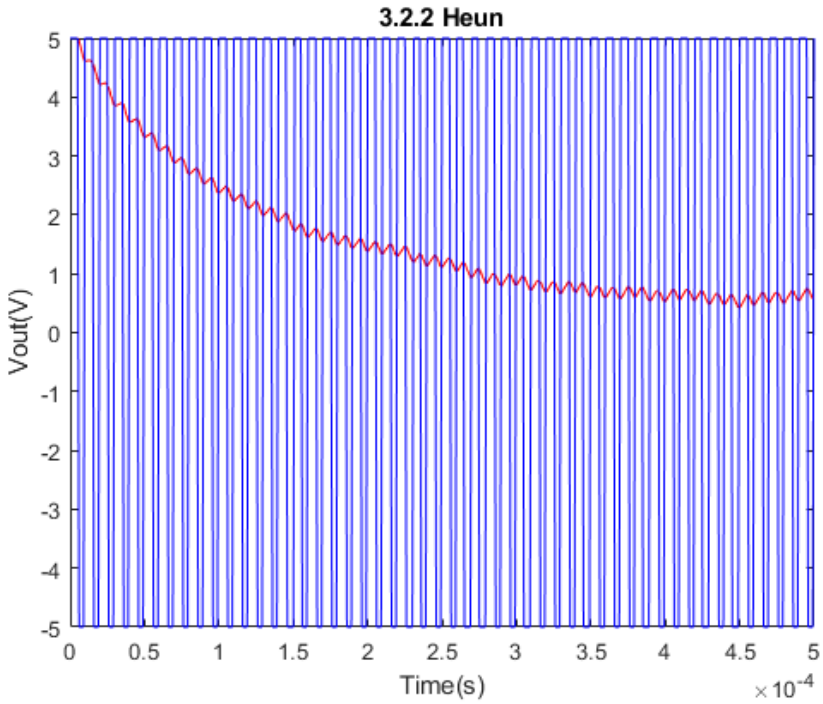

Figure 23) 3.2.2 Heun, Square wave, $T = 10^{-5}s$

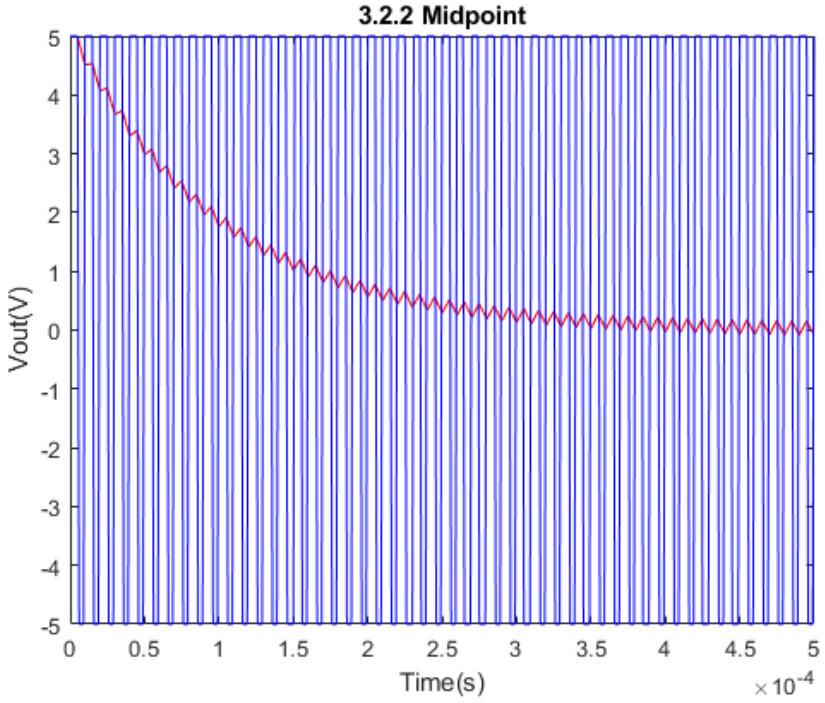

Figure 24) 3.2.2 Midpoint, Square wave, $T = 10^{-5}s$

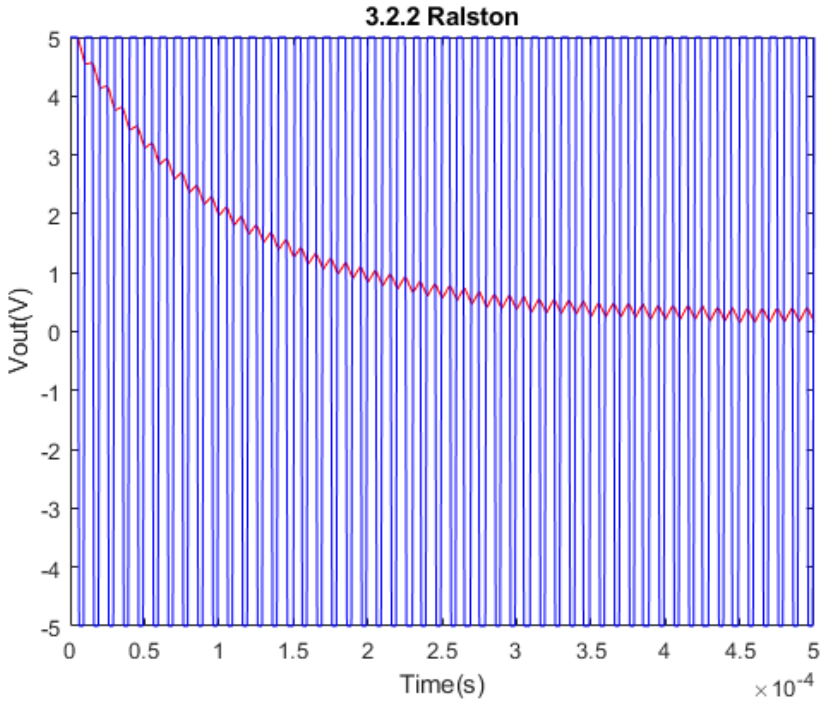

Figure 25) 3.2.2 Ralston, Square wave, $T = 10^{-5}s$

3.2.3 Sawtooth $T = 10\ \mu s$

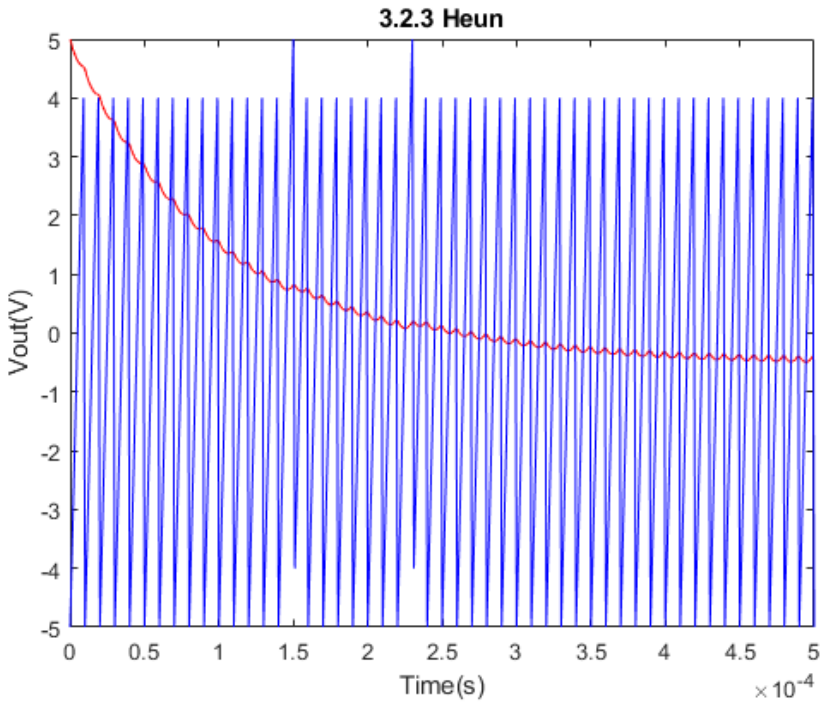

Figure 26) 3.2.3 Heun, Sawtooth wave, $T = 10^{-5}s$

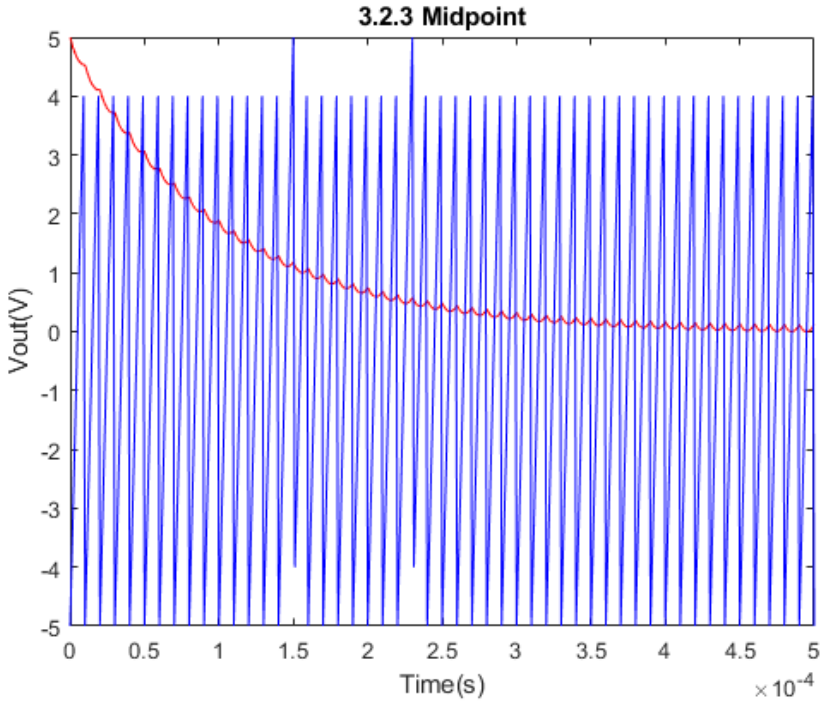

Figure 27) 3.2.3 Midpoint, Sawtooth wave, $T = 10^{-5}s$

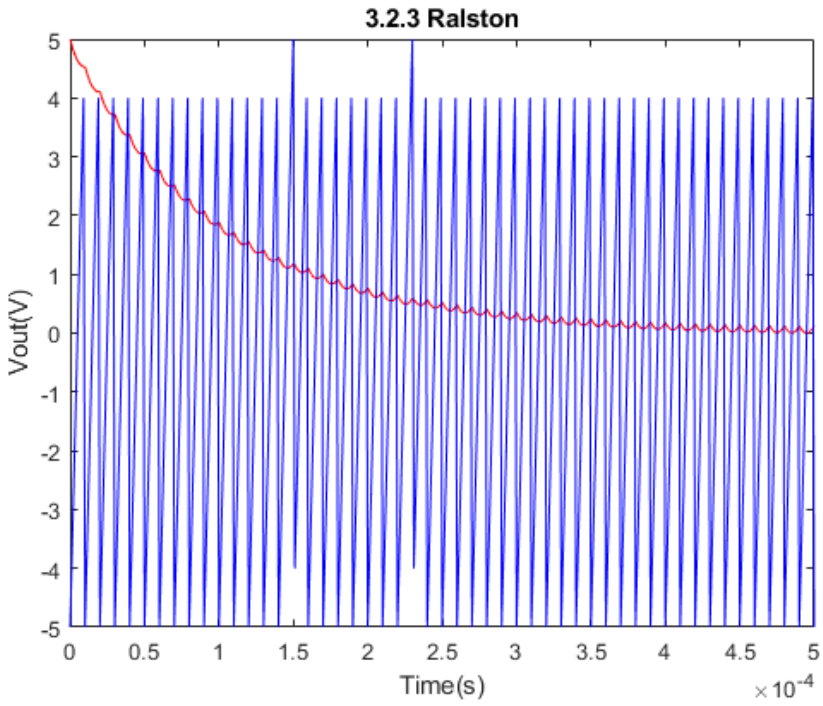

Figure 28) 3.2.3 Ralston, Sawtooth wave, $T = 10^{-5}s$

3.3.1 Sine $T = 500\ \mu s$

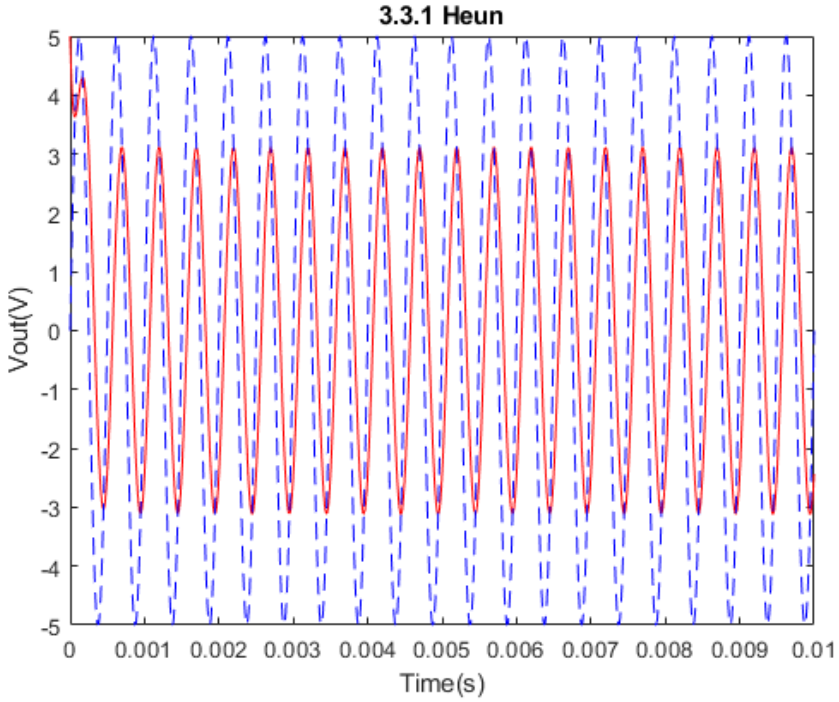

Figure 29) 3.3.1 Heun, Sine wave, $T = 5 \times 10^{-4}s$

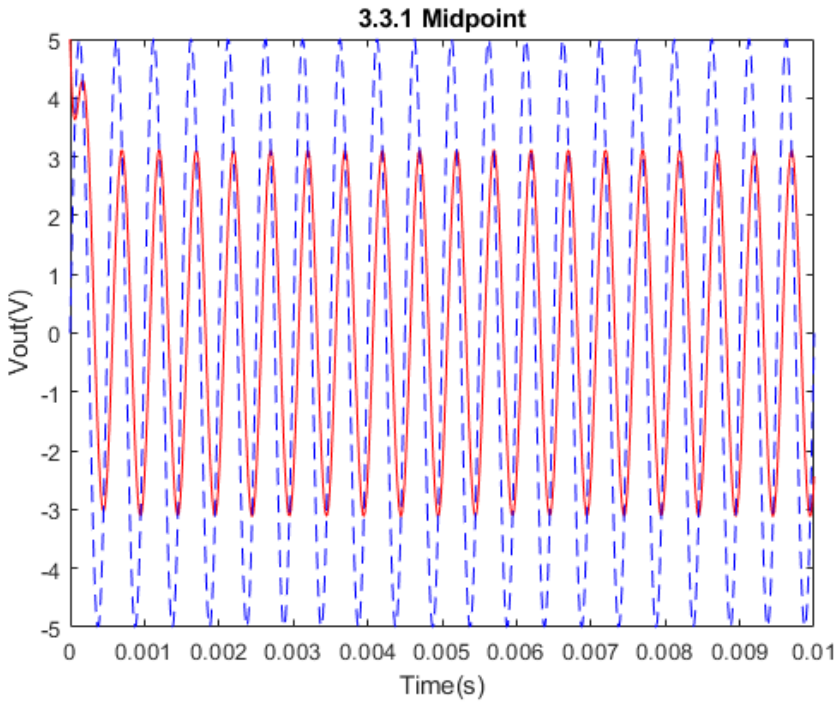

Figure 30) 3.3.1 Midpoint, Sine wave, $T = 5 \times 10^{-4}s$

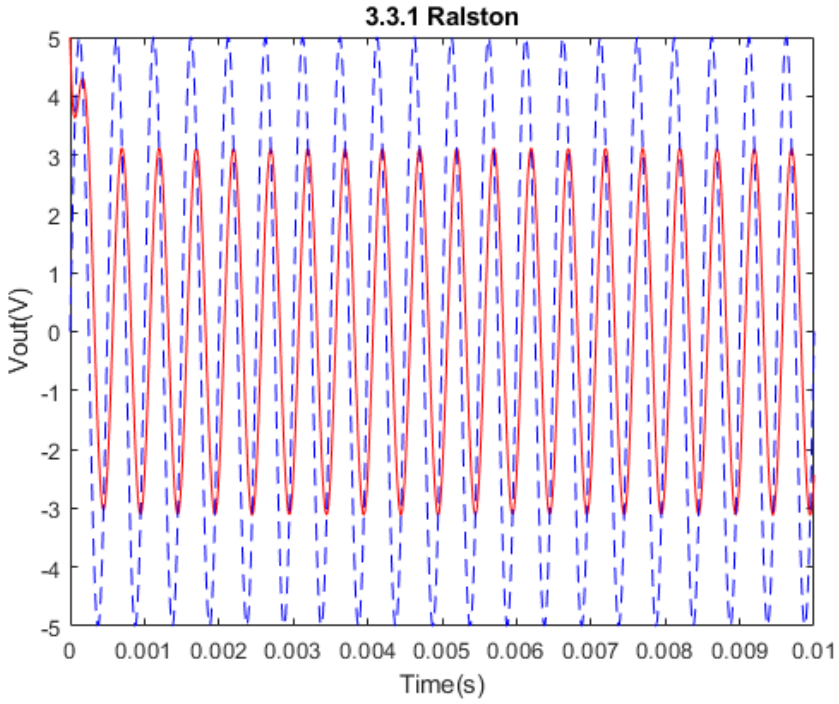

Figure 31) 3.3.1 Ralston, Sine wave, $T = 5 \times 10^{-4}s$

3.3.2 Square $T = 500\ \mu s$

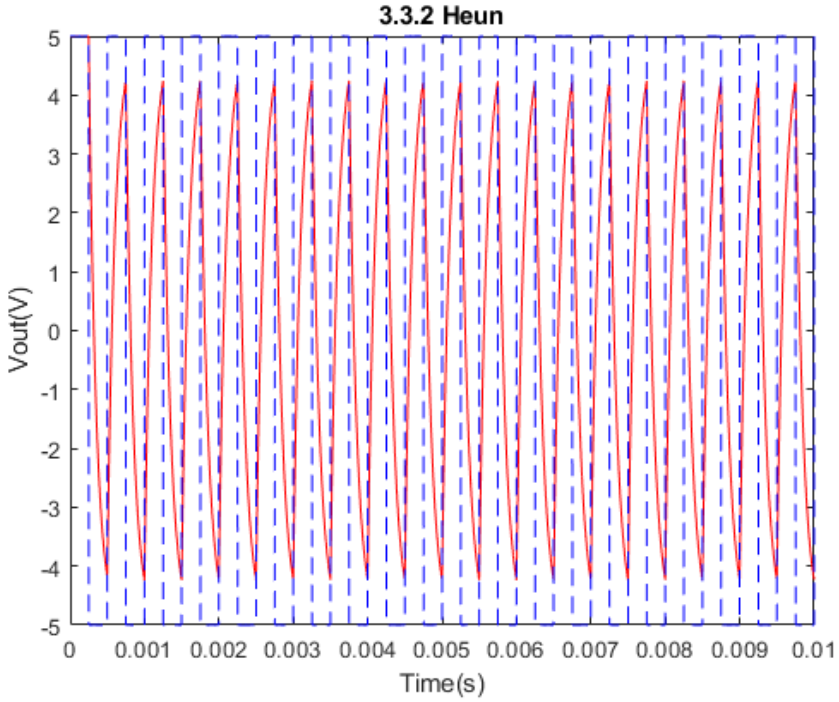

Figure 32) 3.3.2 Heun, Square wave, $T = 5 \times 10^{-4} s$

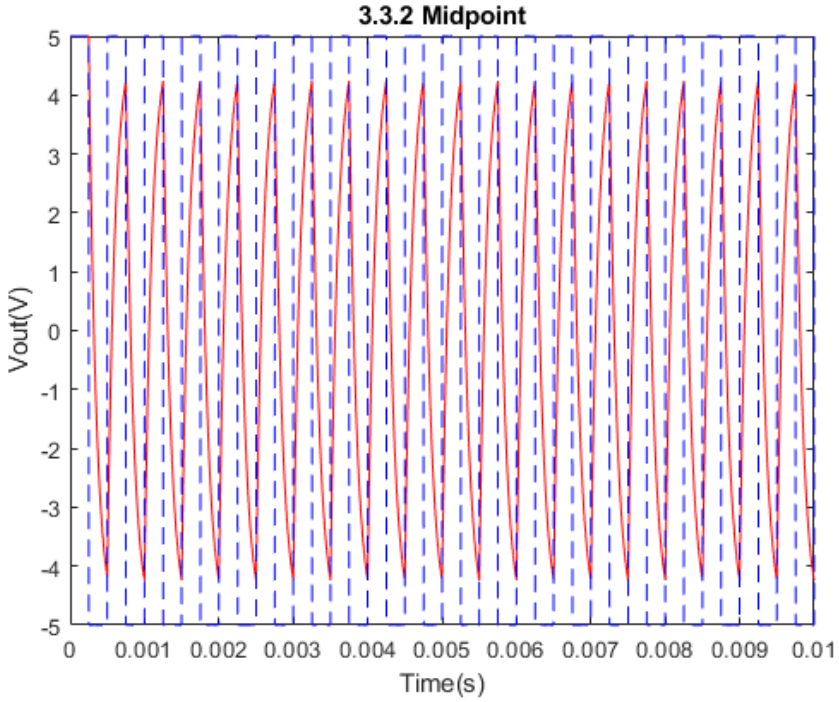

Figure 33) 3.3.2 Midpoint, Square wave, $T = 5 \times 10^{-4} s$

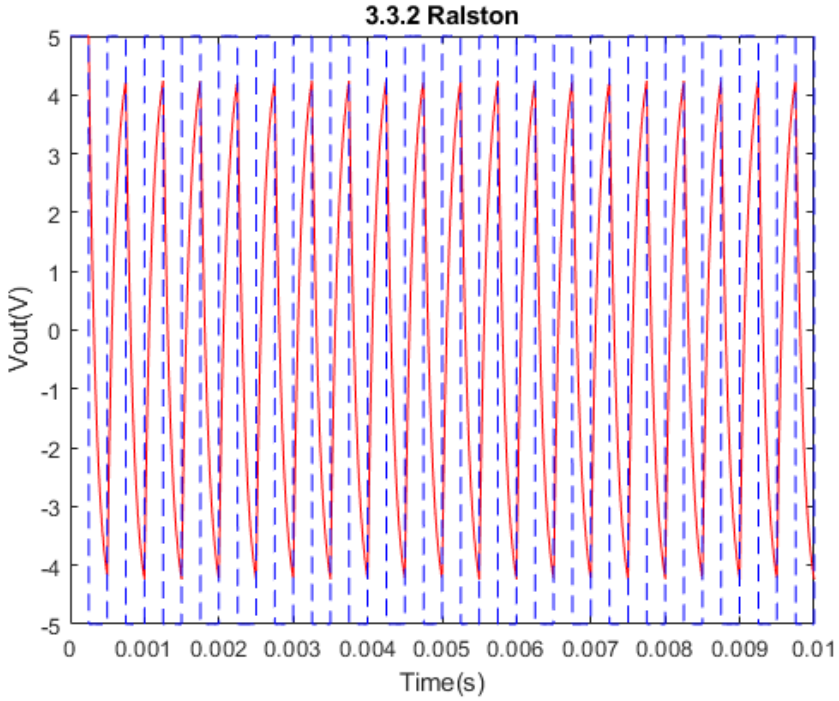

Figure 34) 3.3.2 Ralston, Square wave, $T = 5 \times 10^{-4}s$

3.3.3 Sawtooth $T = 500\ \mu s$

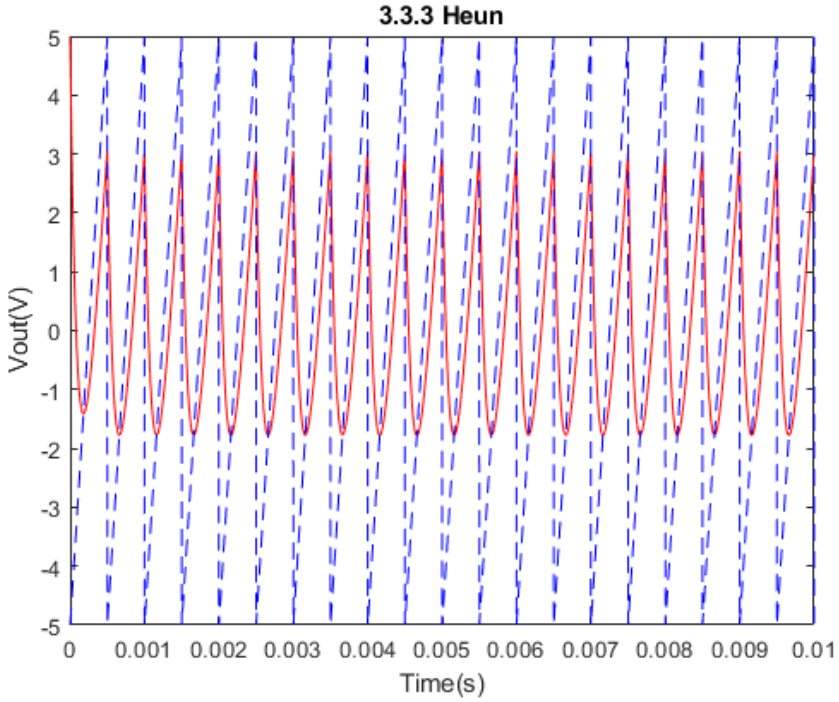

Figure 35) 3.3.3 Heun, Sawtooth wave, , $T = 5 \times 10^{-4}s$

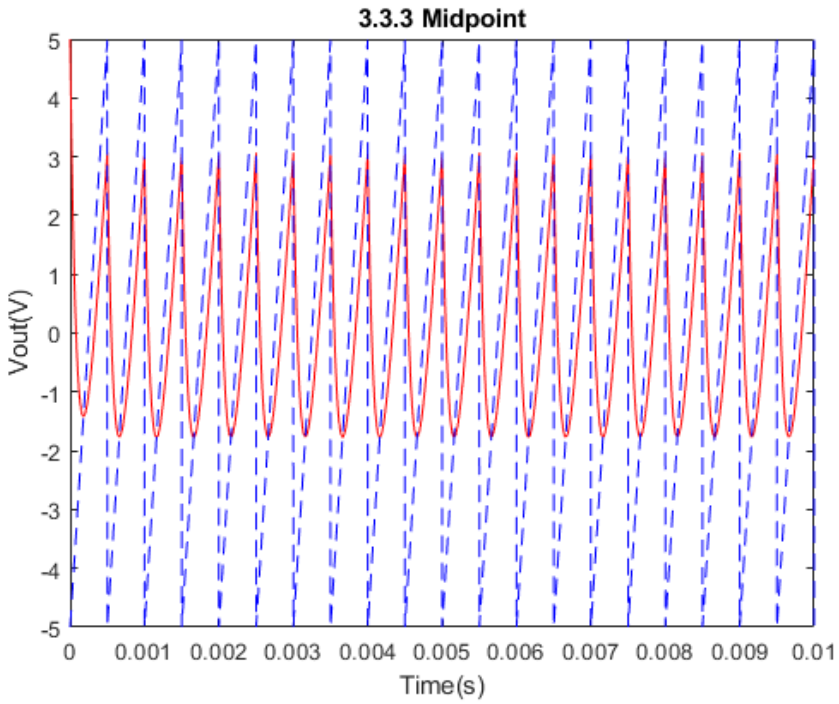

Figure 36) 3.3.3 Midpoint, Sawtooth wave, $T = 5 \times 10^{-4}s$

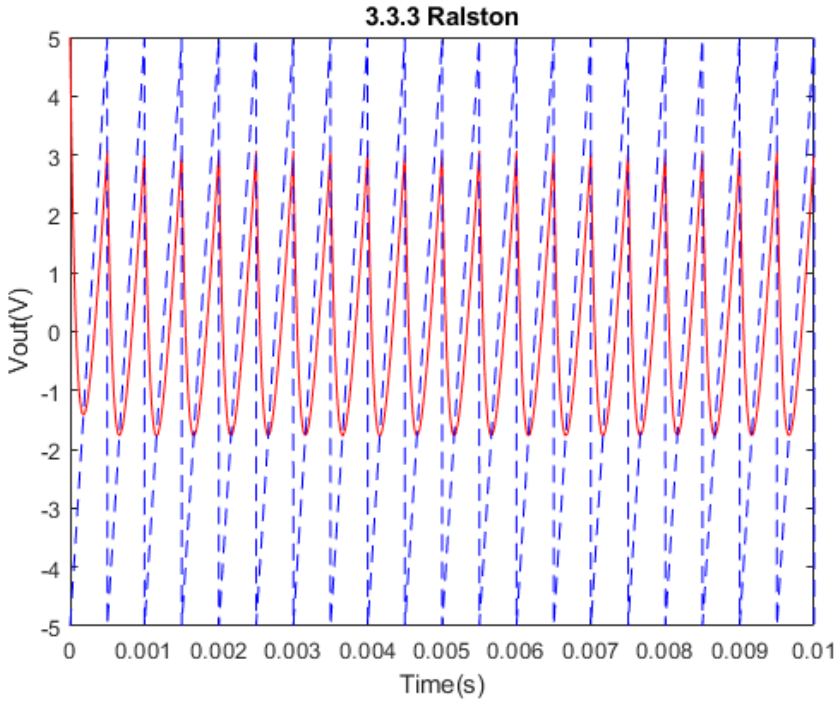

Figure 37) 3.3.3 Ralston, Sawtooth wave, $T = 5 \times 10^{-4}s$

3.4.1 Sine $T = 1000\ \mu s$

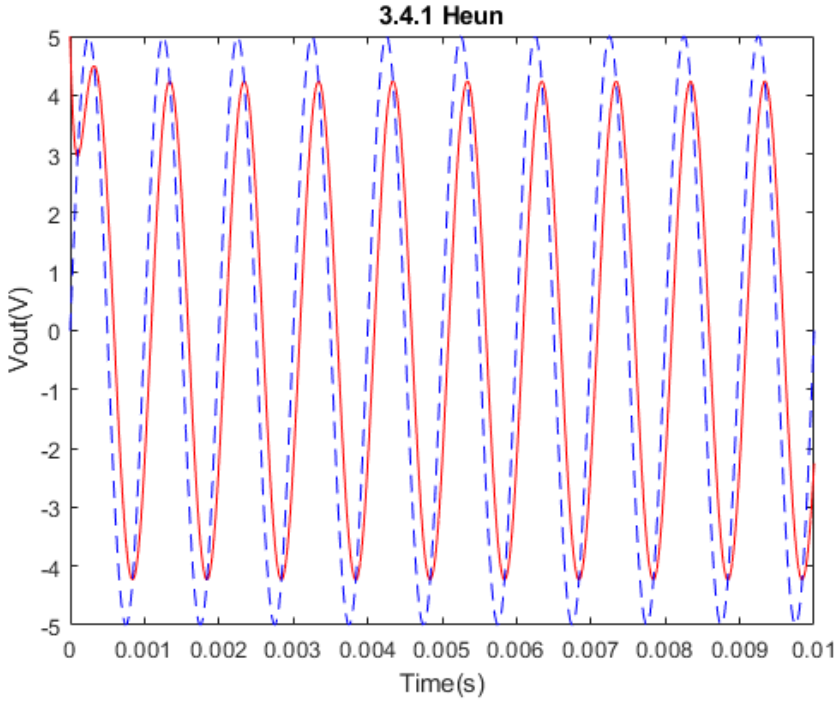

Figure 38) 3.4.1 Heun, Sine wave, $T = 10^{-3}s$

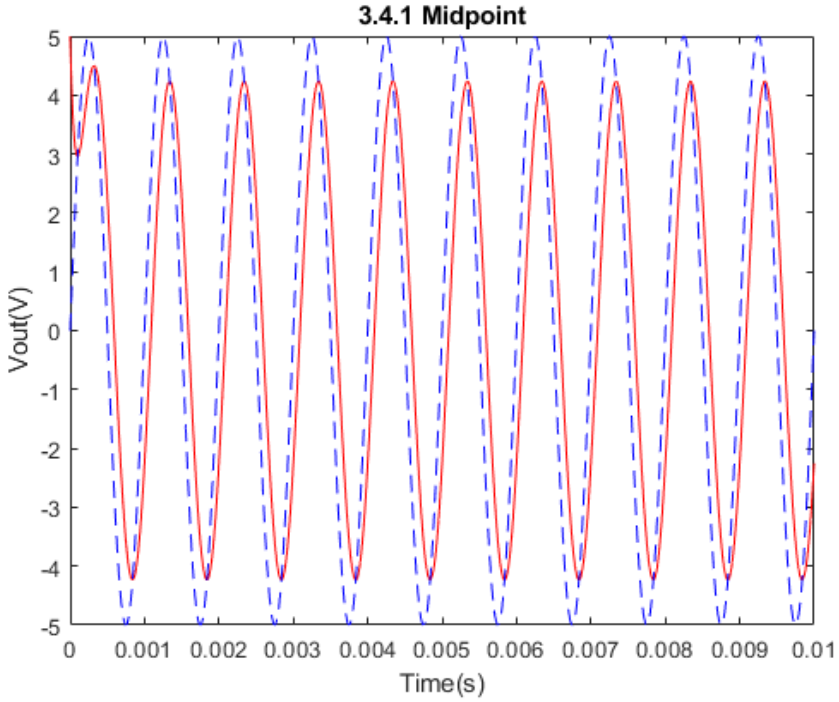

Figure 39) 3.4.1 Midpoint, Sine wave, $T = 10^{-3}s$

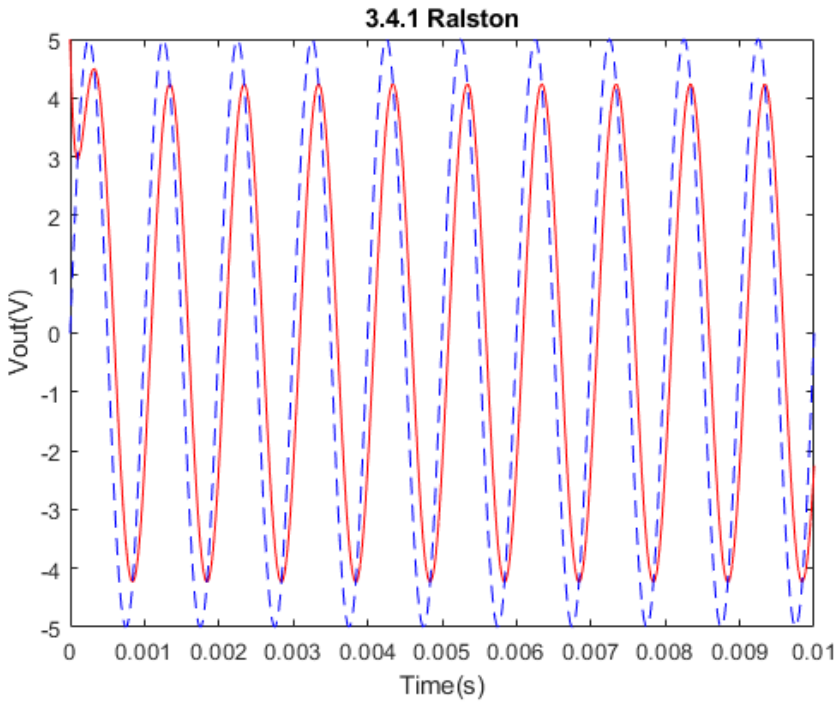

Figure 40) 3.4.1 Ralston, Sine wave, $T = 10^{-3}s$

3.4.2 Square $T = 1000\ \mu s$

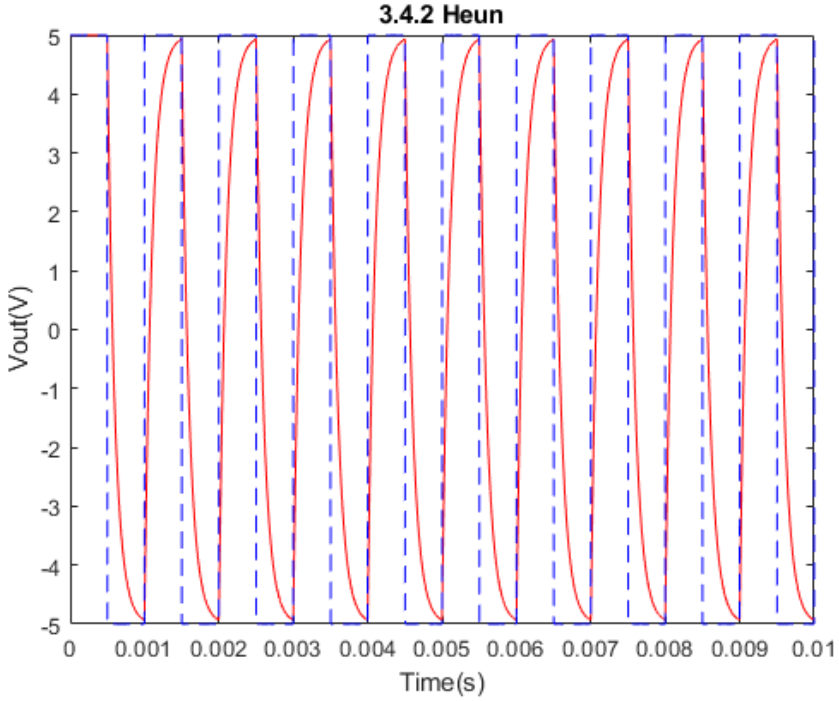

Figure 41) 3.4.2 Heun, Square wave, $T = 10^{-3}s$

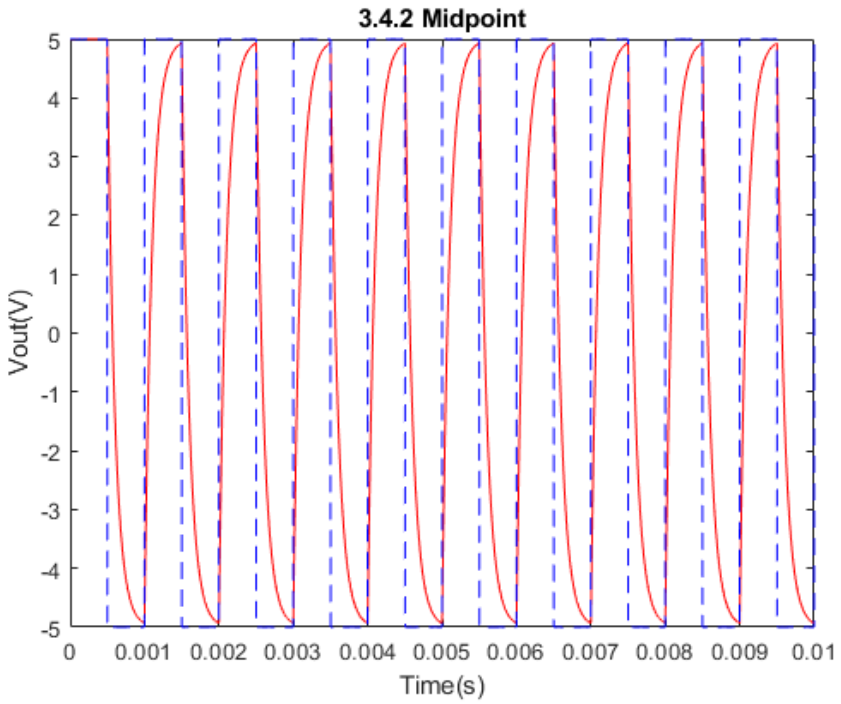

Figure 42) 3.4.2 Midpoint, Square wave, $T = 10^{-3}s$

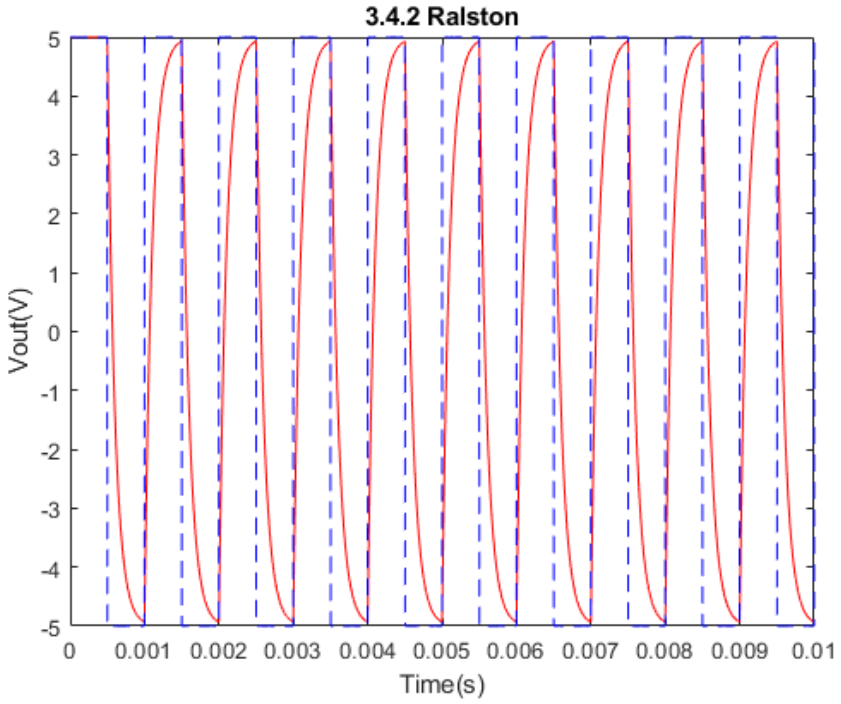

Figure 43) 3.4.2 Ralston, Square wave, $T = 10^{-3}s$

3.4.3 Sawtooth $T = 1000\ \mu s$

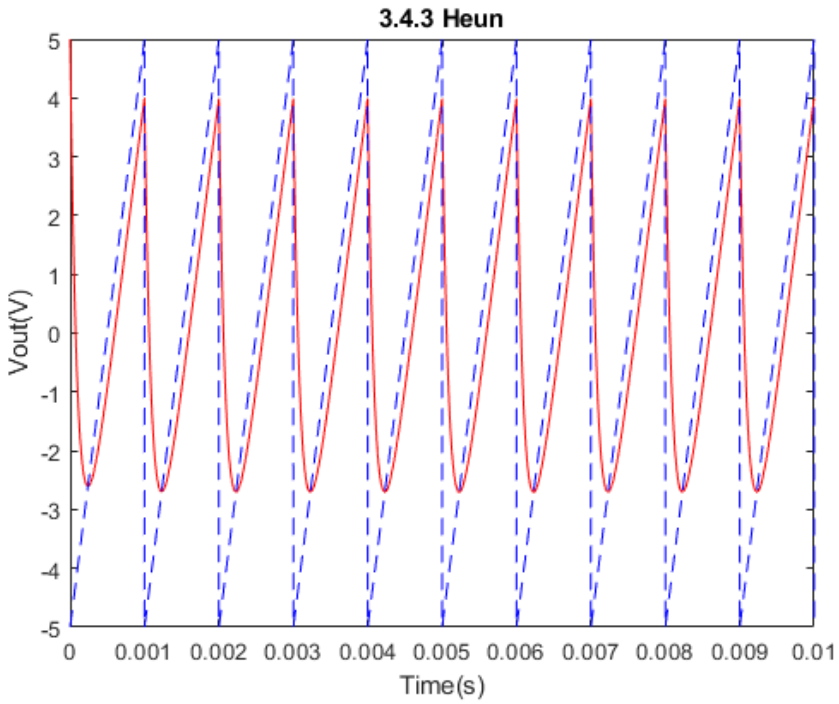

Figure 44) 3.4.3 Heun, Sawtooth wave, $T = 10^{-3}s$

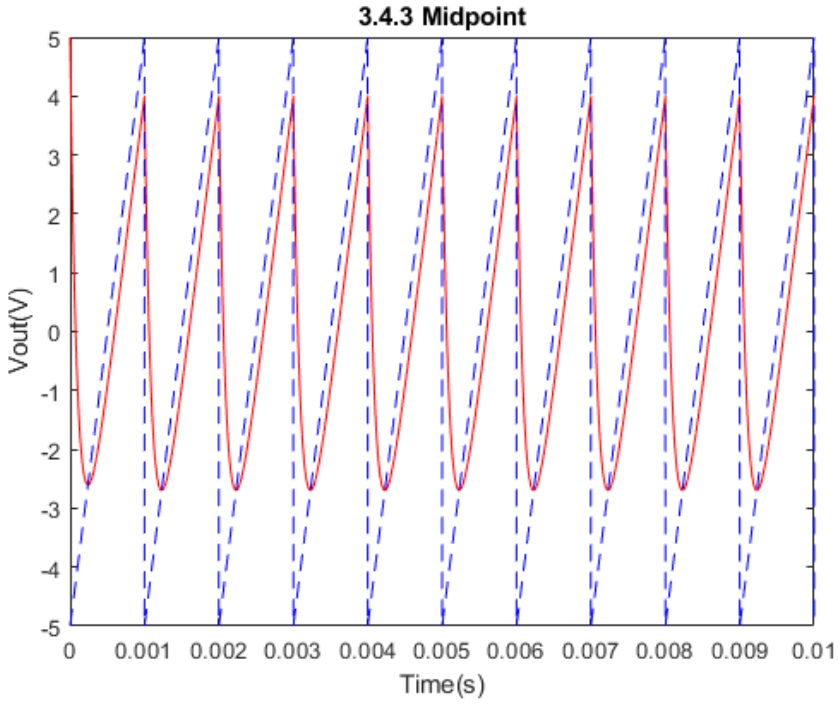

Figure 45) 3.4.3 Midpoint, Sawtooth wave, $T = 10^{-3}s$

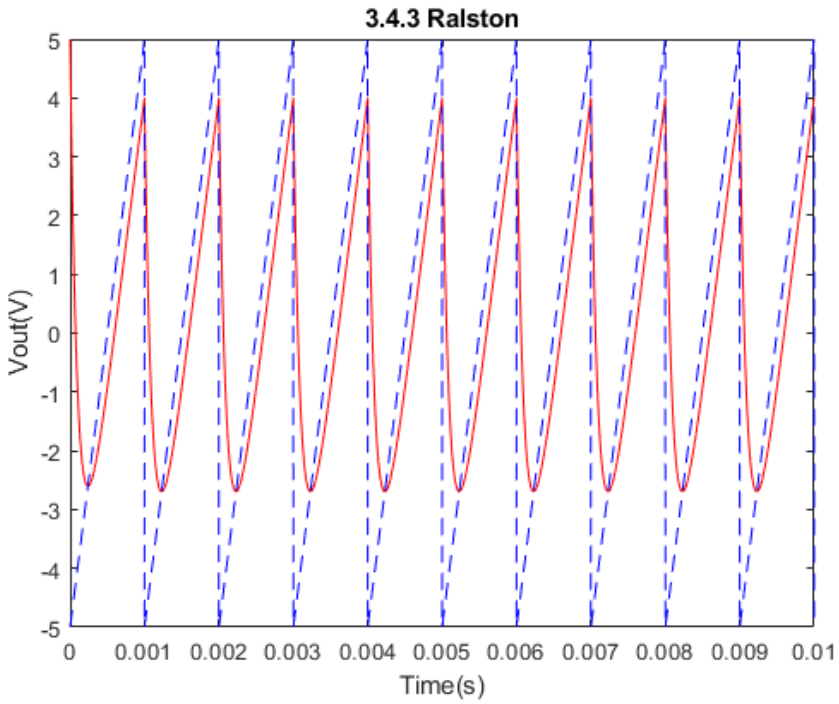

Figure 46) 3.4.3 Ralston, Sawtooth wave, $T = 10^{-3}s$

The graphs obtained with the last set of inputs seem to be the most interesting as we are observing the behaviour of different waveforms at different frequencies. It is mentioned at the beginning that circuit that we are using is a low pass filter with a cut-off frequency of $\sim 1.6kHz$ so we would expect to see lower amplitudes as we increase the frequency beyond the threshold. A good example for illustrating this is case 3.2 in which $T = 10\ \mu s$ (the lowest value of the period, which corresponds to the highest value of the frequency) as the output voltage is very low, being almost equal to 0, whereas in the last case 3.4 when $T = 1000\ \mu s$(the lowest frequency we use) , the amplitude of the output wave is barely affected by the filter.

After plotting the required graphs, we tried a few variations. As we can see from the previously obtained graphs the differences between the methods are hard to spot so for the following section we only used one of the second order Runge-Kutta methods:

1) We started by changing the initial conditions in particular: y(0) = 0V. For this particular case we observed the interesting behaviour of the decaying exponentials

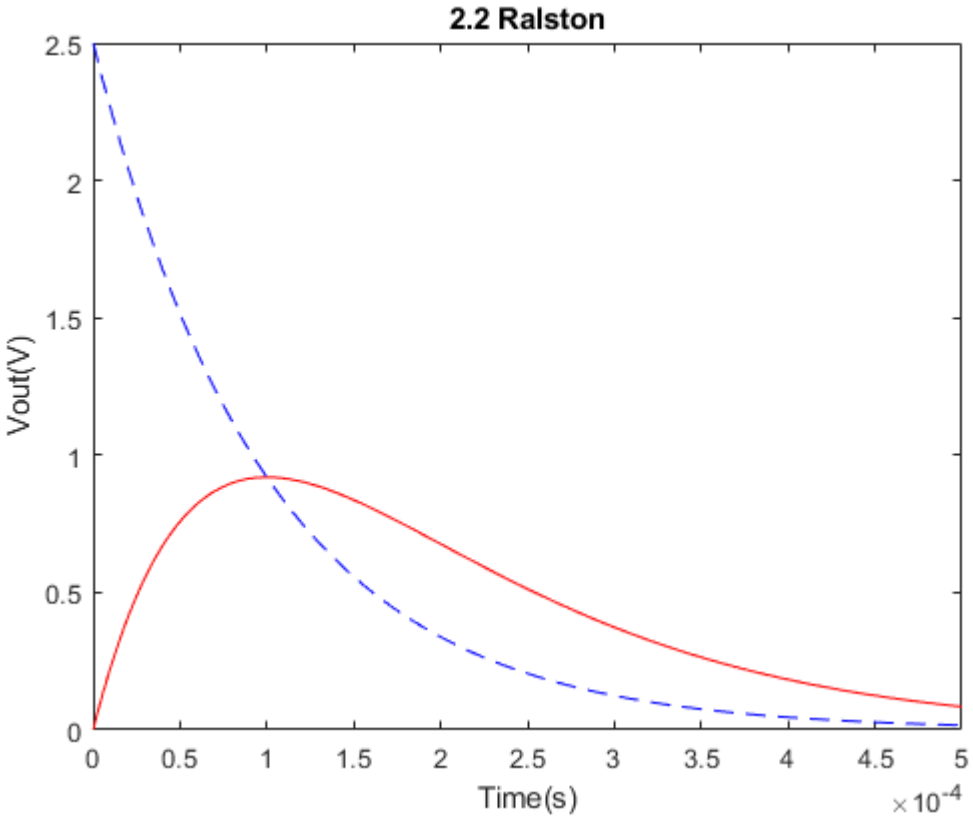

Figure 47) Ralston, decaying exponential, y(0) = 0

2) Then we varied the frequency: plotting three graphs one for each case:
   - $f = 0.8\,kHz$

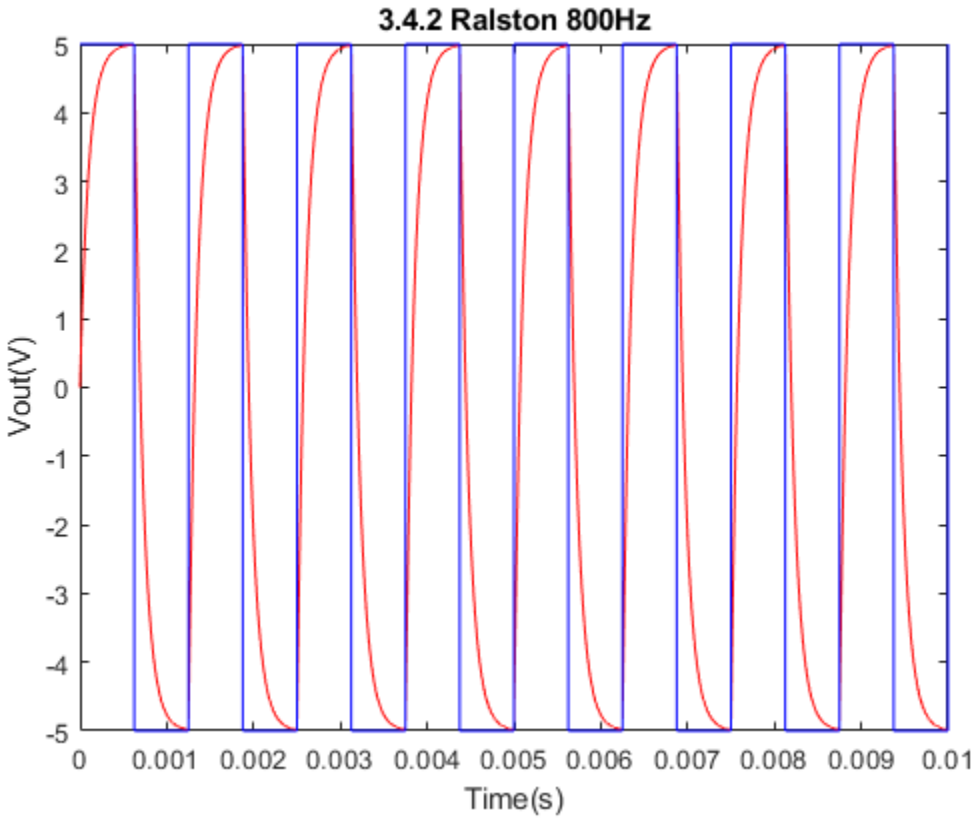

Figure 48) Ralsotn, square wave, 800Hz

- $f = 1.6kHz$ (cut-off frequency),

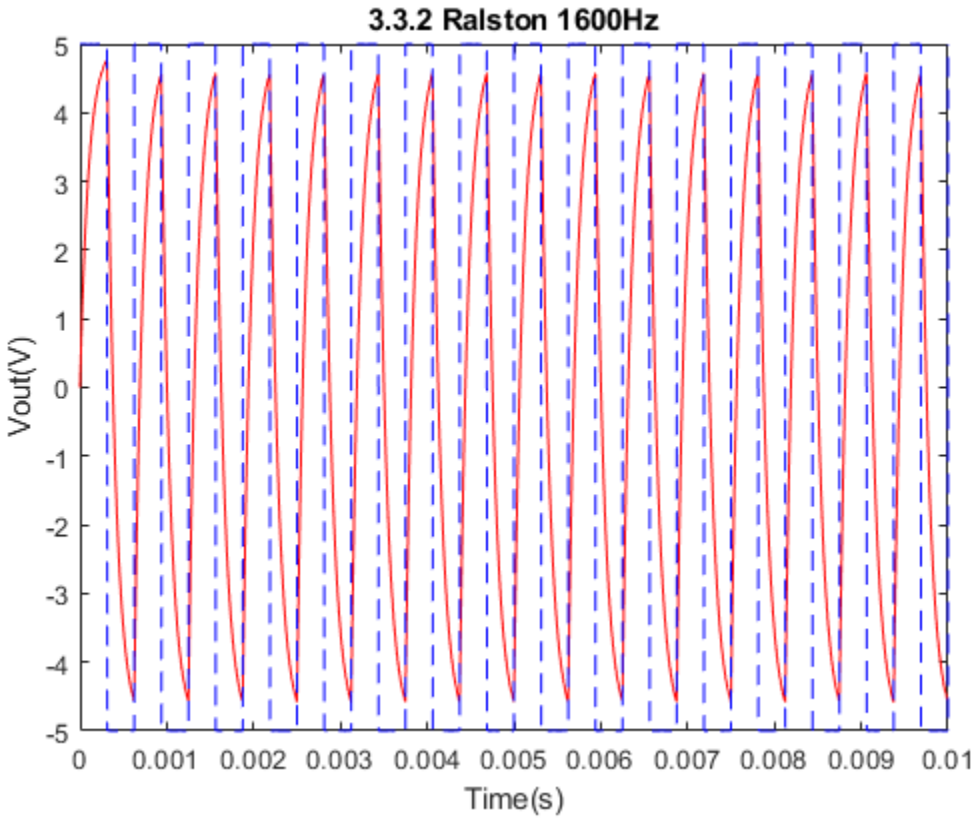

Figure 49) Ralston, square wave, 1600Hz

- $f = 3.2kHz$

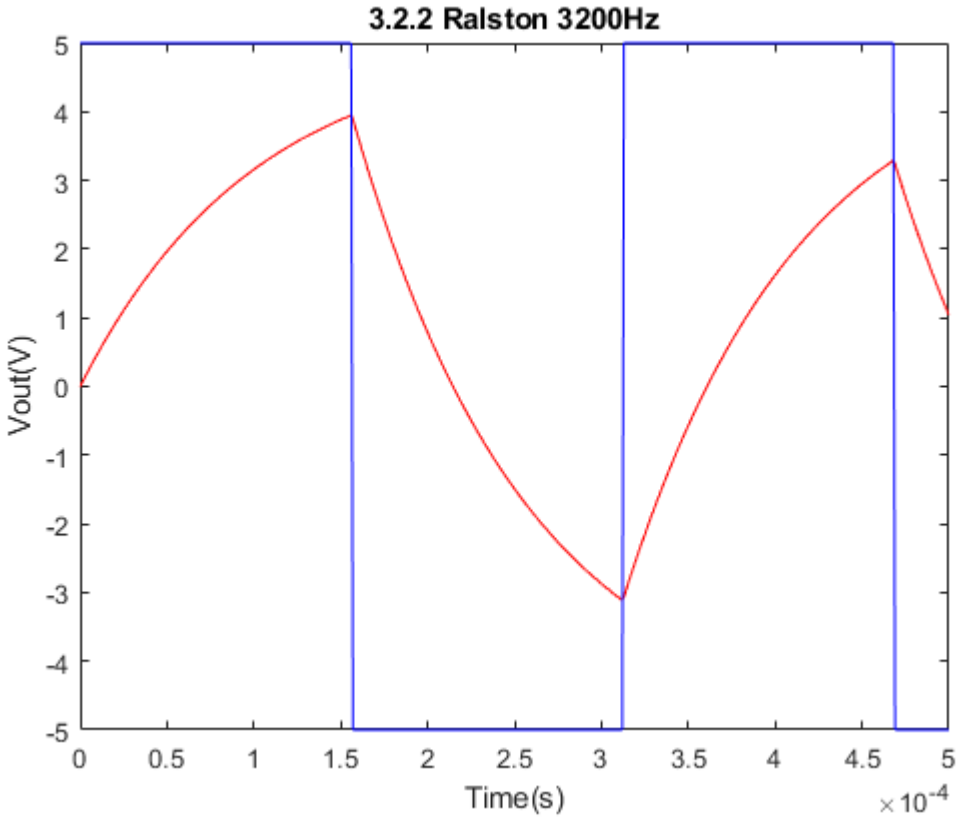

Figure 50) Ralston, square wave, 3200Hz

Note: the waveform used for generating the graphs above is the square wave

3) Finally, we varied the R and C components in order to obtain filters of different cut-off frequencies:
   - $R' = R/2, C' = C;$

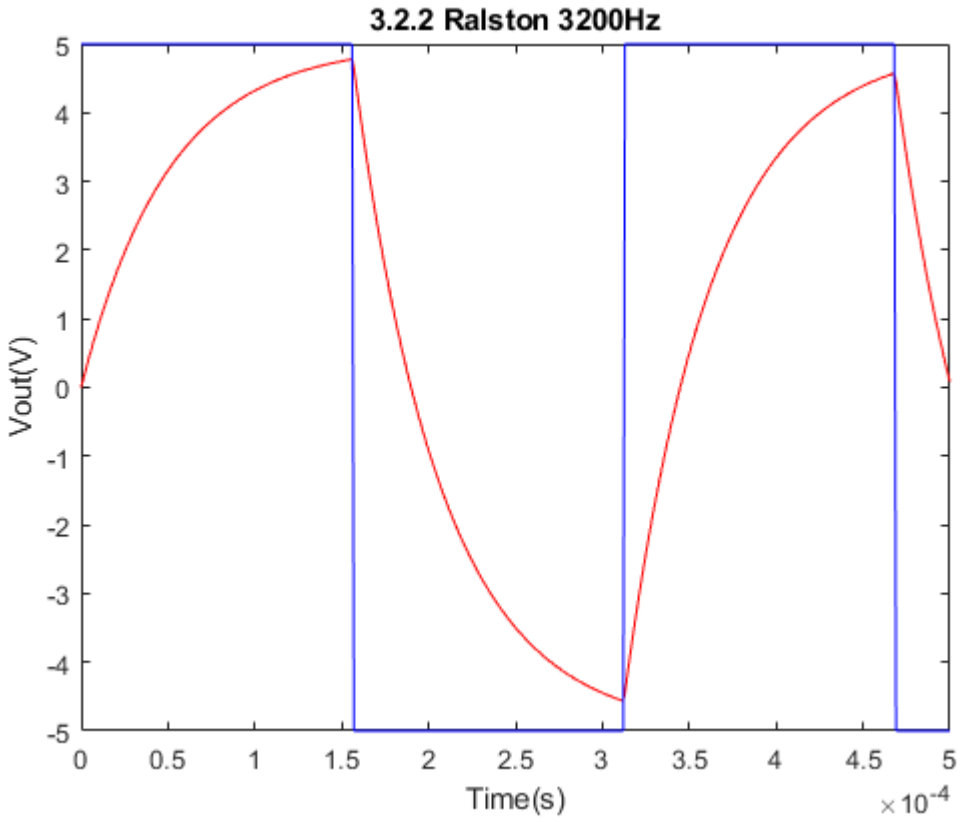

Figure 51) Ralston 3200Hz R'=R/2

By comparing this graph with the previous one we can state that the cut-off frequency increases as the value of the resistance decreases

- $R' = R, C' = C/2;$

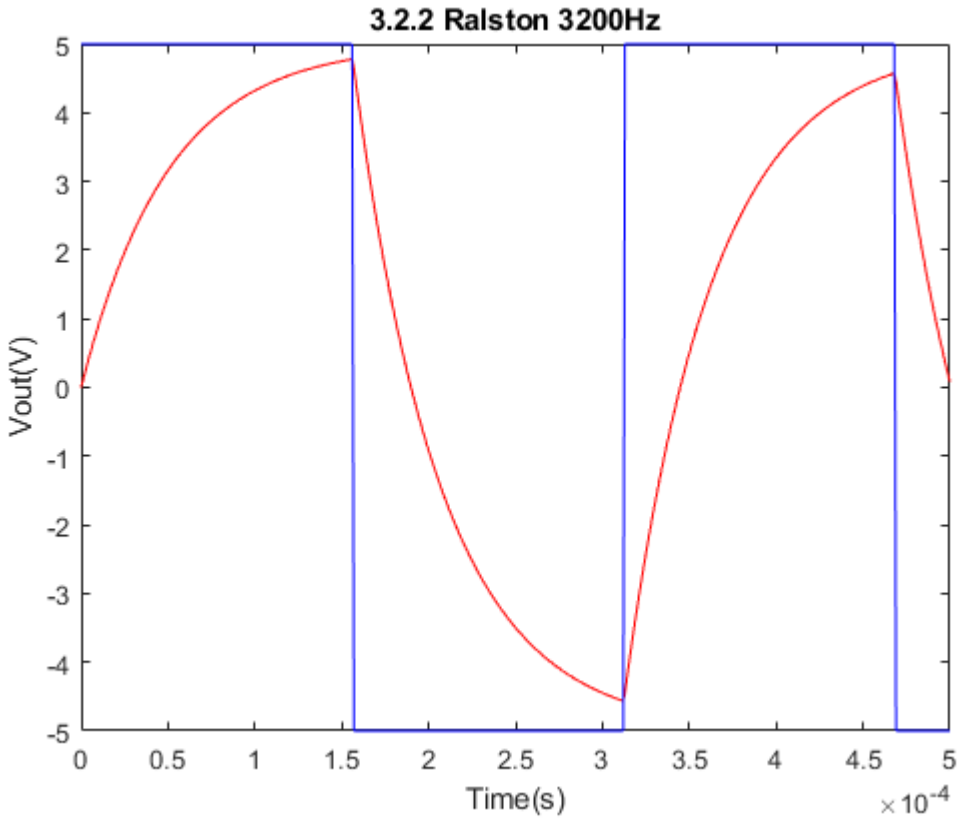

Figure 52) Ralston square wave 3200Hz C'=C/2

This time we made C half of what it was before while keeping R at the initial value and we got a similar graph, which suggests that the cut-off frequency increases as the value of the capacitance decreases

Exercise 2

In this exercise of the report, the error analysis of the previous methods is carried out using the script called **error_script.m**. In this case, the initial conditions and the circuits parameters have remained the same (i.e. $R = 1\ k\Omega$, $C = 100nF, qc(0) = 500nC$) and $Vin(t) = 5\cos(\omega t)$ where $\omega = 10^4 \times 2\pi\ rads^{-1}$. To perform the error analysis the values of the exact solution to the linear first order ODE must be compared with the approximations that were generated from the previous methods.

Solving the ODE

The ODE given has the form

$$R\frac{dq}{dt} + \frac{q}{C} = 5\cos(\omega t)$$

Let $q(t) = y(t)$ and rearrange to get

$$\frac{dy}{dt} + \frac{1}{RC}y = \frac{5}{R}\cos(\omega t)$$

(1)

The integrating factor $\mu(t)$ is now

$$\mu(t) = e^{\int \frac{dt}{RC}} = e^{\frac{t}{RC}}$$

Multiply both side of equation (1) with the integrating factor

$$e^{\frac{t}{RC}}\frac{dy}{dt} + e^{\frac{t}{RC}}y = e^{\frac{t}{RC}} \times \frac{5}{R}\cos(\omega t)$$

$$\frac{d}{dt}\left(e^{\frac{t}{RC}}y\right) = e^{\frac{t}{RC}} \times \frac{5}{R}\cos(\omega t)$$

Integrate both sides and rearrange

$$y = \frac{\frac{5}{R}\int e^{\frac{t}{RC}}\cos(\omega t)\,dt}{e^{\frac{t}{RC}}}$$

Integrate the right-hand side by parts

$$y = \frac{5C(\omega RC sin(\omega t) + \cos(\omega t))}{1 + (\omega RC)^2} + Ae^{\frac{-t}{RC}}$$

Where A is a constant

Apply the initial condition (i.e. $y(0) = 500nC$)

$$5 \times 10^{-7} = \frac{5C}{1 + (\omega RC)^2} + A$$

$$A = 5 \times 10^{-7} - \frac{5C}{1 + (\omega RC)^2}$$

Thus, the final solution is

$$y(t) = \frac{5C(\cos(\omega t) + \omega RC\,\sin(\omega t))}{1 + (\omega RC)^2} + \left(5 \times 10^{-7} - \frac{5C}{1 + (\omega RC)^2}\right)e^{\frac{-t}{RC}}$$

(2)

Generating the error plots

The code for exercise 4 (i.e. **error_script.m**) is shown below

```
%Clean up
close all;
clear;
clc;
```

```
%Choose step size, initial condition, final value
x0 = 0;
y0 = 500e-9;
h = 1e-6;
xf = 1e-3;

%Choose circuit constants
R = 1e+3;
C = 100e-9;
W = 2*pi*10^4;

%Create some constants so that the equation is easier to write
B = (5*C)/((W*R*C)^2 + 1);
A = y0 - B;

%Choose Vin and Vout
Vin = @(x) 5*cos(W*x);
Vout =@(x) B*(cos(W*x) + W*R*C*sin(W*x)) + A*exp(-x/(R*C));

dy = @(x,y) Vin(x)/R - y/(C*R);

%Let ye = qc(t)
[x,y_estimation] = RK2(dy,x0,y0,h,xf);

%Generate the arrays for Vin(t) and Vout(t)
y_real = arrayfun(@(x) Vout(x), x);

error = y_real - y_estimation;

%Plot Vin(t) and Vout(t)
plot(x, error);
title('Error with Heun method');
xlabel('Time(s)');
ylabel('Error of Charge(C)');
```

The plots below were generated with $h = 10^{-6}$ and $t\_final = 0.3ms$

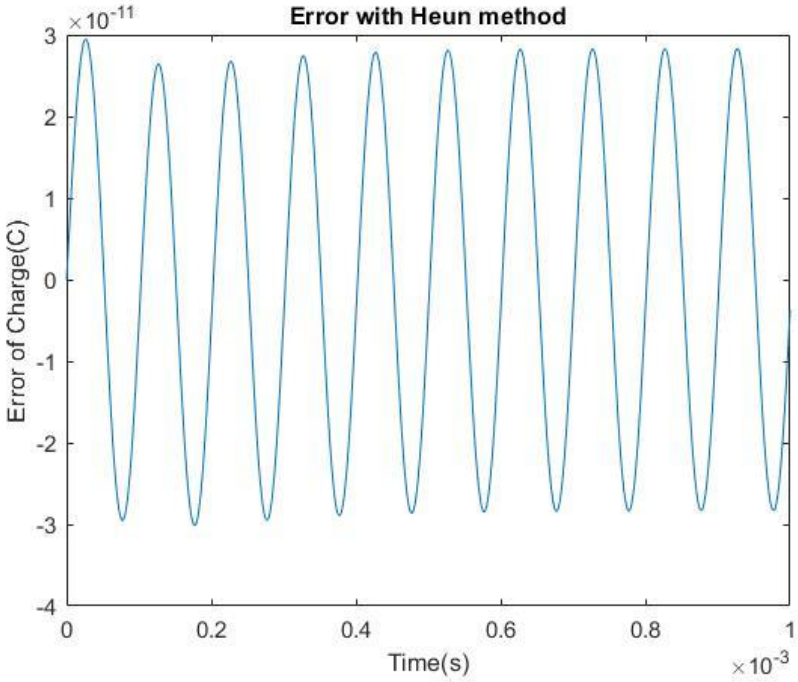

Figure 53

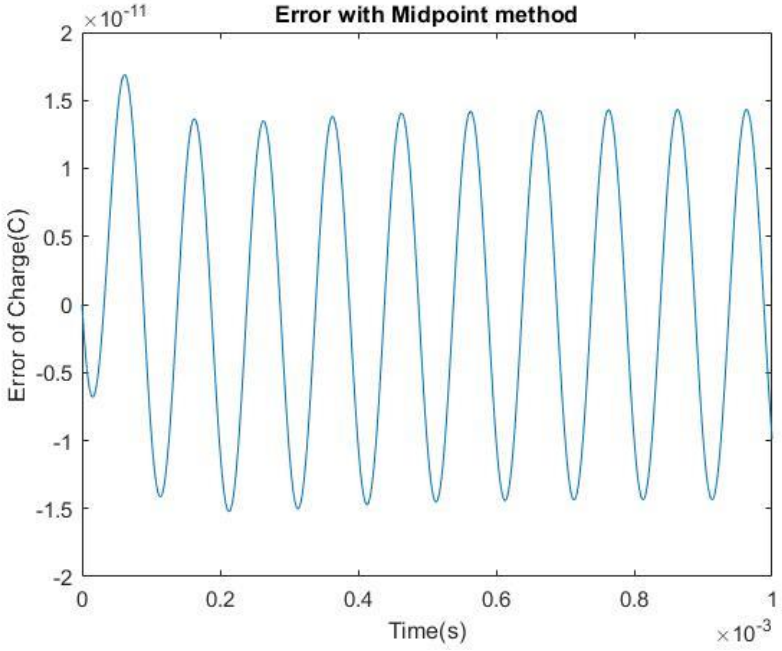

Figure 54

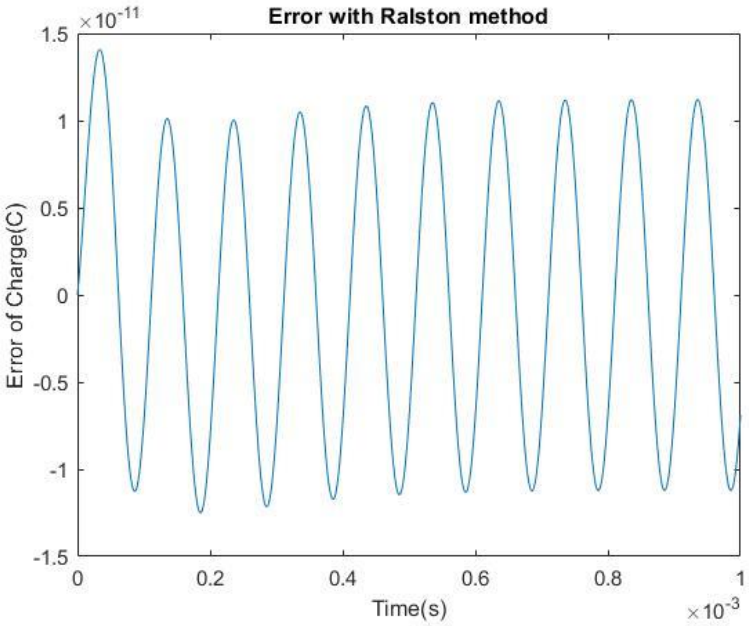

Figure 55

From the figures above, we can conclude that the error itself is has a sinusoidal shape. This seems reasonable as the difference between two sinusoidal functions (i.e. the error) is itself a sinusoidal function. Furthermore, the initial peek in the error is due to the error involved in calculating the transient state in which the charge changes rapidly. Finally, we can see that in this case, the best method to use would be Ralston which resulted in the smallest error followed by Midpoint and Heun.

By varying the step size from $h = 2 \times 10^{-5}$ to $h = 10^{-8}$, the following plot was generated

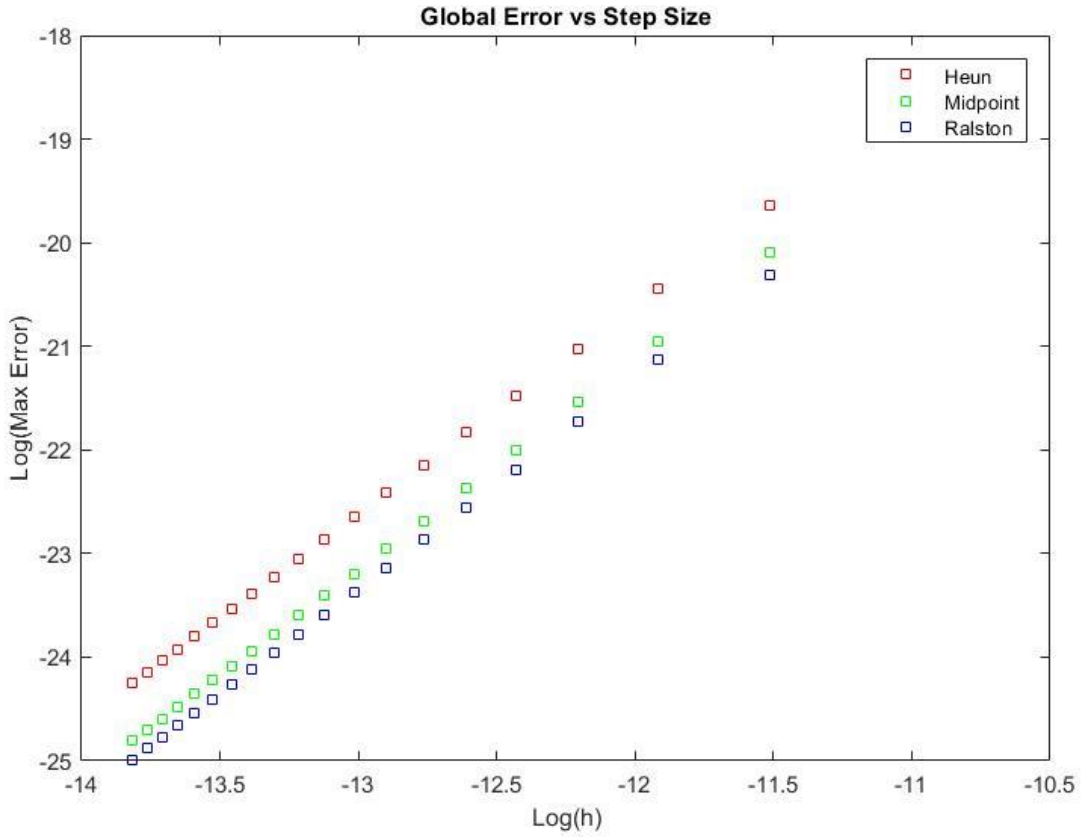

Figure 56

Furthermore, using matlab's polyfit function, the slops were calculated to be

$$Heun\ Slope\ \approx 1.9966$$

$$Midpoint\ Slope\ \approx 2.0578$$

$$Ralston\ Slop\ \approx 2.0531$$

Since this is a logarithmic plot, the gradient of the line corresponds to the order of the order. In this case, all the gradients are approximately 2 so the global error is of order $O(h^2)$ which is what is expected from second order RK methods.

2. RLC Circuit

Background

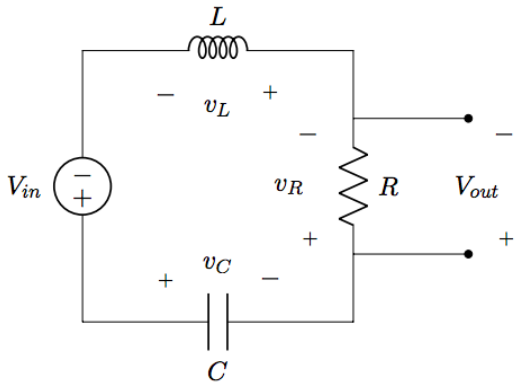

Figure 57: RLC Circuit

An RLC circuit is an example of a harmonic oscillator, where Voltage and Current can be thought of as to harmonically oscillate. The system can be modelled as follows:

$$v_L(t) + v_C(t) + v_R(t) = V_{in}(t)$$

Or

$$L\frac{d}{dt}i_L(t) + Ri_L(t) + \frac{1}{C}\int_0^t i_L(t)dt = V_{in}(t)$$
$$L\frac{d^2}{dt^2}q_C(t) + R\frac{d}{dt}q_C(t) + \frac{1}{C}q_C(t) = V_{in}(t)$$

We can use Runge-Kutta methods to solve 1st Order ODE's. As the specification requires, in particular, we can use the 4th order Runge-Kutta method. The algorithm is used to calculate some future y_{i+1} based on a current known value y_i. It can be demonstrated as follows:

$$y_{i+1} = y_i + \frac{h}{6}(k_1 + 2k_2 + 2k_3 + k_4)$$

where:

$$k_1 = h * f(x_i, y_i)$$
$$k_2 = h * f\left(x_i + \frac{h}{2}, y_i + \frac{k_1 * h}{2}\right)$$
$$k_3 = h * f\left(x_i + \frac{h}{2}, y_i + \frac{k_2 * h}{2}\right)$$
$$k_4 = h * f(x_i + h, y_i + k_3 * h)$$

and:

$$h = size\ of\ interval$$

Given that the 4th order Runge-Kutta Method only applies to 1st order ODE's, we must attempt to convert our 2nd order Differential Equation modelling the RLC behavior, into 2 coupled 1st order Differential Equations as follows:

$$If \quad y = q_C(t) \;\; and \; z = \frac{d}{dt} q_c(t)$$

Then:

$$y' = \; z$$

$$z' = \frac{V_{in}(t) - R * z - \frac{y}{C}}{L}$$

Given that we have two ODE's in y and $\frac{dy}{dt}$, and our function implemented in RKM4.m requires that our two coupled functions be of the form:

$$z' = f_1(x, y, z) \;\; and \;\; y' = f_2(x, y, z)$$

We simply rewrite the coupled differential equations as follows:

$$f_1(y, q, t) = \frac{d}{dt} q_C(t)$$

$$f_2(y, q, t) = \frac{V_{in}(t) - R * \frac{d}{dt} q_c(t) - \frac{q_C(t)}{C}}{L}$$

Exercise 3

The implementation for 4th order Runge-Kutta was implemented as thus:

```
function [x1, x2] = RK4(dx1, dx2, x1_0, x2_0, h, t0,  tf)
    x1(1) = x1_0;
    x2(1) = x2_0;
    N = round((tf - t0) / h); %nr of steps: (interval size)/(step
size)
    t = t0 : h : N*h;
    %Setting up fourth order RK
    a1 = 1/6;
    a2 = 1/3;
    a3 = 1/3;
    a4 = 1/6;
    %Inefficient solution
    %{
    k1 = @(t, x1, x2) dx1(t, x1, x2);
    k2 = @(t, x1, x2) dx1(t + 0.5*h, x1 + 0.5*k1(t, x1, x2)*h, x2 +
0.5*j1(t, x1, x2)*h);
    k3 = @(t, x1, x2) dx1(t + 0.5*h, x1 + 0.5*k2(t, x1, x2)*h, x2 +
0.5*j2(t, x1, x2)*h);
    k4 = @(t, x1, x2) dx1(t + h, x1 + k3(t, x1, x2)*h, x2 + j3(t, x1,
x2)*h);
```

```
    j1 = @(t, x1, x2) dx2(t,x1,x2);
    j2 = @(t, x1, x2) dx2(t + 0.5*h, x1 + 0.5*k1(t, x1, x2)*h, x2 +
0.5*j1(t, x1, x2)*h);
    j3 = @(t, x1, x2) dx2(t + 0.5*h, x1 + 0.5*k2(t, x1, x2)*h, x2 +
0.5*j2(t, x1, x2)*h);
    j4 = @(t, x1, x2) dx2(t + h, x1 + k3(t, x1, x2)*h, x2 + j3(t, x1,
x2)*h);
    %}

    for i = 1 : length(t) - 1
        k1 = dx1(t(i), x1(i), x2(i));
        j1 = dx2(t(i), x1(i), x2(i));
        k2 = dx1(t(i) + 0.5*h, x1(i) + 0.5*h*k1, x2(i) + 0.5*h*j1);
        j2 = dx2(t(i) + 0.5*h, x1(i) + 0.5*h*k1, x2(i) + 0.5*h*j1);
        k3 = dx1(t(i) + 0.5*h, x1(i) + 0.5*h*k2, x2(i) + 0.5*h*j2);
        j3 = dx2(t(i) + 0.5*h, x1(i) + 0.5*h*k2, x2(i) + 0.5*h*j2);
        k4 = dx1(t(i) + h, x1(i) + h*k3, x2(i) + h*j3);
        j4 = dx2(t(i) + h, x1(i) + h*k3, x2(i) + h*j3);

        x1(i+1) = x1(i) + h*(a1*k1 + a2*k2 + a3*k3 + a4*k4);
        x2(i+1) = x2(i) + h*(a1*j1 + a2*j2 + a3*j3 + a4*j4);
    end
```

The Script file used to test the function was written as follows:

```
close all;
clear;
clc;

%Forcing term
Vin = @(t) 5;
%Vin = @(t) 5.*exp(-(t.^2)./0.000003);
%Vin = @(t) 5*square(500*2*pi*t); %needs another add-on
%Vin = @(t) sin(500*2*pi*t);

%Initial conditions
x1(1) = 500*10^-9;
x2(1) = 0;

%Stepsize and final time
h = 0.0001;
t(1) = 0;
tf = 0.5;

%Circuit characteristics
R = 250;
C = 3.5*10^-6;
L = 600*10^-3;

%Let x1 = qc and x2 = d/dt(qc)
```

```
dx1 = @(t, x1, x2) x2;
dx2 = @(t, x1, x2) (Vin(t) - (1/C)*x1 - R*x2)/L;

[x1, x2] = RK4(dx1, dx2, x1(1), x2(1), h, t(1), tf);

N = round((tf - t(1)) / h); %nr of steps: (interval size)/(step size)
t = t(1) : h : N*h;

Vout = R*x2;
plot(t, Vout, 'r');
hold on;
fplot(Vin,'--');
xlabel('time (s)');
ylabel('Vout (V)');
xlim([0,0.05]);
ylim([-2,6]);
legend('Vout','Vin')
```

1. Step signal with amplitude V_{in}=5V

Note: For all Graphs plotted after this point up until section 3, The convention used is as follows:

'- -': Dotted lines plotted in blue represent V_{in}
'—':Lines plotted in red represent V_{out}

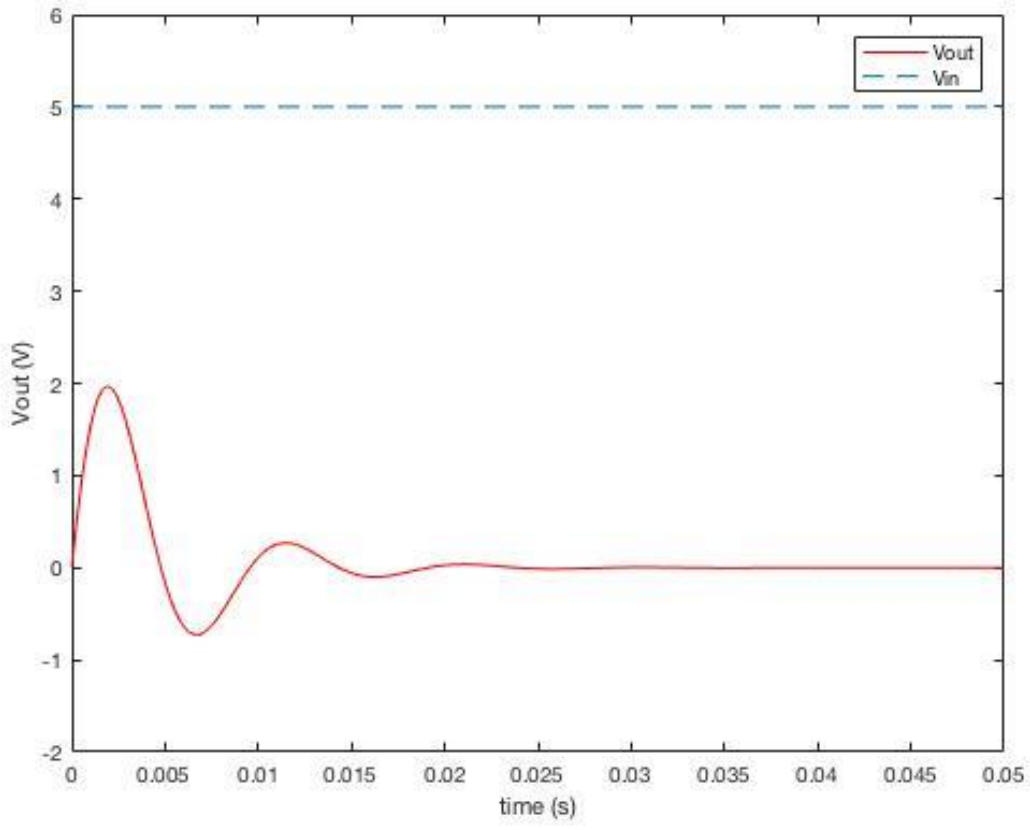

Figure 58: Step signal with amplitude Vin=5V

The transfer function of the circuit can be represented as:

$$\frac{V_{out}}{V_{in}} = \frac{j\omega RC}{(j\omega)^2 LC + (j\omega)RC + 1}$$

For analysis purposes, we can calculate the quality factor of the denominator quadratic as:

$$\zeta = \frac{b}{\sqrt{4ac}}$$

Therefore:

$$\zeta = \frac{RC}{\sqrt{4LC}}$$

For the system shown, the quality factor $\zeta \approx 0.3$. For $\zeta < 1$, our system is underdamped and oscillation is expected to occur.

To investigate what happens at critical damping, we set $\zeta \approx 1$ by changing the Resistor value from

$$R = 250\Omega \; to \; R = \; 828.1\Omega,$$

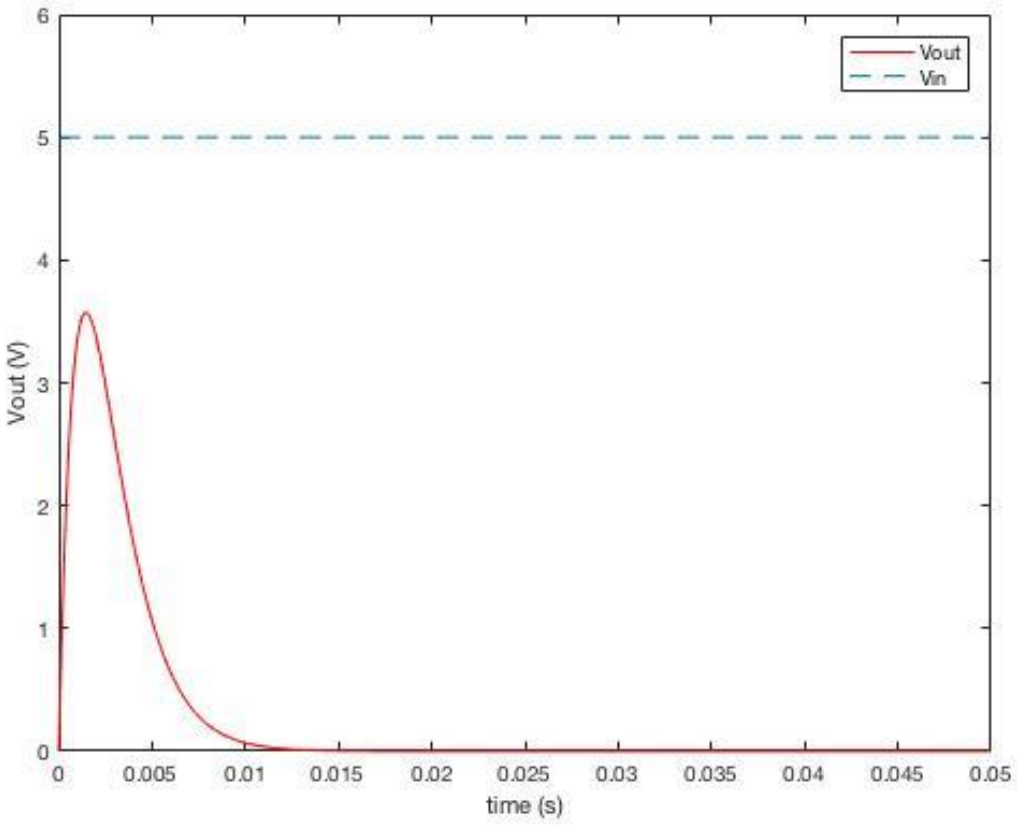

Figure 59: Quality Factor set to 1

As we can see, with a critically damped system, there are no oscillations.

2. V_{in} = Impulsive signal with decay

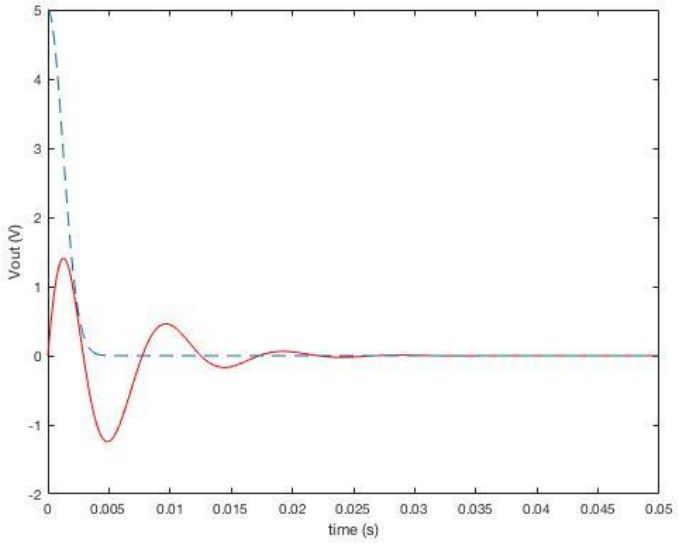

Figure 60: Impulsive Signal with Decay

As seen, the exponential decay is of the order e^{-t^2} so Vin approaches 0 rapidly. As stated before, oscillation in the circuit as the system is underdamped, shown by quality factor analysis.

If we set $\zeta \approx 1$ by changing the Resistor value from $R = 250\Omega$to $R = 828.1\Omega$, we can see again what happens to the system at critical damping.

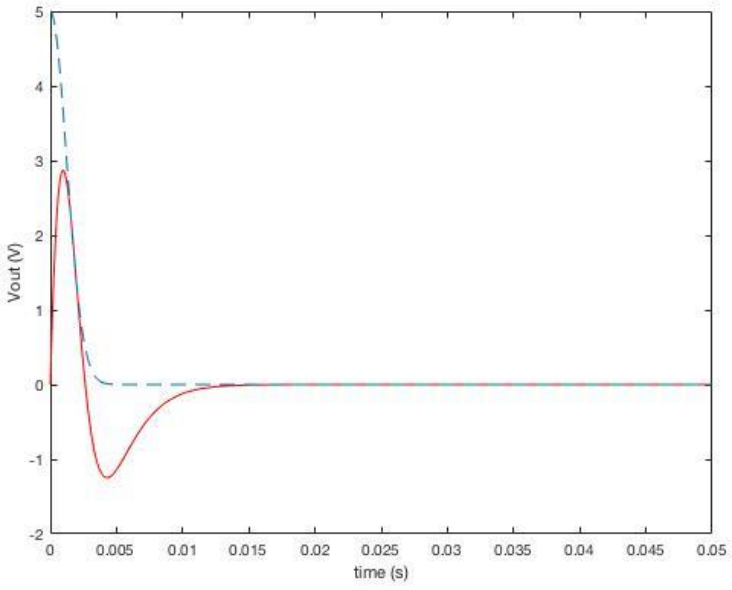

Figure 61: Impulsive signal with decay at critical damping

3. V_{in} = square wave with amplitude V_{in} = 5V

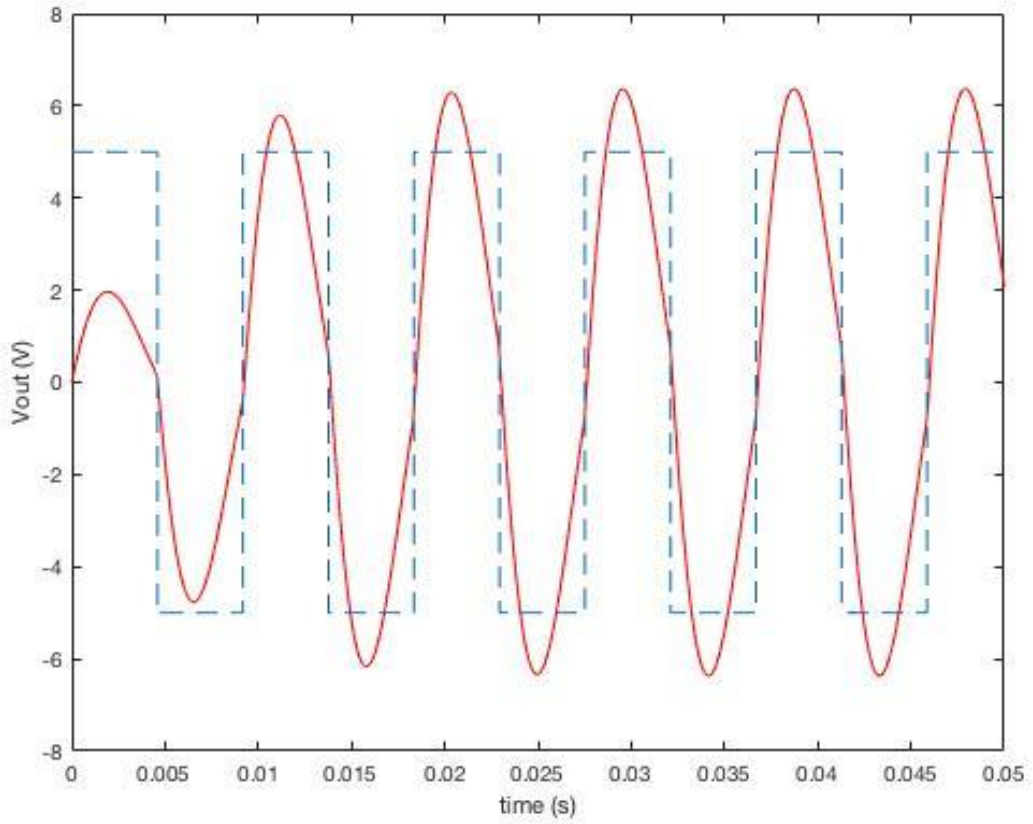

Figure 62: 109Hz square wave

While investigating periodic functions, it is worthwhile to calculate resonant frequency for analysis:

$$\omega_0 = \frac{1}{\sqrt{LC}} \; and \; f_0 = \frac{1}{2\pi\sqrt{LC}}$$

For our circuit, it is calculated that for resonance to occur, the input frequency must be approximately $109.8Hz$. Since the input for the circuit in Fig 2.3.1 is $109Hz$, the resonant and input frequencies are close, even though the amplitude is not maximum, resonance does occur, as shown by $Vout > Vin$ at some points on the graph.

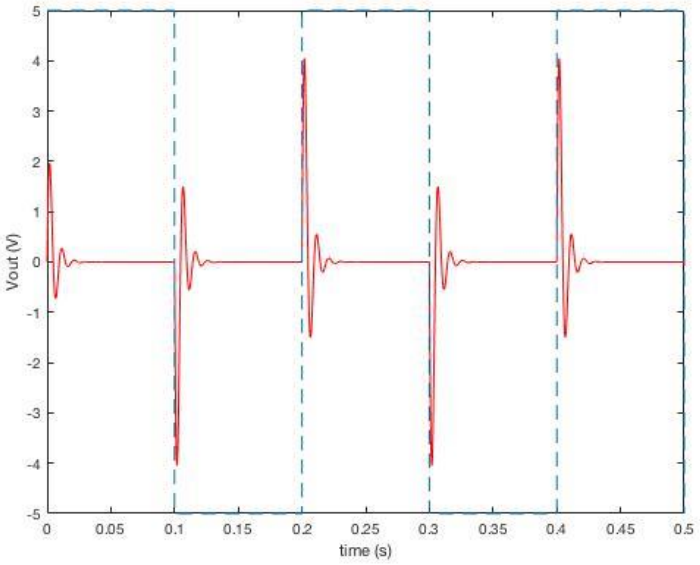

Figure 63: 5Hz square wave

When the input frequency is set to 5Hz, much less than the resonant frequency, the circuit can achieve steady state value

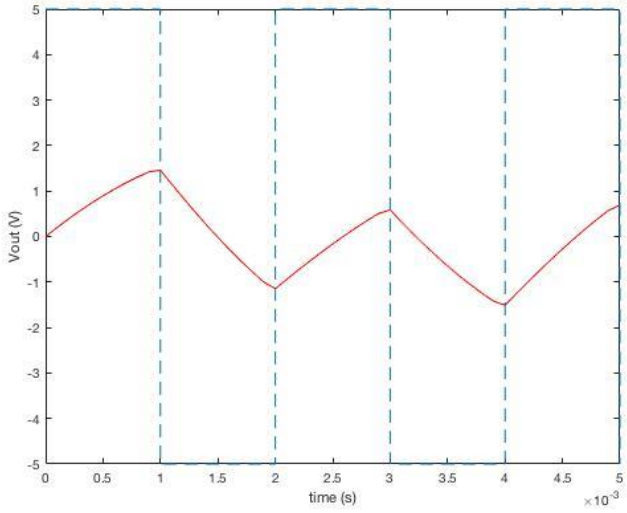

Figure 64: 500Hz square wave

When the input frequency is set to 500Hz, much higher than the resonant frequency, Vin switches while $Vout$ is in transient state and before it reaches a steady state value, which can be explained physically by the capacitor charging and discharging.

4. V_{in} = Sine wave with amplitude V_{in} = 5V

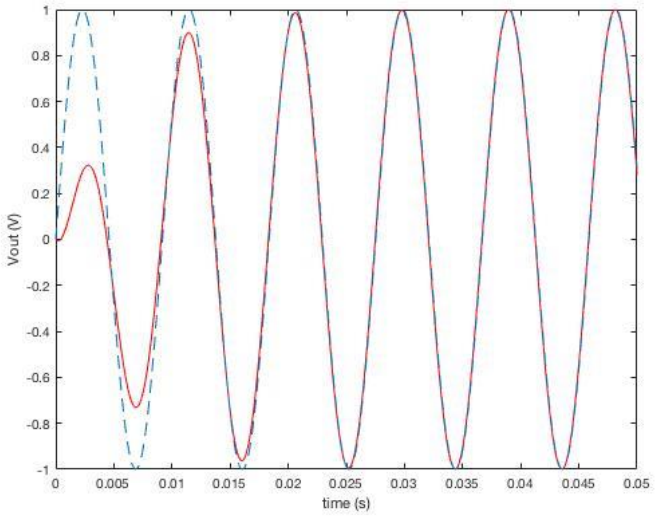

Figure 65: Sine wave with input frequency as 109Hz

For a sinusoid input waveform, the output waveform is expected to be a sinusoid, matching in shape with the input waveform. As demonstrated in part 3, since $109Hz$ is comparable with the resonance frequency, resonance is present, even though amplitude is not completely maximised. After the circuit reaches steady state value, the output waveform matches closely to the input waveform.

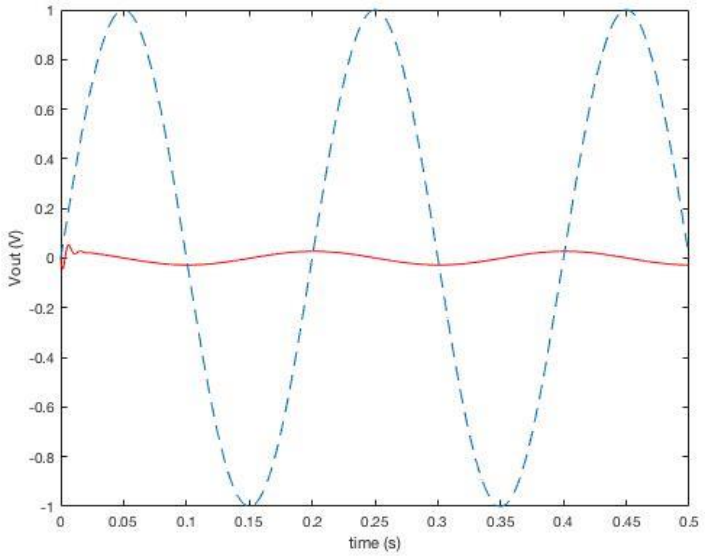

Figure 66: Sine wave with input frequency as 5Hz

If we set V_{in} to an input of 5Hz, much below the resonant frequency, we observe an output voltage with:

1. Output frequency is the same as input frequency
2. Output Voltage is much less than the input voltage
3. There is a phase difference between output and input waveforms

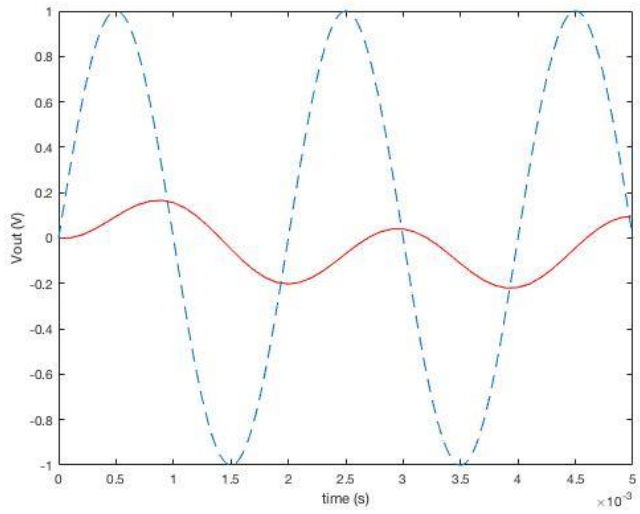

Figure 67: Sine wave with input frequency as 500Hz

In the above case, the input frequency is much higher than resonant frequency, so we observe again:

1. Input Frequency is equal to Output Frequency
2. Phase difference between input and output waveforms
3. V_{out} changes rapidly before circuit achieves steady state

3. Relaxation

The approaches outlined so far will only work for Initial Value Problem (IVP) in which all the initial conditions are given for the same value of the independent variable. In Boundary Valued Problems (BVP); however, the initial conditions are not given for the same value of the independent variable. In such cases, solutions might not even exist and even if they do, we cannot use the approaches for solving IVP (e.g. the RK method) since there is no guarantee that the desired final boundary value will be reached.

In BVP, the method of Finite Differences can be used to compute an estimation for the solution of the differential equation. Effectively, the method of finite differences breaks up the interval in question into equal segments and then uses the equal segments to develop a set of simultaneous equations. The solution of these equations will give an approximation to the solution of the differential equation.

Developing Tools

Before using the Finite Difference method, we must develop some tools. Specifically, we need to find some estimations for the first and second derivative of a function.

The derivative of a function f at x_0 is

$$f'(x_0) = \lim_{h\to\infty} \frac{f(x_0+h) - f(x_0)}{h} \tag{3}$$

Given that h is small, an obvious approximation of $f'(x_0)$ is

$$\frac{f(x_0+h) - f(x_0)}{h} \tag{4}$$

This formula is known as the forward difference if $h > 0$ and the backward difference if $h < 0$.

We can also develop another more accurate approximation of $f'(x_0)$ using the Taylor Series. Using the Taylor series for $f(x)$ around the point x_0 we can estimate $f(x_0+h)$ and $f(x_0-h)$ to be

$$f(x_0+h) = f(x_0) + f'(x_0)h + \frac{f^{(2)}(x_0)h^2}{2!} + \frac{f^{(3)}(x_0)h^3}{3!} + \cdots \tag{5}$$

$$f(x_0-h) = f(x_0) - f'(x_0)h + \frac{f^{(2)}(x_0)h^2}{2!} - \frac{f^{(3)}(x_0)h^3}{3!} - \cdots \tag{6}$$

Subtract the two equations above to get

$$f(x_0 + h) - f(x_0 - h) = 2f'(x_0) + \frac{2f^{(3)}(x_0)h^3}{3!} + \cdots$$

$$f'(x_0) = \frac{f(x_0 + h) - f(x_0 - h)}{2h} - \frac{f^{(3)}(x_0)h^2}{3!} + \cdots$$

$$f'(x_0) = \frac{f(x_0 + h) - f(x_0 - h)}{2h} + Error$$

Where the error is $O(h^2)$.

The same can be done for the second derivative; however, instead of subtracting equation (5) from equation (6), we add them to get.

$$f^{(2)}(x_0) = \frac{f(x_0 + h) - 2f(x_0) + f(x_0 - h)}{h^2} + Error$$

Where the error again is $O(h^2)$

These equations can also be extended to partial derivative. Thus, for a function $u(x, y)$

$$\frac{\partial u}{\partial x} \approx \frac{u(x + h, y) - u(x - h, y)}{2h}$$

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

(7)

Finite Differences

Given that we have the Laplace Equation

$$\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} = 0$$

(8)

We divide the x and y intervals into N segments of equal length h where

$$h = \frac{1}{N}$$

Let $x_i = ih$ and $y_j = jh$. Thus

$$x_0 = 0, x_1 = h, x_2 = 2h \ldots$$

(9)

$$y_0 = 0, y_1 = h, y_2 = 2h \ldots$$

(10)

Furthermore, let $U_i^j = u(x_i + y_j)$

Using (7) we can rewrite (8) as

$$\frac{U_{i+1}^{j} - 2U_i^j + U_{i-1}^j}{h^2} + \frac{\left(U_i^{j+1} - 2U_i^j + U_i^{j-1}\right)}{h^2} = 0$$

$$U_{i+1}^{j} + U_{i-1}^{j} + U_i^{j+1} + U_i^{j-1} - 4U_i^j = 0$$

(11)

In the simplest form, we are interest in the values of Laplace's equation on a rectangle taken as $0 \leq x \leq a$ and $0 \leq y \leq b$. We divide x and y according to equations (9) and (10) respectively. To solve the equation, the values of $u(x, y)$ must be known on the boundaries $u(x, b) = \Phi_1(x), u(a, y) = \Phi_2(y), u(x, 0) = \Phi_3(x), u(0, y) = \Phi_4(y)$

The boundary functions give us the values of U_i^j on the edges of the rectangle. For example, if we let $a = 1$, $b = 1$ and $N = 3$ for simplicity. Using equation (11) we can generate a system of equations

$$E1: U_2^1 + U_1^2 - 4U_1^1 = -(U_0^1 + U_1^0)$$

$$E2: U_1^1 + U_2^2 - 4U_2^1 = -(U_2^0 + U_3^1)$$

$$E3: U_1^1 + U_2^2 - 4U_1^2 = -(U_0^2 + U_1^3)$$

$$E4: U_1^2 + U_2^1 - 4U_2^2 = -(U_2^3 + U_3^2)$$

This can also be written in matrix form $A\boldsymbol{x} = \boldsymbol{b}$

$$\begin{bmatrix} -4 & 1 & 1 & 0 \\ 1 & -4 & 0 & 1 \\ 1 & 0 & -4 & 1 \\ 0 & 1 & 1 & -4 \end{bmatrix} \begin{bmatrix} U_1^1 \\ U_2^1 \\ U_1^2 \\ U_2^2 \end{bmatrix} = - \begin{bmatrix} U_0^1 + U_1^0 \\ U_2^0 + U_3^1 \\ U_0^2 + U_1^3 \\ U_2^3 + U_3^2 \end{bmatrix}$$

(12)

This matrix can then be solved using matrix methods. However, for large matrices it gets computationally expensive.

Jacobi and Gauss-Siedel Iterative methods

Iterative methods are used to matrix systems with a high number of zero entries since in these cases, they are often more efficient than matrix methods (e.g. Gaussian elimination).

An iterative method solves an $n \times n$ linear system in the form $A\boldsymbol{x} = \boldsymbol{b}$ by first estimating an initial solution $\boldsymbol{x}^0$ (basically a guess) and then generating a sequence of vectors which represent more accurate solutions until some maximum error tolerance is reached.

The Jacobi iterative method generates the component x_i^k of $\boldsymbol{x}^k$ from the components of $\boldsymbol{x}^{k-1}$ using

$$x_i^k = \frac{1}{a_{ii}}\left(-\sum_{\substack{j=1 \\ j\neq i}}^{n} \left(a_{ii}x_j^{k-1}\right) + b_i \right)$$

We continue to do this until we generate a solution where

$$\frac{\|\boldsymbol{x}^k - \boldsymbol{x}^{k-1}\|_\infty}{\|\boldsymbol{x}^k\|_\infty} < percetage\ error\ tolerance$$

Where $\|\boldsymbol{x}\|_\infty = \max(|x_1|, \dots, |x_n|)$ (i.e. the maximum norm of the vector)

The Gauss-Siedel method is an improvement to the Jacobi. It comes from the observation that we only use the elements of $\boldsymbol{x}^{k-1}$ to computer x_i^k. However, for $i > 1$ we have already computer some of the components of $\boldsymbol{x}^k$ which are supposed to be better approximation of the actual solutions. Thus, we would use them to compute x_i^k. The method now becomes

$$x_i^k = \frac{1}{a_{ii}}\left(-\sum_{j=1}^{i-1} \left(a_{ii}x_j^k\right) - \sum_{j=i+1}^{n} \left(a_{ii}x_j^{k-1}\right) + b_i \right)$$

The Guass-Siedel method is ideal for solving the matrix equation (12) that we generated above since for each row we can only have a maximum of 4 non-zero columns. Effectively, the algorithm now becomes

$$\left(U_i^j\right)_{new} = \left(U_i^j\right)_{old} + r_i^j$$

Where the residual r_i^j is given by

$$r_i^j = \frac{U_{i+1}^i + U_{i-1}^j + U_i^{(j+1)} + U_i^{j-1} - 4U_i^j}{4}$$

In this case, r_i^j is similar to the $percetage\ error\ tolerance$ that we defined above be we didn't divide it.

Exercise 4

In this exercise, we effectively use the Guass-Siedel method to plot the solution to Laplace's equation given different initial condition.

The code for exercise 4 is shown below

```
%clean from previous run
close all;
clear;
clc;
```

```
%Choose gridsize and residual size tolerance
h = 0.05;
max_res = 1e-6;

%Choose the border functions
fi1 = @(x) 1;
fi2 = @(y) 0;
fi3 = @(x) heaviside(x-0.2)-heaviside(x-0.8);
fi4 = @(y) 0;

%Setting up
x(1) = 0;
xf = 1;
yf = 1;
n = round((xf - x(1))/h);
x = 0:h:n*h;
y = 0:h:n*h;
z = zeros(length(x),length(y));


sum = 0;

for i = 1:length(x)
    z(i,length(y)) = fi1(x(i));
    z(length(x),i) = fi2(y(i));
    z(i,1) = fi3(x(i));
    z(1,i) = fi4(y(i));
end

%Calculate the average value
total = 4*(n + 1) - 4;
avg = sum/total;

%Set all the inner points as this average value
for i = 2:length(x) - 1
    for j = 2:length(y) - 1
            z(i,j) = avg;
    end
end

while (true)
    %set the maximum residue found in this iteration to 0
    %it will be updated in the for loop
    r_max = 0;
    for i = 2:length(x) - 1
        for j = 2:length(y) -1
            %Calculate the residue
            r = (z(i+1,j) + z(i-1,j) + z(i,j+1) + z(i,j-1) -
4*z(i,j))/4;
            %Set the new value of the point
            z(i,j) = z(i,j) + r;
```

```
            if(abs(r) > r_max)
                r_max = abs(r);
            end
        end
    end
    %if the meximum residue value found in this iteration is smaller
    %than our maximum tolerence break
    if(r_max < max_res)
        break;
    end
end

%It is important not to forget to transpose because of the way matlab
%handles matrices
mesh(x,y,z');
xlabel('x-axis');
ylabel('y-axis');
zlabel('z-axis');
title('Solution to Laplace Equation');
```

For the boundary conditions given

$u(0,y) = u(1,y) = 0$, for $0 < y < 1$

$u(x,1) = 0$, for $0 < x < 1$

$u(x,0) = 0$, for $0 < x < 0.2$ and $0.8 < x < 1$

$u(x,0) = 1$, for $0.2 < x < 0.8$

With $h = 0.02$ and $residue = 10^{-4}$, the following plot was generated

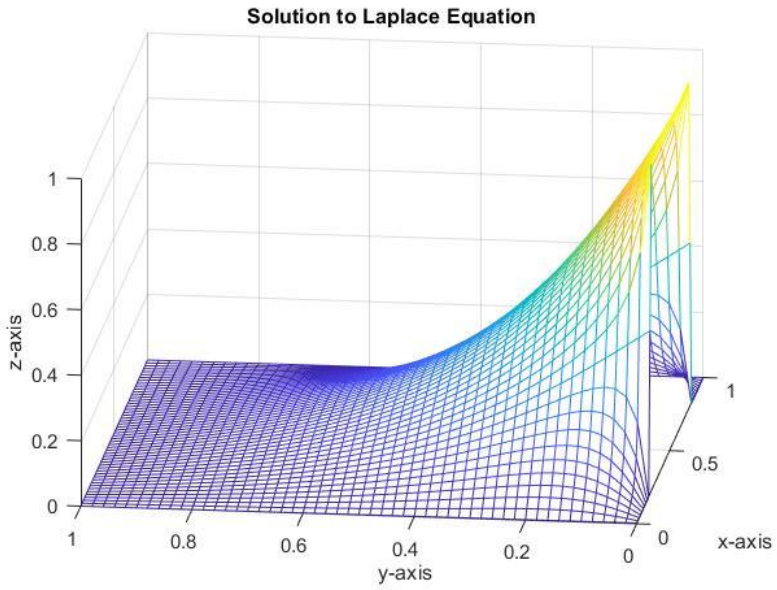

Figure 68

An interesting observation is how the $residue$ affects the overall final shape of the plot. We tried for 3 different residue values

With $h = 0.02$ and $residue = 10^{-2}$, the following plot was generated

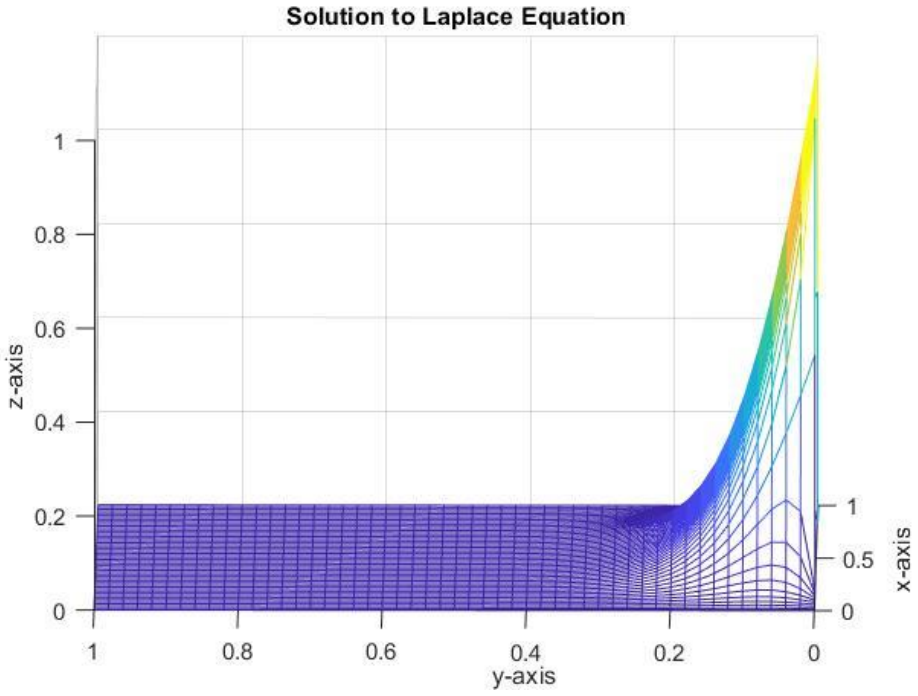

Figure 69

With $h = 0.02$ and $residue = 10^{-4}$, the following plot was generated

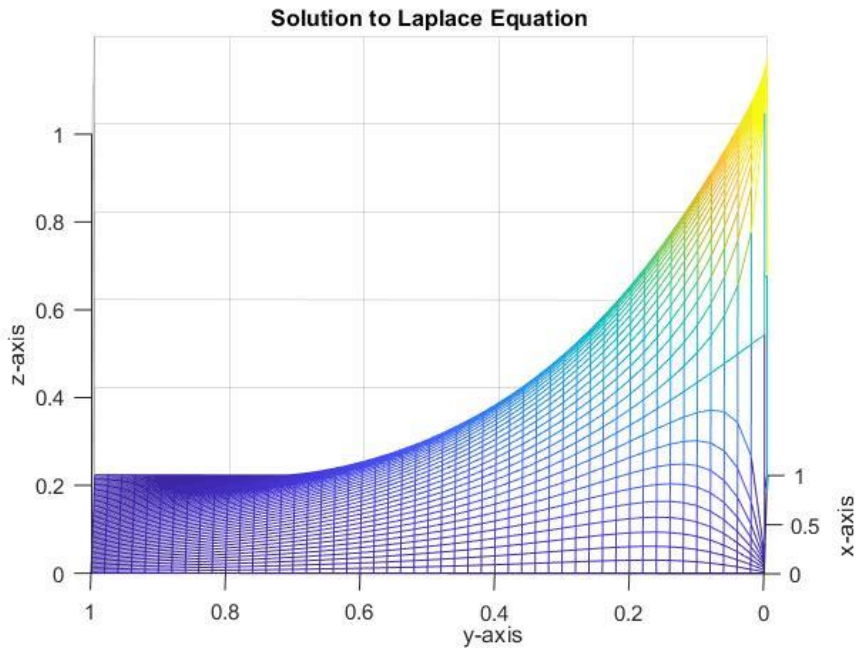

Figure 70

With $h = 0.02$ and $residue = 10^{-6}$, the following plot was generated

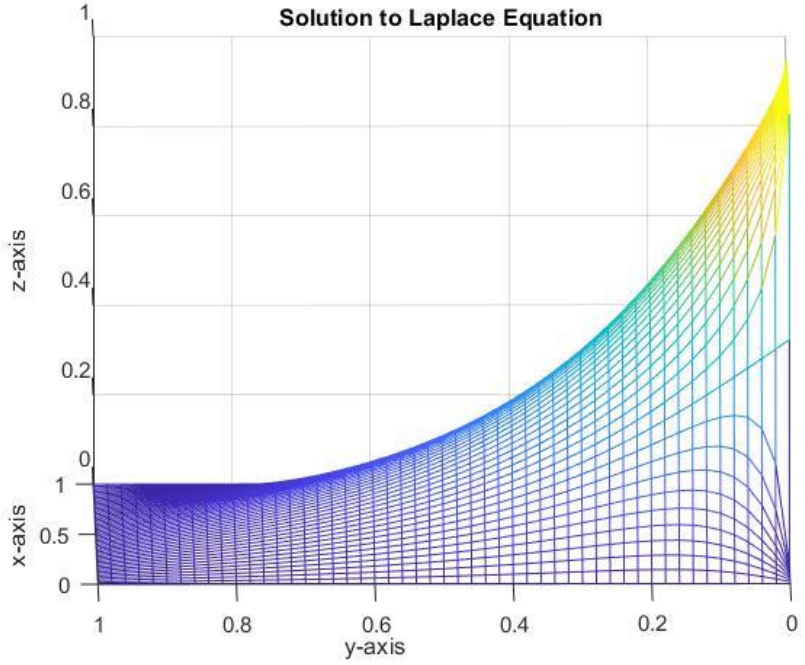

Figure 71

From the specify angle, we can see that as we decrease the $residue$, because the plot becomes more accurate, it takes longer for the graph to go to zero (i.e. the slope of the graph decreases since 2 adjacent points have values that are closer to each other).

We also tried to estimate the solution for more complex boundary conditions.

For the boundary conditions given by

$u(0,y) = u(1,y) = 0\ ,$ for $0 < y < 1$

$u(x,1) = \sin(6\pi x)\ ,$ for $0 < x < 1$

$u(x,0) = 0,$ for $0 < x < 1$

With $h = 0.02$ and $residue = 10^{-6}$, the following plot was generated

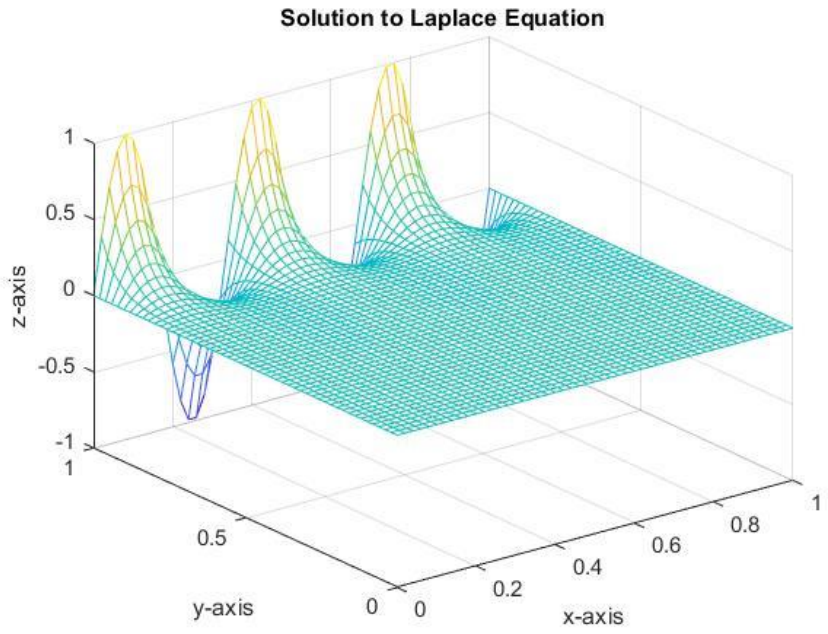

Figure 72

For the boundary conditions given by

$u(0,y) = u(1,y) = \sin(\pi y)\ ,$ for $0 < y < 1$

$u(x,1) = u(x,0) = \sin(\pi x)\ ,$ for $0 < x < 1$

With $h = 0.02$ and $residue = 10^{-6}$, the following plot was generated

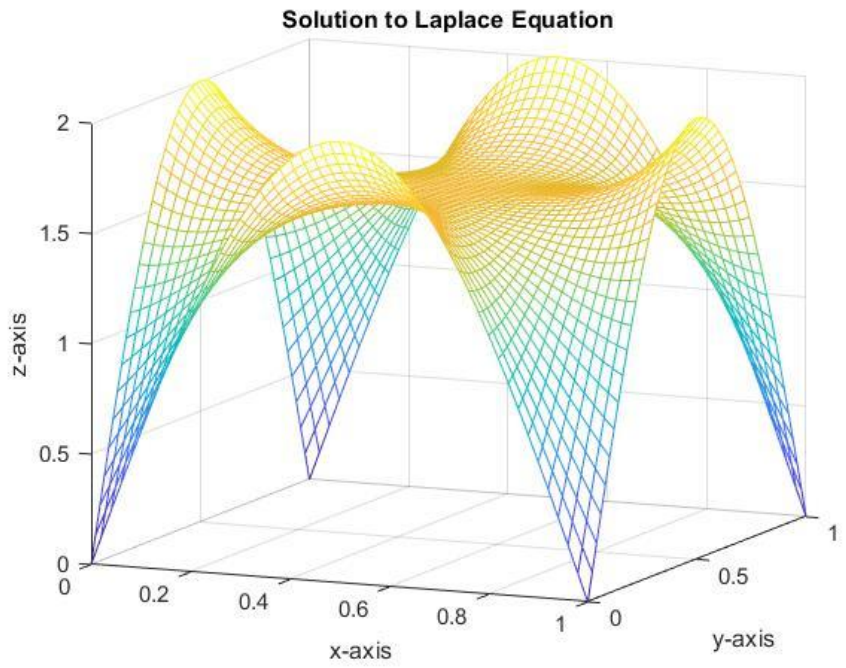

Figure 73

When the boundary conditions do not match at the corners, depending on which boundary it is, the code will select one of the two values provided. An example of this is shown below

For the boundary conditions given by

$u(0,y) = u(1,y) = 0,$ for $0 < y < 1$

$u(x,1) = 1\,,$ for $0 < x < 1$

$u(x,0) = 0,$ for $0 < x < 1$

With $h = 0.04$ and $residue = 10^{-6}$, the following plot was generated

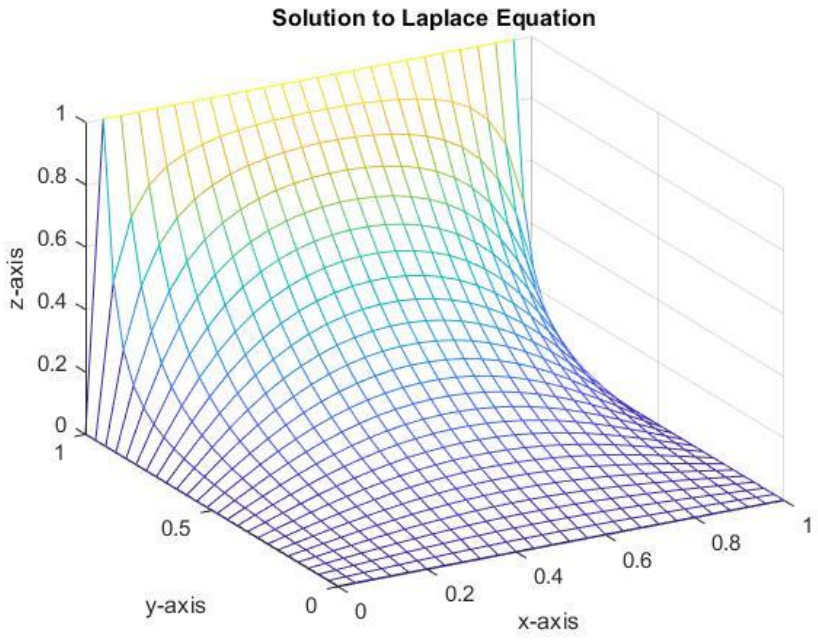

Figure 74

As we can see, the discontinuity in the boundary conditions doesn't really affect the plot since the corners are never considered in the estimation of the other points.

Exercise 5

This method can also be extended nicely to the Poison equation which has the form

$$\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} = g(x, y)$$

By simply setting the residual r_i^j to be

$$r_i^j = \frac{U_{i+1}^i + U_{i-1}^j + U_i^{(j+1)} + U_i^{j-1} - 4U_i^j - h^2 g_i^j}{4}$$

The code for exercise 5 (i.e. relaxation2.m) is shown below:

```
close all;
clear;
clc;

%Choose gridsize and residual size
h = 0.02;
max_res = 1e-3;

%Choose the border functions
fi1 = @(x) -2*x^2 + 3*x + 2;
```

```
fi2 = @(y) -y^2 + 3*y + 1;
fi3 = @(x) -2*x^2 + x + 2;
fi4 = @(y) -y^2 + y + 2;

%Choose forcing term and the actual function to compare
g = @(x,y) -6;

%Setting up
x(1) = 0;
xf = 1;
yf = 1;
n = round((xf - x(1))/h);
x = 0:h:n*h;
y = 0:h:n*h;
z = zeros(length(x),length(y));

%Calculating the actual values
[X, Y] = meshgrid(0:h:n*h);
% The U function represents the solution to the Poison Equation
U = -2*X.^2 - Y.^2 + X + Y + 2*X.*Y + 2;

sum = 0;

for i = 1:length(x)
    z(i,length(y)) = fi1(x(i));
    z(length(x),i) = fi2(y(i));
    z(i,1) = fi3(x(i));
    z(1,i) = fi4(y(i));
end

%Calculate the average value
total = 4*(n + 1) - 4;
avg = sum/total;

%Set all the inner points as this average value
for i = 2:length(x) - 1
    for j = 2:length(y) - 1
            z(i,j) = avg;
    end
end

while (true)
    %set the maximum residue found in this iteration to 0
    %it will be updated in the for loop
    r_max = 0;
    for i = 2:length(x) - 1
        for j = 2:length(y) -1
            %Calculate the residue
            r = (z(i+1,j) + z(i-1,j) + z(i,j+1) + z(i,j-1) -
h^2*g(x(i),y(j)) - 4*z(i,j))/4;
            %Set the new value of the point
```

```
            z(i,j) = z(i,j) + r;
            if(abs(r) > r_max)
                r_max = abs(r);
            end
        end
    end
    %if the meximum residue value found in this iteration is smaller
    %than our maximum tolerence break
    if(r_max < max_res)
        break;
    end
end

%It is important not to forget to transpose because of the way matlab
%handles matrices
%Uncomment the following line to get the estimation for the solution
%mesh(x,y,z');

%Plot error
er = abs(z' - U); mesh(X,Y,er);
xlabel('x-axis');
ylabel('y-axis');
zlabel('z-axis');
title('Solution to the Poison Equation');
```

We only change a few lines to generate the solution for the Poison equation! Specifically, the highlighted lines indicate the changes that have been made to the code.

In this case, we also generate an error plot which shows the error made in calculating each point on the graph. We can choose between plotting the estimation and the error by just changing two lines.

With $h = 0.02$ and $residue = 10^{-3}$, the following estimation plot was generated

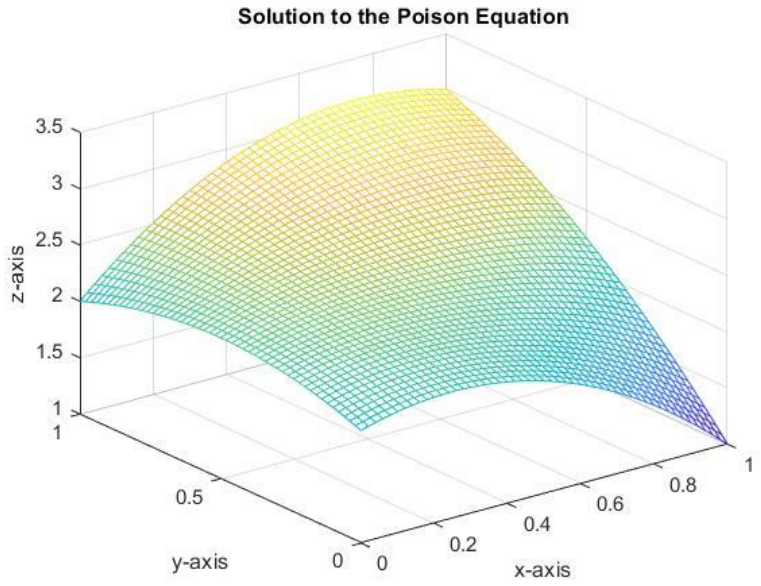

Figure 75

For the same inputs, the following error plot was generated

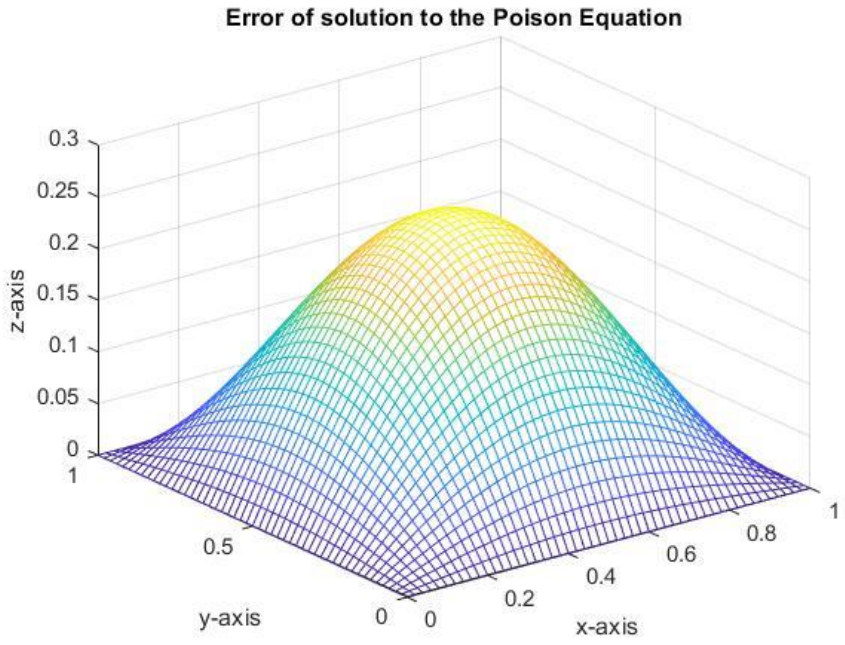

Figure 76

As we can see from figure 9, the error gets larger as we move closer to the center of the plot. Initially, this might seem sort of strange; however, we must realize that the estimations made for the points in the center are themselves based on points closer to the edges which are also just estimations in themselves. The closer to the center a point is, the more estimations the point was based on. Hence the points closer to the center will have the highest error.

Another observation is the maximum error is much larger than the $residue$ that we set. However, the residue assumes that the points nearby the point we are estimating are the correct values which isn't true in this case.

Nevertheless, we can see how the $residue$ affects the error of the plot in the following figures

With $h = 0.02$ and $residue = 10^{-2}$, the following plot was generated

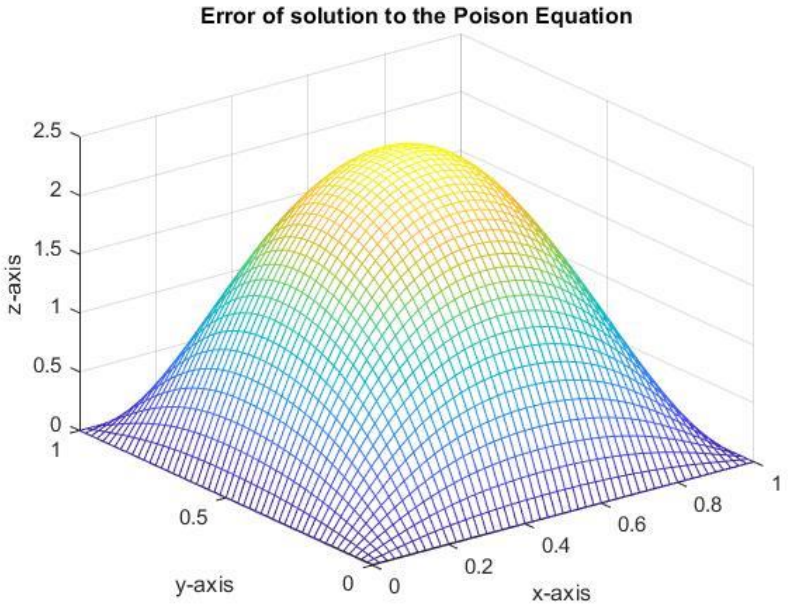

Figure 77

With $h = 0.02$ and $residue = 10^{-4}$, the following plot was generated

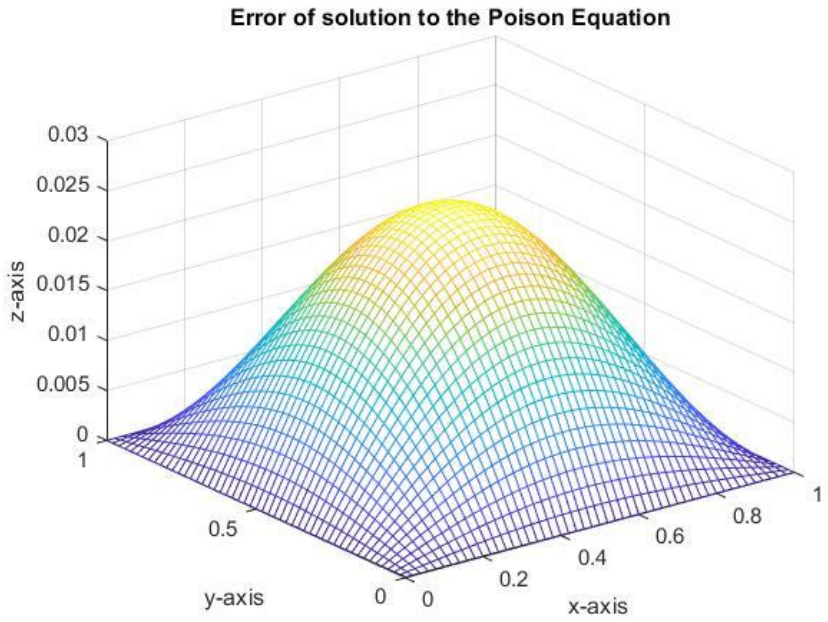

Figure 78

With $h = 0.02$ and $residue = 10^{-6}$, the following plot was generated

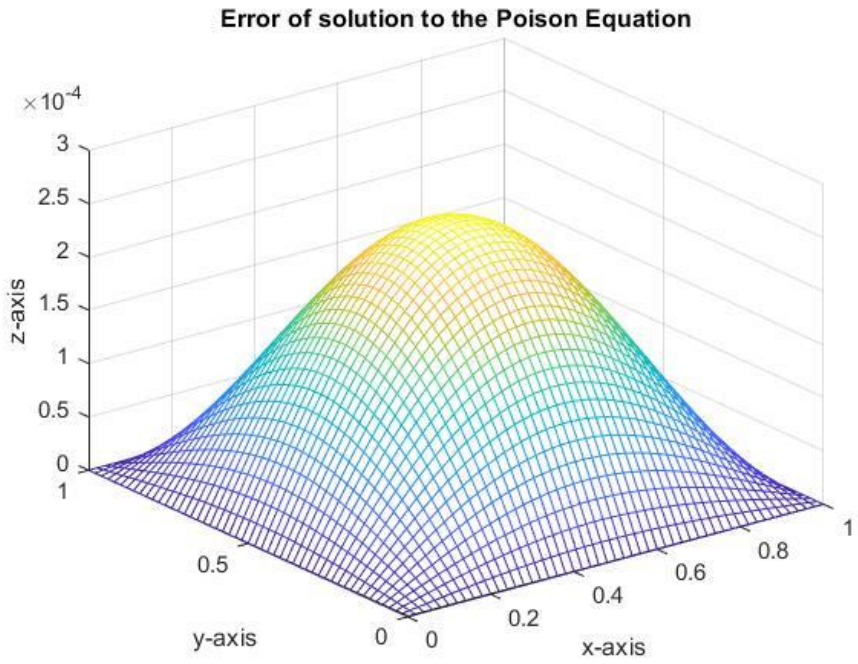

Figure 79

As we can see, it seems that the error at each point is proportional to the $residue$.

Bonus

So far, we have been assuming that the best way to get our approximations to converge to the actual solution is by setting the $residue$ (i.e. the difference between our estimation and the true solution) to be equation to zero.

For example, in Exercise 4, we set the $residual$ to be

$$r_i^j = \frac{U_{i+1}^i + U_{i-1}^j + U_i^{(j+1)} + U_i^{j-1} - 4U_i^j}{4}$$

We then set the new value to be

$$\left(U_i^j\right)_{new} = \left(U_i^j\right)_{old} + r_i^j \qquad (13)$$

Thus, we are essentially choosing $\left(U_i^j\right)_{new}$ such that the $residual$ is zero. However, choosing $\left(U_i^j\right)_{new}$ such that the new $residual$ is zero is not necessarily the most efficient way to converge the value of $\left(U_i^j\right)$ to the actual solution. We can modify equation (13) to

$$\left(U_i^j\right)_{new} = \left(U_i^j\right)_{old} + \omega r_i^j$$

Where ω is positive. If $0 < \omega < 1$ then the procedure is called under relaxation method and if $1 < \omega$ then the procedure is called over relaxation method (SOR). The over relaxation methods are used to accelerate the convergence of the Guass-Siedel technique.

It can be proven that for SOR methods, can converge only if $0 < \omega < 2$[1]

Exercise 4 with SOR

The code for exercise 4 and 5 can be adjusted to implement SOR by simply changing a few lines. The new code for exercise 4 and 5 with SOR is shown below

```
close all;
clear;
clc;

%Choose gridsize and residual size
h = 0.02;
max_res = 1e-4;
omega = 1.0;

%Choose the border functions
fi1 = @(x) 0;
fi2 = @(y) 0;
fi3 = @(x) heaviside(x-0.2)-heaviside(x-0.8);
```

[1] Add reference

```
fi4 = @(y) 0;

%Choose forcing term and the actual function to compare
g = @(x,y) 0;

%Setting up
x(1) = 0;
xf = 1;
yf = 1;
n = round((xf - x(1))/h);
x = 0:h:n*h;
y = 0:h:n*h;
z = zeros(length(x),length(y));

%Calculating the actual values
[X, Y] = meshgrid(0:h:n*h);
% The U function represents the solution to the Poison Equation
%U = -2*X.^2 - Y.^2 + X + Y + 2*X.*Y + 2;

sum = 0;

for i = 1:length(x)
    z(i,length(y)) = fi1(x(i));
    z(length(x),i) = fi2(y(i));
    z(i,1) = fi3(x(i));
    z(1,i) = fi4(y(i));
end

%Calculate the average value
total = 4*(n + 1) - 4;
avg = sum/total;

%Set all the inner points as this average value
for i = 2:length(x) - 1
    for j = 2:length(y) - 1
            z(i,j) = avg;
    end
end

iteration = 0;
while (true)
    %set the maximum residue found in this iteration to 0
    %it will be updated in the for loop
    r_max = 0;
    for i = 2:length(x) - 1
        for j = 2:length(y) -1
            %Calculate the residue
            r = (z(i+1,j) + z(i-1,j) + z(i,j+1) + z(i,j-1) -
h^2*g(x(i),y(j)) - 4*z(i,j))/4;
            %Set the new value of the point
            z(i,j) = z(i,j) + omega*r;
```

```
            if(abs(r) > r_max)
                r_max = abs(r);
            end
        end
    end
    %if the meximum residue value found in this iteration is smaller
    %than our maximum tolerence break
    if(r_max < max_res)
        break;
    end
    iteration = iteration + 1;
end

%It is important not to forget to transpose because of the way matlab
%handles matrices
%Uncomment the following line to get the estimation for the solution
mesh(x,y,z');

%Plot error
%er = abs(z' - U); mesh(X,Y,er);
xlabel('x-axis');
ylabel('y-axis');
zlabel('z-axis');
title('Solution to the Laplace Equation with SOR');
```

The only thing that has changed is that we now added the omega variable and an iteration variable so that we can compare different values of omega so see which one converges faster.

For the comparisons, we kept the h and $residue$ values constant with $h = 0.02$ and $residue = 10^{-4}$

Finally, to make comparing the different values a bit easier, we ran the script shown below

```
myIterations = zeros(1,20);
myOmegas = zeros(1,20);

for myI = 0:19
    %Set the new value of Omega
    omega = 1+ 0.05*myI;
    %Run the sor script
    sor
    %Add iteration to the myIterations array and the Omegas to the
myOmegas
    %array
    myIterations(myI+1) = iteration;
    myOmegas(myI+1) = omega;
end

%Plot
plot(myOmegas,myIterations);
xlabel('Omega value');
```

```
ylabel('Number of iteration');
title('Omega vs Number of iterations');
```

Effectively, all it does is run the sor script for different values of ω it then records the numbers of iterations that were needed for the $residue$ to be satisfied and plots the results.

For exercise 4, we managed to plot the following

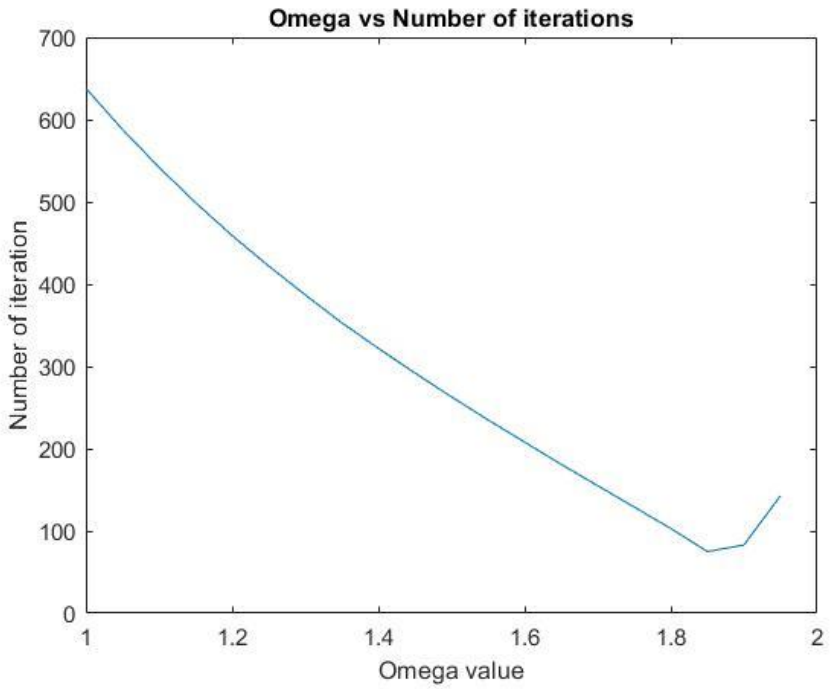

Figure 80

From the figure above, we can see that there is a certain range of ω values for which the number of iterations was significantly decreased (in this case it's around $\omega = 1.85$)

Exercise 5 with SOR

If we do the same process for exercise 5, we the following plot

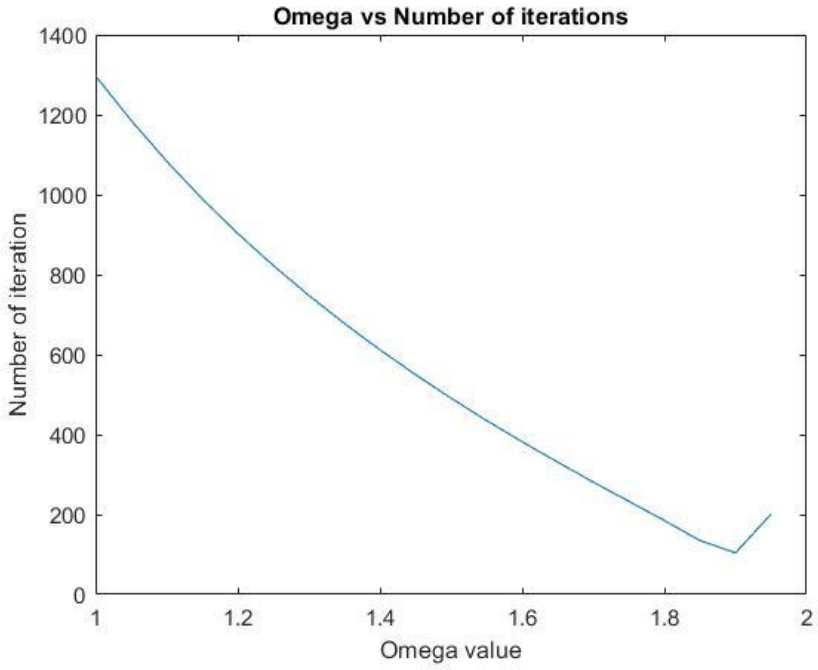

Figure 81

The two plots have a similar shape! This is because, in both cases, we are trying to derive an approximation to the linear system defined by $A\boldsymbol{x} = \boldsymbol{b}$ and by changing the boundary functions and the forcing term, we are only changing $\boldsymbol{b}$ while A remains the same. It seems reasonable that changes in the values of $\boldsymbol{b}$ shouldn't affect the shape of the graph.

We can try to confirm this by changing the value of h which will effectively change A

With $h = 0.1$ and $residue = 10^{-6}$, the following plot was generated

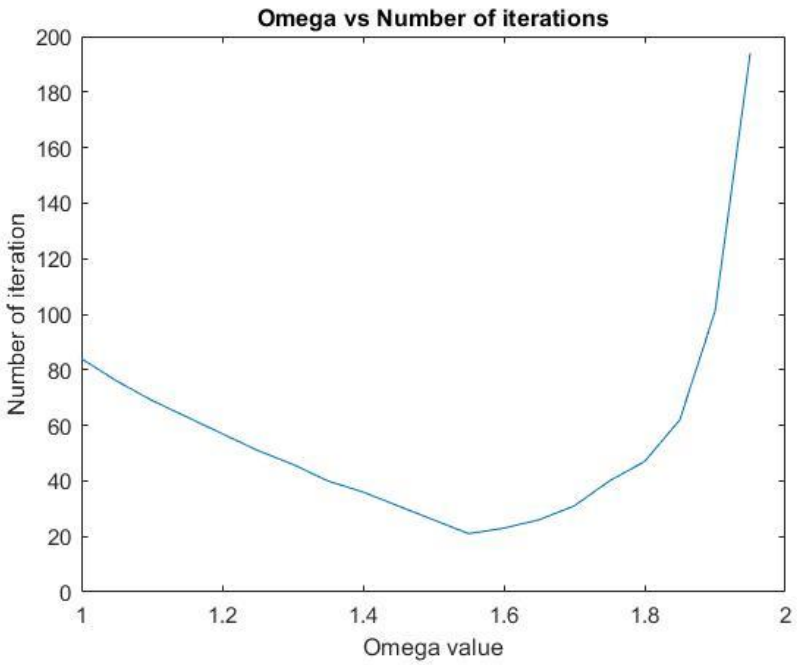

Figure 82

As we can see the shape of the graph has changed and now the minimum amount of iterations seems to have occurred at about $\omega = 1.55$

It is also interesting to observe how the choice of ω affects the error of the approximation. This is illustrated in the following graphs in which the h and $residue$ values were kept constant at $h = 0.02$ and $residue = 10^{-4}$.

With $\omega = 1$ following plot was generated

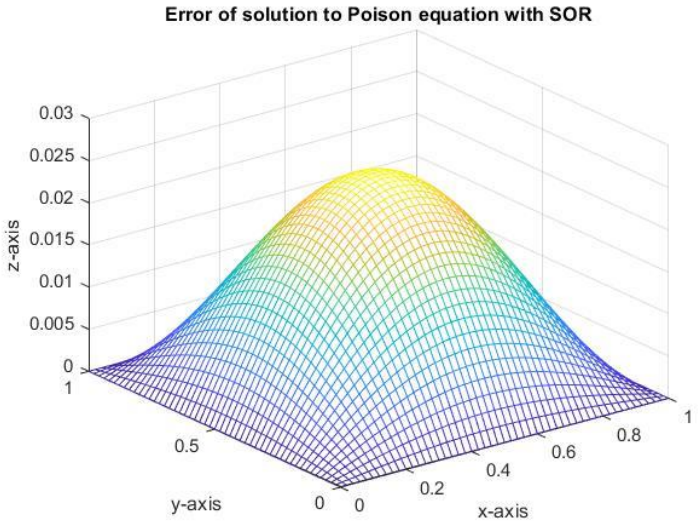

Figure 83

With $\omega = 1.2$ following plot was generated

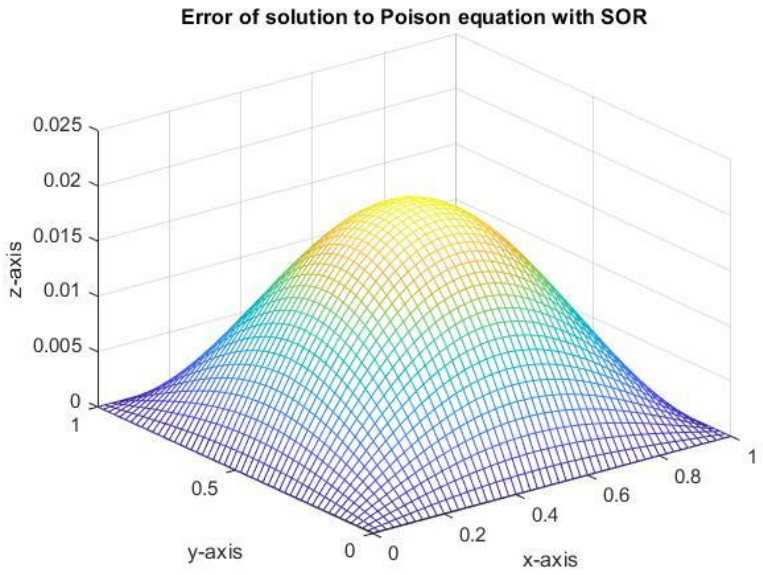

Figure 84

With $\omega = 1.4$ following plot was generated

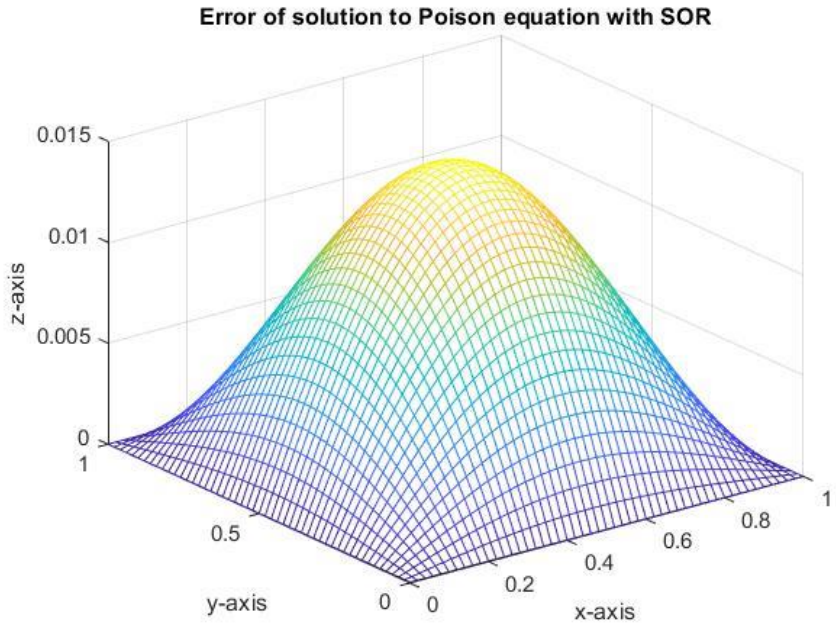

Figure 85

With $\omega = 1.6$ following plot was generated

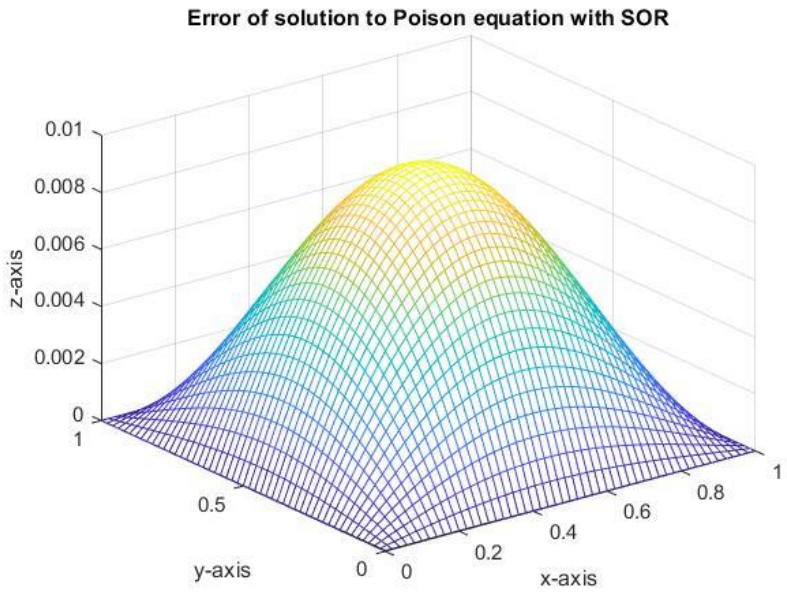

Figure 86

With $\omega = 1.87$ following plot was generated

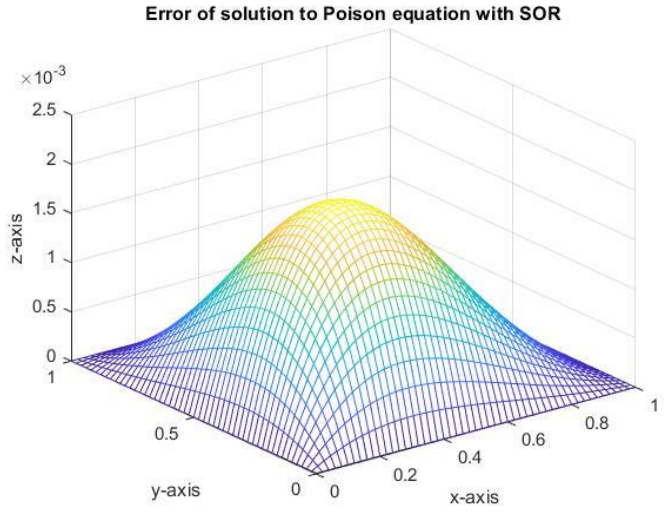

Figure 87

While with $\omega = 1.9$ which is only slightly above what we estimated to be our ω value for the minimum amount of iterations, this plot was generated

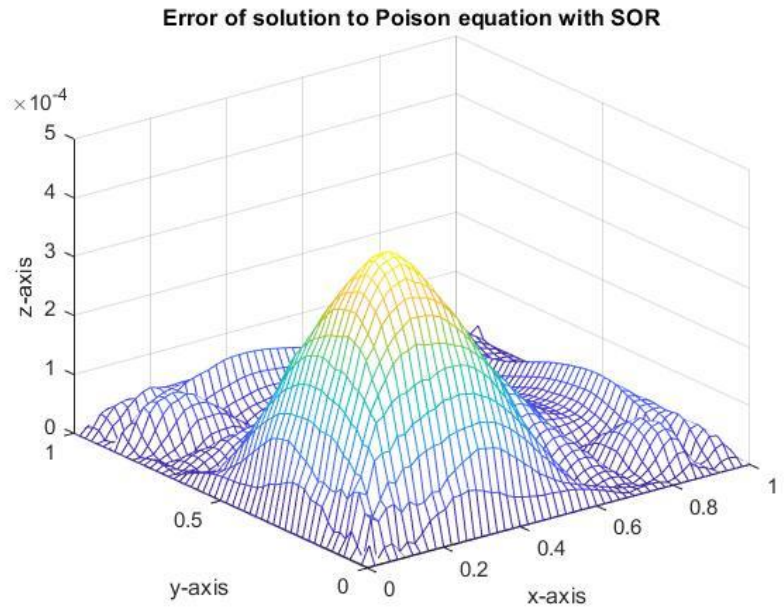

Figure 88

With $\omega = 1.92$ following plot was generated

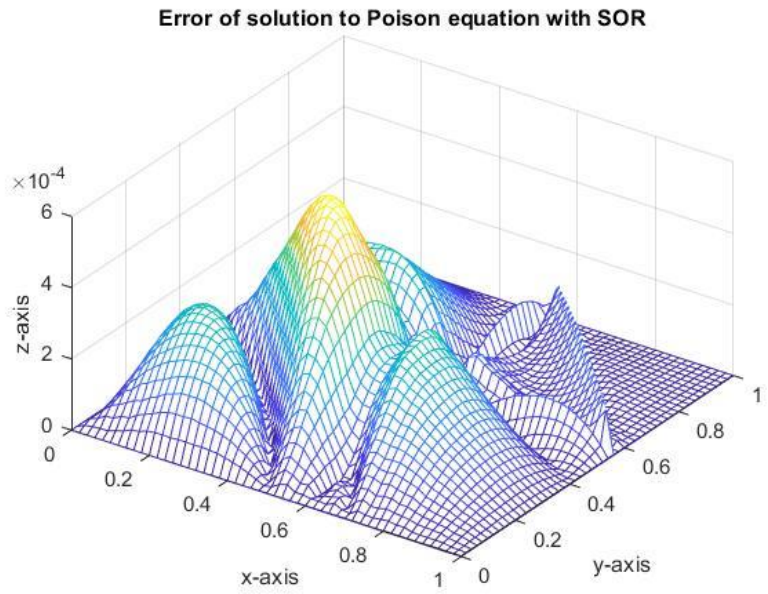

Figure 89

As we can see some sort of *instability* issue has occurred once exceed the best ω. This might be caused because of an overestimation of the point.

Finally, for with $\omega = 1.99$ (something near the max of 2) the following plot was generated

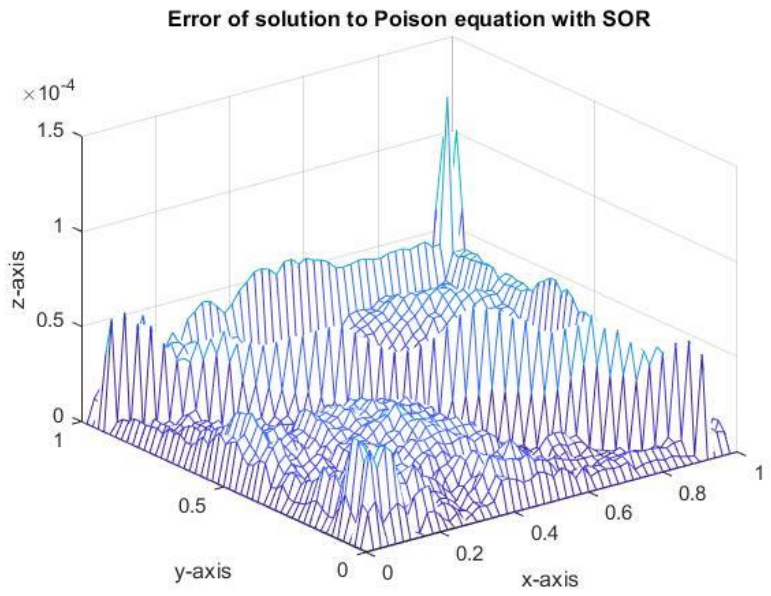

Figure 90

As we can see higher values of ω (up to a point) not only help find the solution faster but also make the solution more accurate given a constant $residue$.

References

1. Burden, Richard L., and J. Douglas. Faires. Numerical Analysis. Brooks/Cole, Cengage Learning, 2011.