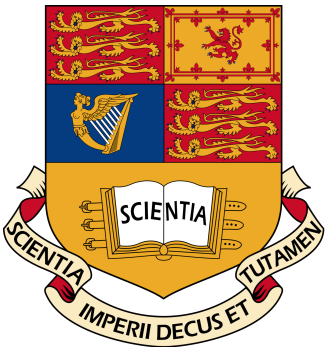
Mathematics EIE 2

Numerical Analysis of ODEs using Matlab

Numerical Analysis of ODEs using Matlab

Submitted By Group 13:
Maëlle Guerre CID 01496040
Adrian Koch CID 01517013
Hussain Kurabadwala CID 01535364
Joanna Merrick CID 01504728
Marcus Neo CID 01541100
Omar Sharif CID 01349634



Contents

1 Part 1 - RC Circuit

1.1 Background

Exercise 1. *Model the behaviour of the RC-circuit using three second-order Runge-Kutta methods: Heun's method, the midpoint method and a third method of your choice. The first two were covered in lectures, you will have to find the third one yourself.*

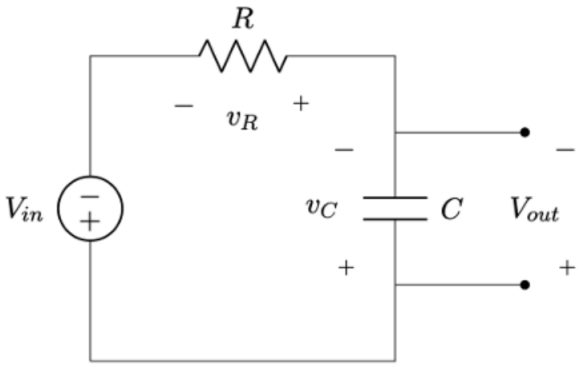


Figure 1: RC Circuit used for analysis

With initial condition: $qc(0) = 500nC$
And parameters: $R = 1000\Omega$, $C = 100nF$

To simplify our process of plotting the graphs in terms of V_{out} against V_{in}, we have decided to transform the equations to be in terms of V_{out} instead of q_c, using the following steps:

$$V_{in}(t) = q_c(t)\frac{1}{C} + Rq_c'(t) \tag{1}$$

$$V_{out}(t) = \frac{q_c(t)}{C} \implies V_{out}'(t) = \frac{q_c'(t)}{C} \tag{2}$$

$$Rq_c'(t) = RCV_{out}'(t) \tag{3}$$

Substituting Equation 3 and Equation 2 into Equation 1

$$V_{\text{in}}(t) = RCV_{out}'(t) + V_{out}(t) \tag{4}$$

$$V'_{\text{out}}(t) = \frac{V_{in}(t) - V_{out}(t)}{RC} \tag{5}$$

It is of greater interest to view the graph with the time axis in terms of milliseconds rather than seconds. As such, a simple division by a factor of 1000 would suffice. This implies: $V_{out}'(t) = \frac{V_{out}(t) - V_{in}(t)}{1000RC}$ where $V_{out}(0) = 5V, R = 1000\Omega, C = 100nF$

1.2 Understanding the Second Order Runge-Kutta Method

Predictors and Correctors

The main idea behind the Runge-Kutta methods is the concept of predictor and corrector.

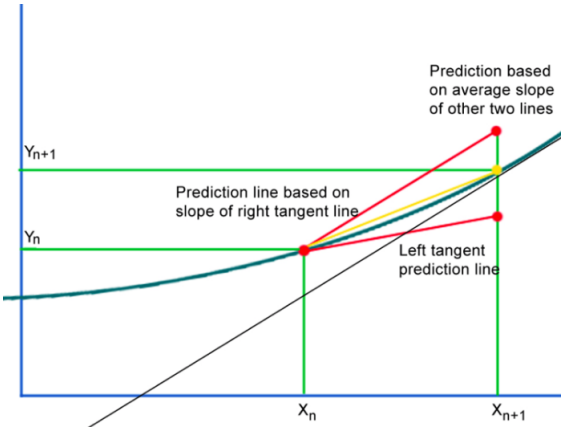


Figure 2: Visualising Heun's Method [2]

Using Heun's method as an example (Figure 2), the notion of predictors and correctors can be clearly portrayed, with the left and right tangent lines being the predictor and corrector respectively. Giving an increasing function, these two red lines are the overestimated and underestimated predictions, whilst the yellow line is the much more precise average of the two. By successive approximations above and below the ideal tangent Heun's method tends to the exact value of the function. As described in more detail later on, Heun's method improves its accuracy quadratically[1] with step size.

[1]Quadratically: The function increases based on a second-degree function (rather than linearly).

The Second Order Runge-Kutta Methods

Both the midpoint method and Heun's method are efficient in providing numerical solutions for first order Ordinary Differential Equations. These methods, together with infinitely many other variations, are classified as second order Runge-Kutta methods.

Although there are infinitely many variations of the 2nd order Runge-Kutta methods, the a, b, p and q parameters (Equation 7) must satisfy the following set of equations:

$$a \in [0, 1], \qquad b = 1 - a, \qquad p = q = \frac{1}{2b} \tag{6}$$

By changing the values of a, b, p and q we may switch between different methods. Three of such methods will be the focus of this exercise:

Variable	Heun's Method	Midpoint Method	Custom Method
a	1/2	0	1/4
b	1/2	1	3/4
p	1	1/2	2/3
q	1	1/2	2/3

Interpretation of the Runge-Kutta parameters

Consider the values of a, b, p and q in the context of each method.

1. Heun's method takes the predicted gradient at the current position, uses this predictor to find the point at which the second gradient should be taken and then averages the two gradients. This corresponds to $a = b = 0.5$ and $p = q = 1$.
2. The midpoint method takes the same initial predictor as Heun's and then takes the new gradient at a point half-way along the predicted gradient. This corresponds to $a = 0$, $b = 1$ and $p = q = 0.5$.
3. Our custom method uses the same initial predictor as the midpoint and Heun methods. It then takes the new gradient $\frac{2}{3}$ along the predicted one, weighting the predictor by $\frac{1}{4}$ and the new gradient by $\frac{3}{4}$.

The values of a and b are the weightings of the predictor and second gradient respectively. The values of p and q represent the point along the predicted gradient where the second gradient is taken - 0 being the same as the predictor and 1 being at the next point predicted by the predictor.

$$\begin{aligned} y_{(i+1)} &= y_i + y_i(ak_1 + bk_2)h, \\ predictor: k_1 &= f(x_i, y_i), \\ corrector: k_2 &= f(x_i + ph, y_i + qk_1h) \end{aligned} \tag{7}$$

1.3 Circuit Analysis

The circuit is a low pass filter with a transfer function of $\frac{-1}{1+j\omega RC}$ and a cut-off frequency of 1592 Hz.

Transfer Function Derivation Nodal analysis at node X, gives:

$$\frac{Y+X}{R} + \frac{Y-0}{\frac{1}{j\omega C}} = 0 \implies Y(1+j\omega RC) = -X \tag{8}$$

$$\frac{Y}{X} = -\frac{1}{(1+j\omega RC)} \tag{9}$$

Finding the impulse response of the circuit

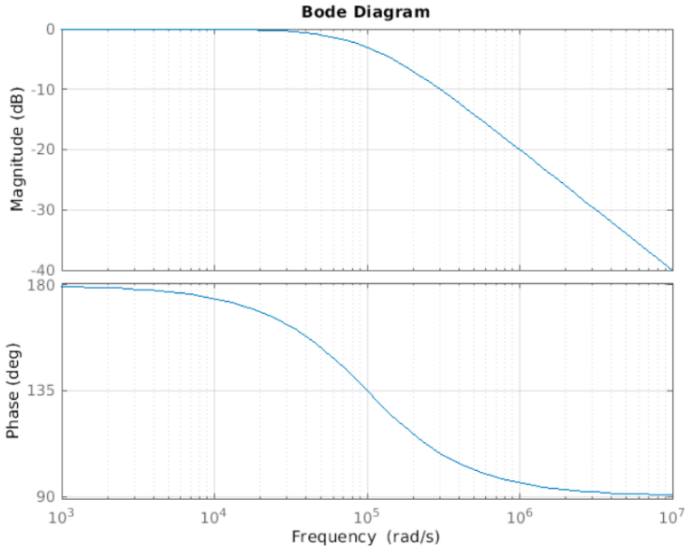


Figure 3: Bode plot for RC circuit

The step response of the system in terms of the Laplace variable s is $Y = \frac{1}{1+RCs}(\frac{1}{s})$. Taking the inverse Laplace transform yields the system response in the time domain.

Expressing the term above using partial fractions:

$$\frac{1}{s(1+RCs)} = \frac{1}{s} - \frac{RC}{1+RCs} = \frac{1}{s} - \frac{1}{\frac{1}{RC}+s} \tag{10}$$

Taking the inverse Laplace Transform:

$$\mathscr{L}^{-1}\left[\frac{1}{s} - \frac{1}{\frac{1}{RC}+s}\right] = 1 - e^{\frac{t}{RC}} \tag{11}$$

Since the transform is a linear operation, a step input a will lead to a response scaled by a factor of a (11). Accounting for the initial condition i we get $i - a(1 - e^{\frac{-t}{RC}})$. In the case of this RC circuit with a step input of 2.5, this becomes $5 - 2.5(1 - e^{-\frac{t}{0.0001}})$.

1.4 Matlab Code for Runge-Kutta Method (RK2.m)

```
function [x_values, y_values] = RK2(ODE, step_size, final_val,
    xi, yi, RKMETHOD) % ODE solver
    % @param ODE The ODE to be solved in the form y' = f(x, y)
    % @param step_size aka h The distance on the x-axis between
        two consecutive steps
    % @param final_val The x-value up to which the ODE is
        evaluated
    % @param xi The initial x-value, @param yi The initial y-
        value

    % Calculate the number of steps
    N = round((final_val - xi) / step_size);
    % Initialise output arrays
    x_values = zeros(1, N);
    y_values = zeros(1, N);
    x_values(1) = xi;
    y_values(1) = yi;

    %%%%%%% EDIT VALUE OF A FOR DIFFERENT METHODS %%%%%%
    % For Dynamic user input:
    a = RKMETHOD;
    % a = 0.5;  % For heun
    % a = 0;  % For Midpoint
    %a = 1/3;  % For Ralston
    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
    b = 1-a;
    p = 1/(2*b);
    q = p;
    % Using the formula y_(i+1) = y(i) + h*g(x_i, y_i, h)
    %  g = a*k_1 + b*k_2;  k_1 = f(x_i, y_i);  k_2 = f(x_i + p*h
        , y_i + q*k_1*h)
    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
    for i = 1 : N - 1 % Run for N - 1 iterations
        K_one = ODE(x_values(i), y_values(i));
        x_values(i+1) = x_values(i) + step_size;
        K_two = ODE(x_values(i)+(p*step_size), y_values(i)+q*
            K_one*step_size);
        y_values(i+1) = y_values(i)+step_size*(a*K_one+b*K_two);
    end  % Return the output arrays
end
```

1.5 Matlab code to test the function (RK2-script.m)

```
% The code here features only two example tests.
% The entire code can be found in the appendix.
M1 = "Midpoint Method";
M2 = "Heun's Method";
M3 = "Our Custom Quarter Method";
M4 = "Input V_in";
% RK2 function has the following inputs:
% RK2(ODE Equation, StepSize, Xfinal, X(0), Y(0), a)
% "a" refers to the variable "a" that determines the RK Method
% a=0 is midpoint, a=0.5 is heun. a has range [0,1]
%%%%%%%%%% TEST 1: Y = 2.5 %%%%%%%%%%%%%%%%%%
ODE = @(x, y) (2.5-y)*10;
figure; hold on;
[out_x1, out_y1] = RK2(ODE, 0.01, 1.5, 0, 5, 0);
a1 = plot(out_x1, out_y1);

[out_x2, out_y2] = RK2(ODE, 0.01, 1.5, 0, 5, 0.5);
a2 = plot(out_x2, out_y2);

[out_x3, out_y3] = RK2(ODE, 0.01, 1.5, 0, 5, 0.25);
a3 = plot(out_x3, out_y3);
input_x = zeros(1, 2001);
input_y = zeros(1,2001);
for i=1:2001
    input_x(i) = 0.00075*(i-1);
    input_y(i) = 2.5;
end
a4 = plot(input_x, input_y);
ylim([2, 5]);

legend([a1; a2; a3; a4], [M1; M2; M3; M4]);
hold off;

title('Test 1: Step signal');
xlabel('Time/ms');
ylabel('Voltage/V');

%% .. other tests not included here
%%%%%%%%%% TEST 5: Y = Sine, period 10us %%%%%%%%%%%%%%%%%%
ODE = @(x,y) ((5*sin(2*pi*(10^2)*x)-y)*10);
figure; hold on;
```

```
[out_x4, out_y4] = RK2(ODE, 0.0002, 0.2, 0, 5, 0);
a4 = plot(out_x4, out_y4);
[out_x5, out_y5] = RK2(ODE, 0.0002, 0.2, 0, 5, 0.5);
a5 = plot(out_x5, out_y5);
[out_x6, out_y6] = RK2(ODE, 0.0002, 0.2, 0, 5, 0.25);
a6 = plot(out_x6, out_y6);
for i=1:2001
    input_x(i) = 0.0001*(i-1);
    input_y(i) = 5*sin(2*pi*(10^2)*input_x(i));
end
a7 = plot(input_x, input_y);
legend([a4; a5; a6; a7], [M1; M2; M3; M4]);
hold off;
title('Test 5: Sine, period 10us');
xlabel('Time/ms');
ylabel('Voltage/V');
```

NOTE: To avoid having to manually change the method in the RK2 function, it is decided that the value of a will be passed as an argument to the function. This allows different methods to be plot on the same graph.

1.6 Output graphs

Step Signal Input

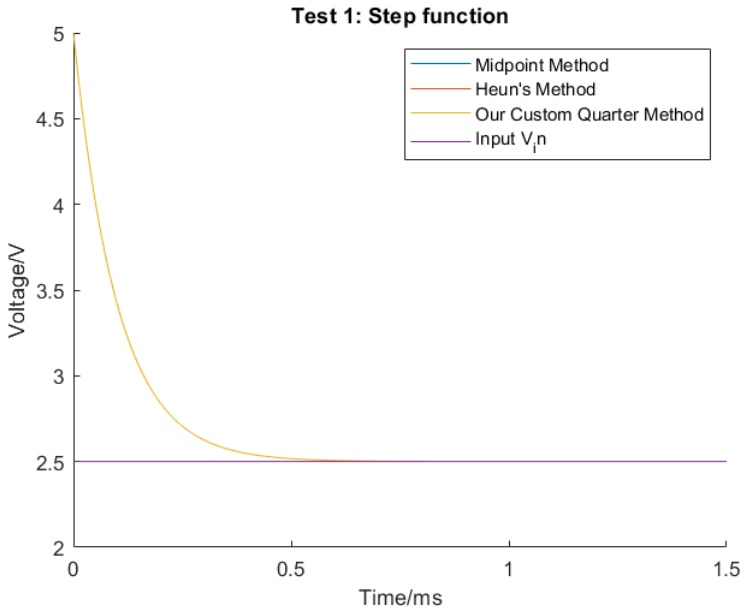


Figure 4: RC Circuit used for analysis

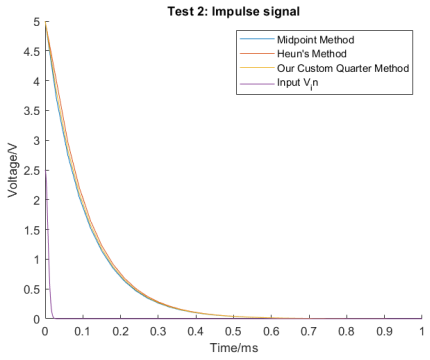
For $V_{in} = 2.5V$, the differential equation of the RC circuit (in milliseconds) is

$$y' + 10y = 25; y(0) = 5$$

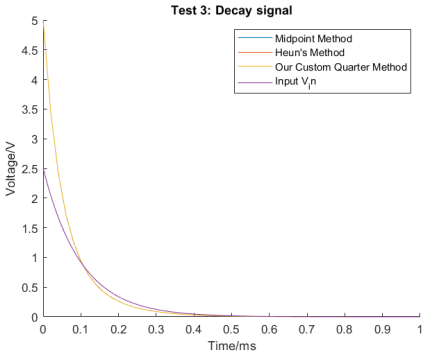
We input a step signal of amplitude 2.5V. The input is smaller than the initial voltage across the capacitor (5V), which causes the capacitor to discharge. Therefore, the output decays to a constant of 2.5V. All three methods give very similar outputs because the output voltage tends towards a constant, making the approximation easier as the numerical methods use the gradient to calculate the next value.

When we increase the time constant of the circuit (RC), the exponent term in the solution of the ODE will decrease slower and, hence, the output will reach steady state later. Since the steady state is constant at 2.5V we can use the step response to isolate the transient to be $V_{out} - 2.5$.

Impulse and Decay Signal Inputs



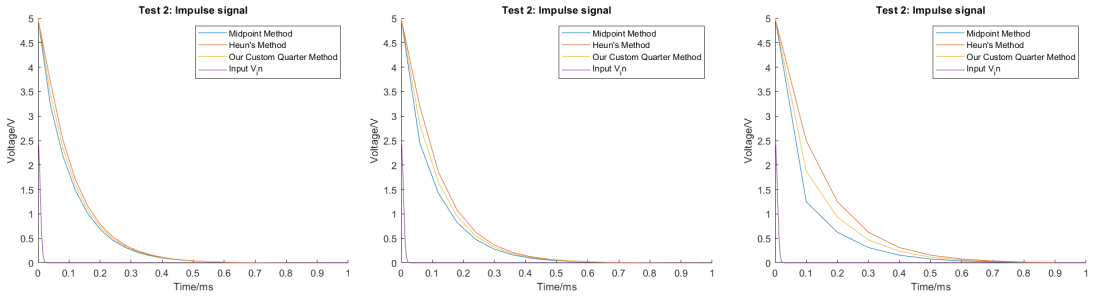
(a) Impulse Signal $V_{in}(x) = 2.5e^{\frac{-x^2}{0.1}}$



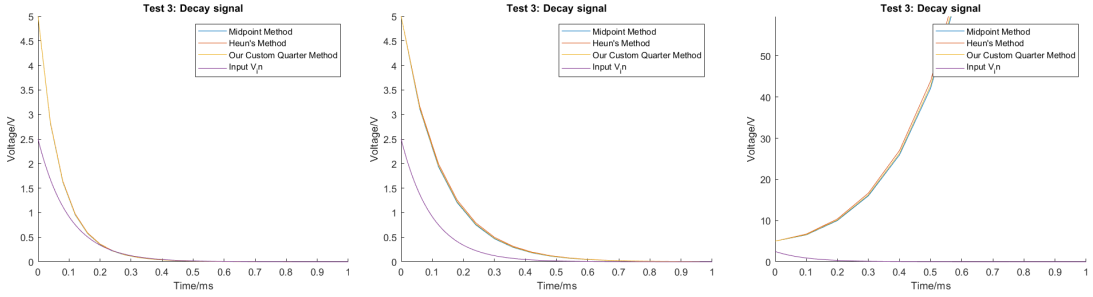
(b) Decay Signal $V_{in}(x) = 2.5e^{\frac{-x}{0.0001}}$

Figure 5: Impulse and Decay, RC Solver Output

All three methods give similar outputs. For the signals above, the gradient tends to a constant, which means that the plots are more precise. However, as the step size increases, the difference between the methods becomes more noticeable. The Midpoint method clearly underestimates the exact solution, whereas both Heun's method and our custom method, despite sticking closer to the exact solution, give an overestimation of the exact solution.



(a) Impulse, Stepsize=60μs (b) Impulse, Stepsize=90μs (c) Impulse, Stepsize=150μs



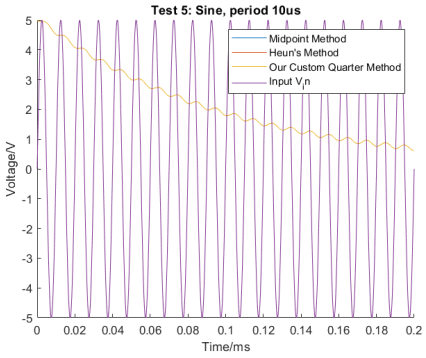
(d) Decay, Stepsize=60μs (e) Decay, Stepsize=90μs (f) Decay, Stepsize=150μs

Figure 6: Impulse and Decay Signals with Varying Stepsizes

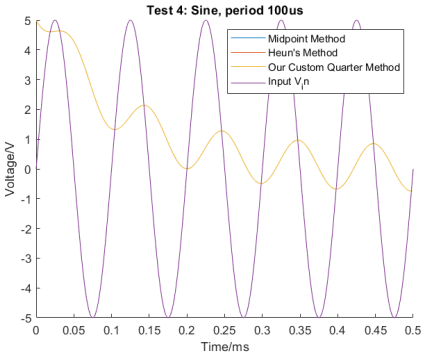
Another important observation we made is that the greater the step size the more inaccurate each method became. Upon reaching a threshold step size, the output values estimated by all of the Runge-Kutta methods will be totally different compared to the exact solution. This is most evident for the decay function with a large step size (Figure 6f), where a large step size causes the estimated solutions to yield a wrong increasing function of range $[5, \infty)$.

Sine Waveform Inputs

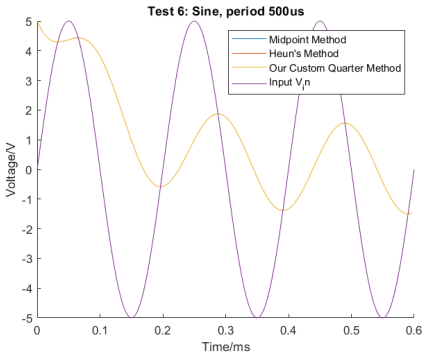
Sinusoidal input to the RC circuit (in milliseconds): $V_{in}(x) = 5\sin\left(\frac{2\pi x}{T}\right)$



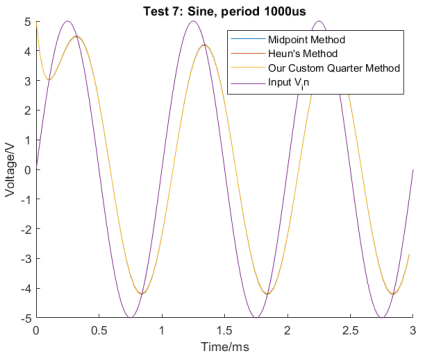
(a) Sine wave, T = 10μs



(b) Sine wave, T = 100μs



(a) Sine wave, T = 0.5ms



(b) Sine wave, T = 1ms

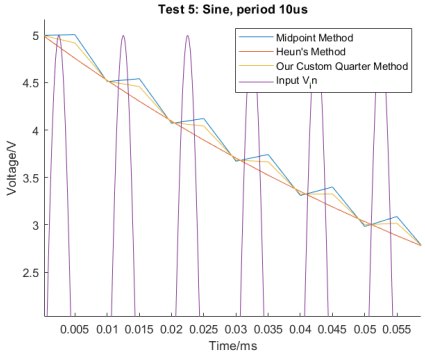
The capacitor charges (the output increases) when the voltage across the capacitor is less than the input. It discharges (the output decreases) when the voltage across the capacitor is greater than the input.
Since the circuit is a low pass filter, the amplitude of the steady state output decreases with higher frequencies. On the tests above this trend is noticeable - for example for T = 1ms, the amplitude is about 4V, whereas for T = 0.5ms, the amplitude is 1V. Whilst t_f varies between graphs, the transient envelope remains the same. As the step size gets comparatively smaller (i.e. the step size remains constant, but the time period increases, making the step size smaller in comparison), the approximation tends towards a linear coefficient. For example, for a period of 1000μs the gradient between 0.7 and 1ms is almost constant. Therefore, we can deduce (and as seen from the graphs) that the methods are more accurate when approximating the input sine wave of high time periods (low frequencies).

For high frequencies the length of time where $V_{in} > V_C = V_{out}$ is not long enough for the capacitor to fully charge. The difference between the capacitor voltage and -5V is greater

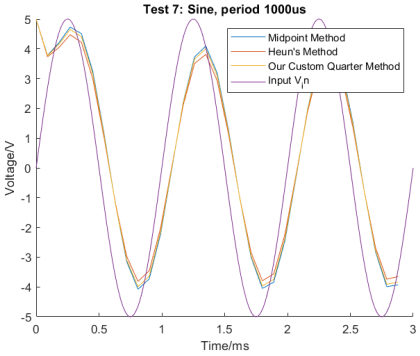
than the difference between the capacitor voltage and 5V; therefore, the output decreases and we have an exponentially decaying envelope, giving a physical explanation for the transient. For higher frequencies, the phase lag is more prominent. This is because, as the frequency starts increasing the $1/s$ factor in the transfer function starts increasing. This causes the circuit to start acting as an integrator since $s \gg \frac{1}{RC}$ and the transfer function can be approximated as $\frac{1}{s}$. Therefore, the sine input gets integrated to a cosine which shows as phase lag between input and output.

For all graphs, the output curve starts at 5V because the capacitor is initially charged. The capacitor discharges until the output becomes purely periodic when it reaches the steady state. For all methods and all frequencies, the steady state is reached at around 0.2 milliseconds.

To see the differences between methods we increased the step size by a suitable factor.



(a) T = 10μs, step size = 1μs



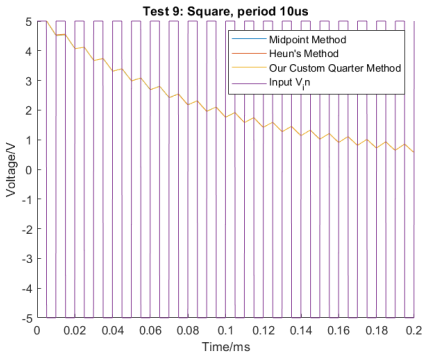
(b) T = 1ms, 90μs

Figure 9: Differences in Methods (Sine Waves)

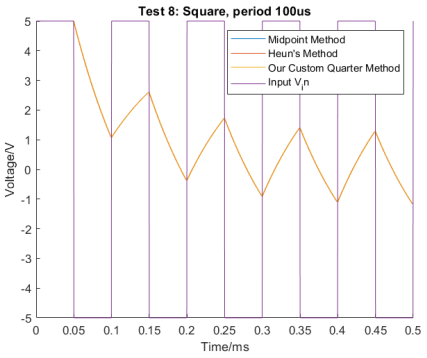
The custom method has $a = 0.25$, which is in between that of the Heun's method ($a = 0.5$) and the midpoint method ($a = 0$). As such, from this it is apparent that the custom method has estimations that lie in between the estimated curves of the Heun's and the midpoint method.

Square Waveform Inputs

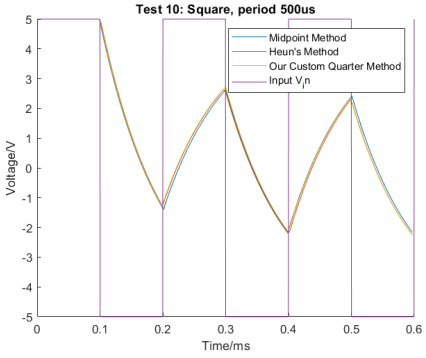
Square Wave input to the RC circuit (in milliseconds): $5square(\frac{2\pi x}{T})$



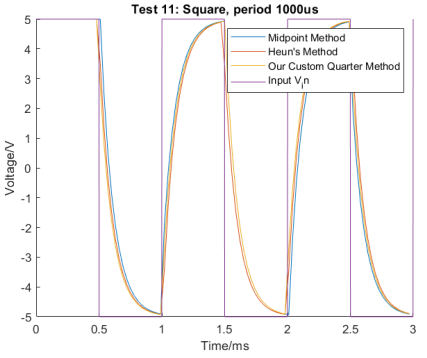
(a) T = 10μs



(b) T = 100μs



(c) T = 0.5ms



(d) T = 1ms

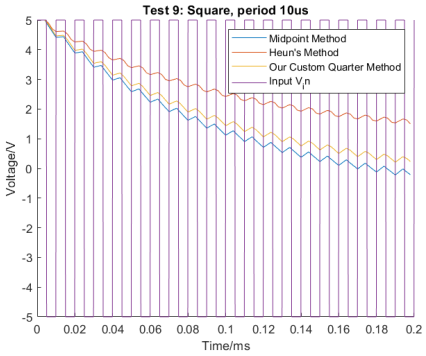
Figure 10: Square Wave Inputs

The capacitor charges on the rising edge of the square wave (we can see the output increasing) and discharges on the falling edge of the square wave (we can see the output decreasing). The rate of charging/discharging depends on the difference between the input voltage and the voltage across the capacitor. V_{out} starts at $5V$ as the capacitor is initially charged. At the falling edge the input falls directly to -5V and the capacitor begins discharging, so the output decreases exponentially. At the next rising edge, the input goes to +5V and so the capacitor starts charging, but charges at a slower rate than discharging because the difference between the capacitor voltage and +5 is smaller than the capacitor voltage and-5. This continues and the discharging rate is always higher, and hence the first 3 graphs show the decaying exponential envelope.

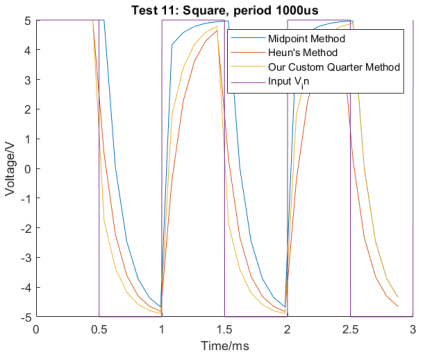
At lower frequencies, the time period is large enough to allow complete discharging and charging. For example, when the period is 1000μs, the capacitor is constantly discharging and charging fully; for capacitors the time to charge is $5RC = 500\mu s$, which is exactly

the amount of time given for each section of the square wave. Additionally, at this time period the transient has almost completely died, so it is only steady state.

To observe the differences in the methods we increased the step size by a suitable factor (Figure 11).



(a) T = 10μs, step size = 2μs



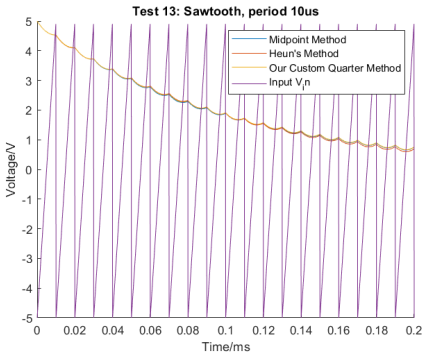
(b) T = 1ms, step size = 90μs

Figure 11: Differences in Methods (Square Wave)

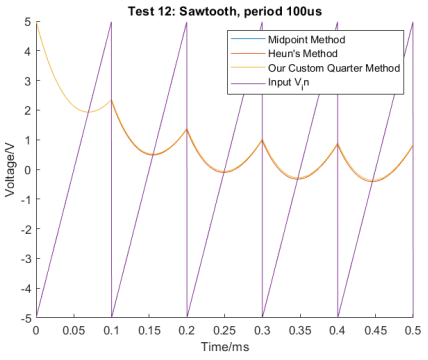
For small periods we observe that, whilst all 3 methods decay in a similar shape, our custom method has a more similar decay envelope to the midpoint method (Figure 11a).

Sawtooth Waveform Inputs

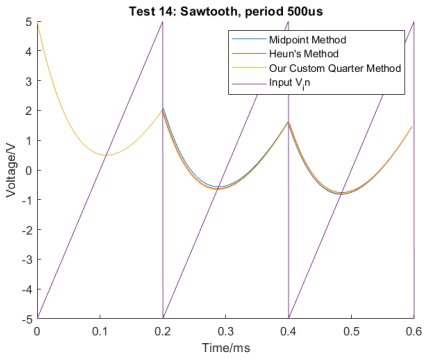
Sawtooth Wave input to the RC circuit (in ms): $5sawtooth(\frac{2\pi x}{T})$



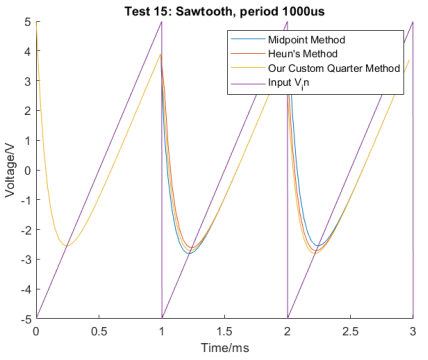
(a) T = 0.5ms



(b) T = 1ms



(c) T = 0.5ms

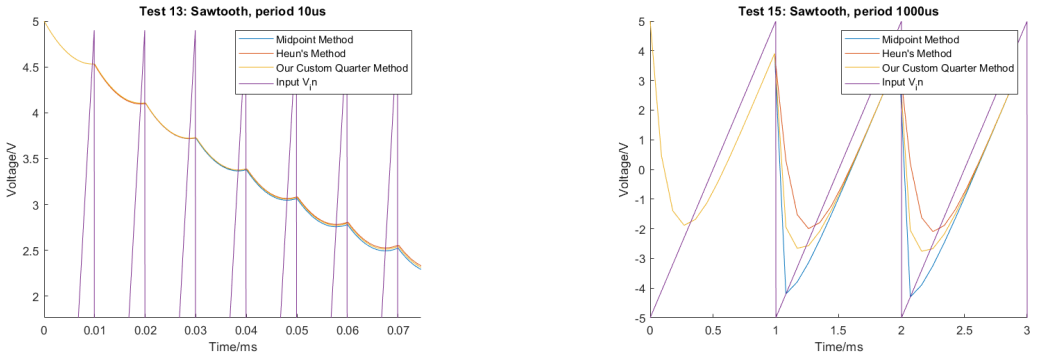


(d) T = 1ms

Figure 12: Sawtooth Wave Inputs

These graphs confirm the analysis made above for square and sinewaves. For the sawtooth wave, the capacitor discharges until the sawtooth voltage is greater than the capacitor voltage, at which point it begins to charge until the sawtooth voltage is less than the capacitor voltage, where it begins to discharge and the process repeats.

To observe the differences in the methods we increased the step size by a suitable factor again.



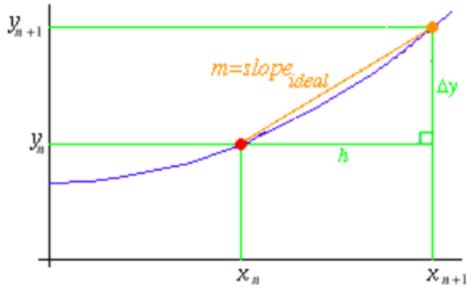
(a) Sawtooth wave, T = 10μs, step size = 2μs (b) Sawtooth wave, T = 1ms, step size = 90μs

Figure 13: Differences in Methods (Sawtooth)

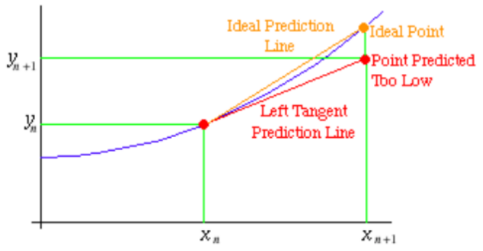
From these graphs we can see that, for the first period of the sawtooth wave input, all the RK2 methods have the same values, after this they deviate.

1.7 Analysis

From the data collected, we find that the Runge-Kutta methods are accurate with sufficiently small step sizes. However, as the step size increases, inputs that tend towards a constant will have better output accuracy compared to other inputs e.g. sinusoids.



(a) Accurate Slope Prediction



(b) Inaccurate Slope Prediction

Figure 14: Causes for the Changes in Accuracy with the Heun Method

The picture above gives another insight in how Heun's method works. The "ideal gradient" (orange line) is the gradient that is successively estimated. For low frequencies, the blue curve (representing the real function), changes its slope slowly enough so that the method is able to compute a good approximation. For example, if it takes 3 iterations for the method to find the exact slope, the numerical method will be accurate if those three iterations take less time than half of the function period for a square wave. The time ∂x depends on h. For high frequencies, the period T is sufficiently greater than h, the real slope of the function between these two points $y'(\partial x)$ is equal to the computer slope m.

$$T \gg h \rightarrow y'(\delta x) = \frac{\Delta y}{h} = m \tag{12}$$

For low frequencies, the slope is not a constant because the function changes too rapidly

$$T < h \rightarrow y'(\delta x) = f(x) \neq m \tag{13}$$

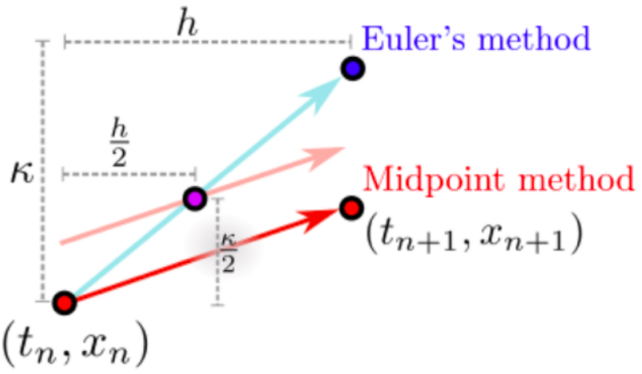


Figure 15: Comparing Midpoint and Euler method

One way to visualise the midpoint method is to compare it to Euler's method. Euler has an error of $O(h^2)$ per step, meaning that the error at each step is proportional to the step size squared, and $O(h)$ overall, whereas the midpoint method (including any other second order Runge-Kutta methods) has a global truncation error of $O(h^2)$. However, both methods will suffer degradation in precision as the step size approaches the function's rate of change. Intuitively, the midpoint method checks the point at the middle of the curve rather than at ends which gives greater precision because it reduces the chance of missing a slope.

1.8 Error Analysis

Background

Exercise 2. *Write a script called error script.m in which you carry out error analysis for the basic case with R, C, $q_C(0)$ as above, and as input a cosine wave of period $T = 100s$ and amplitude $V_{in} = 5V$. Compare the numerical solution with the exact solution, obtaining the error as a function of t. Plot the error function. Vary the time step h for a suitable number and range of values, and obtain a log-log plot to show the order of the error. Be sure to explain and justify your empirical observation.*

The goal here is to carry out error analysis for the section above with the same values of $R, C, q_C(0)$. The input to the system is a cosine wave of period $T = 100\mu s$, amplitude $V_{in} = 5V$. The error analysis computes the difference between the exact solution of the ODE and the solution approximated using numerical methods.

Exact Solution

For the given input:

$$5cos(\frac{2\pi x}{T}), T = 100\mu s \tag{14}$$

The exact solution of the ODE is given by:

$$5 \cdot \frac{2\pi(2\pi e^{-10000x} + \sin 20000\pi x) + \cos 20000\pi x}{1 + 4\pi^2} \tag{15}$$

1.9 Derivation

From Equation (5), we know that $y'(x) = \frac{z(x)-y(x)}{RC}$, where $y(x) = V_{out}(x)$ and $z(x) = V_{in}(x)$. Since $R = 10^3$ and $C = 10^{-7}$, $\frac{1}{RC} = 10^4$. Hence the ODE can be represented by:

$$\begin{aligned} y'(x) &= 10^4(z(x) - y(x)) \\ y'(x) + 10^4 y(x) &= 10^4 z(x) \\ P(x) = 10^4, Q(x) &= 10^4 z(x) \end{aligned} \tag{16}$$

Since $z(x) = 5cos(\frac{2\pi x}{T})$, $T = 100\mu s = 10^{-4}ms$, $z(x) = 5cos(2\pi 10^4 x)$

By observation, Equation (16) is in the form of an exact first order differential equation. Therefore, we can solve for the function solely in terms of $y(x)$ using the following steps:

1. Obtain the integrating factor($\mu(x)$)

$$\begin{aligned} \mu(x) &= e^{\int P(x)dx} \\ &= e^{\int 10^4 dx} \\ &= e^{10^4 x} \end{aligned} \tag{17}$$

2. Rewrite the ODE in terms of $\mu(x)y(x)$

$$\begin{aligned} \mu(x)y(x) &= \int \mu(x)Q(x)dx + C \\ e^{10^4x}y(x) &= \int 50000cos(2\pi 10^4 x)e^{10^4x} \\ &= 50000 \int cos(2\pi 10^4 x)e^{10^4x} \end{aligned} \tag{18}$$

3. Solve the integral using the integration by parts method $\int uv' = uv - \int u'v$

   Let $v' = e^{10^4x} \implies v = 10^{-4}e^{10^4x}, \quad u = cos(2\pi 10^4 x) \implies u' = -2\pi 10^4 sin(2\pi 10^4 x)$

$$\begin{aligned} \int cos(2\pi 10^4 x)e^{10^4x} &= 10^{-4}cos(2\pi 10^4 x)e^{10^4x} + \int \frac{2\pi 10^4}{10^4} sin(2\pi 10^4 x)e^{10^4x}dx \\ &= 10^{-4}cos(2\pi 10^4 x)e^{10^4x} + 2\pi \int sin(2\pi 10^4 x)e^{10^4x}dx \end{aligned} \tag{19}$$

4. Solve the inner integral using the integration by parts method again.

   Let $v' = e^{10^4x} \implies v = 10^{-4}e^{10^4x}, \quad u = sin(2\pi 10^4 x) \implies u' = 2\pi 10^4 cos(2\pi 10^4 x)$

$$\begin{aligned} \int sin(2\pi 10^4 x)e^{10^4x}dx &= 10^{-4}sin(2\pi 10^4 x)e^{10^4x} - \int \frac{2\pi 10^4}{10^4} cos(2\pi 10^4 x)e^{10^4x}dx \\ &= 10^{-4}sin(2\pi 10^4 x)e^{10^4x} - 2\pi \int cos(2\pi 10^4 x)e^{10^4x}dx \end{aligned} \tag{20}$$

5. Substitute Equation(20) into Equation(19)

$$\int cos(2\pi 10^4 x)e^{10^4x} = 10^{-4}cos(2\pi 10^4 x)e^{10^4x} + 2\pi \int sin(2\pi 10^4 x)e^{10^4x}dx$$

$$= 10^{-4}cos(2\pi 10^4 x)e^{10^4x} + 2\pi\left(10^{-4}sin(2\pi 10^4 x)e^{10^4x} - 2\pi \int cos(2\pi 10^4 x)e^{10^4x}dx\right)$$

$$= 10^{-4}cos(2\pi 10^4 x)e^{10^4x} + 2\pi 10^{-4}sin(2\pi 10^4 x)e^{10^4x} - 4\pi^2 \int cos(2\pi 10^4 x)e^{10^4x}dx \tag{21}$$

6. There is now the common term $\int cos(2\pi 10^4 x)e^{10^4x}dx$ on both the left and the right hand side of Equation(21). Some manipulation is then done to achieve:

$$(1 + 4\pi^2) \int cos(2\pi 10^4 x)e^{10^4x} = 10^{-4}cos(2\pi 10^4 x)e^{10^4x} + 2\pi 10^{-4}sin(2\pi 10^4 x)e^{10^4x}$$

$$\int cos(2\pi 10^4 x)e^{10^4x} = \frac{cos(2\pi 10^4 x)e^{10^4x} + 2\pi sin(2\pi 10^4 x)e^{10^4x}}{10^4(1 + 4\pi^2)} + C \tag{22}$$

7. Substitute Equation(22) into Equation(18) to obtain:

$$\begin{aligned} e^{10^4x}y(x) &= 50000\int cos(2\pi 10^4x)e^{10^4x} \\ &= 50000\left(\frac{cos(2\pi 10^4x)e^{10^4x} + 2\pi sin(2\pi 10^4x)e^{10^4x}}{10^4(1+4\pi^2)}\right) + C \\ y(x) &= 5\left(\frac{cos(2\pi 10^4x)e^{10^4x} + 2\pi sin(2\pi 10^4x)e^{10^4x}}{e^{10^4x}(1+4\pi^2)}\right) + \frac{C}{e^{10^4x}} \\ y(x) &= 5\left(\frac{cos(2\pi 10^4x) + 2\pi sin(2\pi 10^4x)}{1+4\pi^2}\right) + \frac{C}{e^{10^4x}} \end{aligned} \tag{23}$$

8. Solve for C when $y(0) = 5$

$$\begin{aligned} y(0) &= 5\left(\frac{cos(2\pi 10^4(0)) + 2\pi sin(2\pi 10^4(0))}{1+4\pi^2}\right) + \frac{C}{e^{10^4(0)}} \\ 5 &= \frac{5}{1+4\pi^2} + C \\ C &= \frac{5(4\pi^2)}{1+4\pi^2} \end{aligned} \tag{24}$$

9. Solve for $y(x)$:

$$\begin{aligned} y(x) &= 5\left(\frac{cos(2\pi 10^4x) + 2\pi sin(2\pi 10^4x)}{1+4\pi^2}\right) + \frac{5(4\pi^2)}{e^{10^4x}(1+4\pi^2)} \\ &= 5\left(\frac{4\pi^2e^{10^{-4}x} + 2\pi sin(2\pi 10^4x) + cos(2\pi 10^4x)}{1+4\pi^2}\right) \\ &= 5\left(\frac{2\pi(2\pi e^{-10000x} + sin(20000\pi x)) + cos(20000\pi x)}{1+4\pi^2}\right) \end{aligned} \tag{25}$$

MATLAB Error script

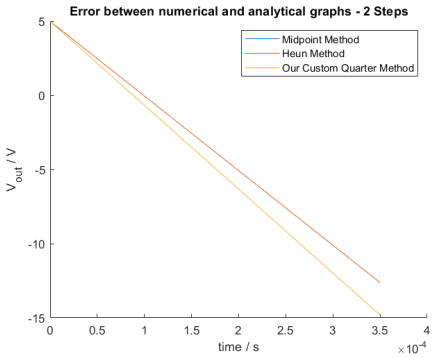
```
% Error analysis for the RC circuit with a cosine input
% the numerical solution is compared to the mathematically computed exact solution
% the error is then plotted for each method.

% THE EXACT SOLUTION OF THE ODE IS GIVEN BY:
% y = 5(2pi(2pi(e^(-10000x))+sin(20000pix))+cos(20000pix))/(1+4pi^2)

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% 1. CREATING THE EXACT EQUATION INTO THE MATRIX
% this exact value ranges between x = 0 and 7x10^-4 seconds
% and has 1000 steps. This mean that each step is 0.7us.
exact_x = zeros(1,1000);
exact_y = zeros(1,1000);
for i = 1:10000
    exact_x(i)= 0.00000007*i;
    exact_y(i) = (5*(2*pi*(2*pi*exp(-10000*exact_x(i))+sin(20000*pi*exact_x(i)))+cos(20000*pi*exact_x(i))))/(1+(4*pi*pi));
end
Solution = @(x) (5*(2*pi*(2*pi*exp(-10000*x)+sin(20000*pi*x))+cos(20000*pi*x)))/(1+(4*pi*pi));

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%2. Analytical Solving
ODE =@(x,y) ((5*cos(2*pi*(10^4)*x)-y)*10000);

%%%%%% SETTING UP THE LEGEND %%%%%%
Aa = "Midpoint Method";
Bb = "Heun Method";
Cc = "Our Custom Quarter Method";
Ee = "Exact Solution";

%%%%%% SETTING THE STEP SIZES %%%%%%
steps = [2, 4, 5, 8, 10, 16, 20, 25, 40, 50, 80, 100, 125, 200, 250, 400, 500, 625, 1000, 1250, 2000, 2500, 5000];
stepsize = 10000./steps.*0.00000007;

```

```
%%%%%% Initialising Array Spaces %%%%%%
A_x=zeros(1,23);
A_yRMS=zeros(1,23);
A_y=zeros(1,23);
B_x=zeros(1,23);
B_yRMS=zeros(1,23);
B_y=zeros(1,23);
C_x=zeros(1,23);
C_yRMS=zeros(1,23);
C_y=zeros(1,23);

%%%%%% DO THE LOOP THROUGH ALL FACTORS OF 10000 %%%%%%
for i = 1:23
    [out_x1, out_y1] = RK2(ODE, stepsize(i), 0.0007, 0, 5, 0);
    [out_x2, out_y2] = RK2(ODE, stepsize(i), 0.0007, 0, 5, 0.5);
    [out_x3, out_y3] = RK2(ODE, stepsize(i), 0.0007, 0, 5, 0.25)
        ;
    figure; hold on;
    title(strcat("Comparison of Graphs - ", num2str(steps(i))," 
        Steps"));
    A = plot(out_x1, out_y1);
    B = plot(out_x2, out_y2);
    C = plot(out_x3, out_y3);
    E = plot(exact_x, exact_y);
    legend([A; B; C; E], [Aa; Bb; Cc; Ee]);
    ylabel("V_{out} / V"); xlabel("time / s");
    hold off;
    A_error_y = zeros(1, steps(i));
    B_error_y = zeros(1, steps(i));
    C_error_y = zeros(1, steps(i));
    for j = 1:steps(i)
        A_error_y(j) = abs(exact_y((10000/steps(i))*j) - out_y1
            (j));
        B_error_y(j) = abs(exact_y((10000/steps(i))*j)- out_y2(
            j));
        C_error_y(j) = abs(exact_y((10000/steps(i))*j) - out_y3
            (j));
    end

    figure; hold on;

    title(strcat("Error between numerical and analytical graphs
```

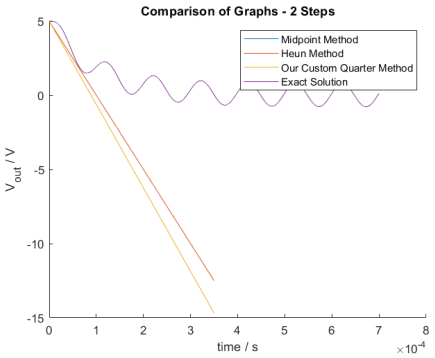
```
        - ", num2str(steps(i)), " Steps"));
73
74      ylabel("V_o_u_t / V");
75      xlabel("time / s");
76      A = plot(out_x1, A_error_y);
77      B = plot(out_x2, B_error_y);
78      C = plot(out_x3, C_error_y);
79      legend([A; B; C], [Aa; Bb; Cc]);
80      hold off;
81      A_x(i) = log10(stepsize(i));
82      A_yRMS(i) = log10(abs(rms(A_error_y)));
83      A_y(i) = log10(mean(A_error_y));
84      B_x(i) = log10(stepsize(i));
85      B_yRMS(i) = log10(abs(rms(B_error_y)));
86      B_y(i) = log10(mean(B_error_y));
87      C_x(i) = log10(stepsize(i));
88      C_yRMS(i) = log10(abs(rms(C_error_y)));
89      C_y(i) = log10(mean(C_error_y));
90      clear A_error_y;
91      clear B_error_y;
92      clear C_error_y;
93  end
94
95  figure; hold on;
96  title("log(average error) against log(step size)");
97  ylabel("log(average error)");
98  xlabel("log(step size)");
99  A = plot(A_x, A_y);
100 B = plot(B_x, B_y);
101 C = plot(C_x, C_y);
102 legend([A; B; C], [Aa; Bb; Cc]);
103 hold off;
104
105 figure; hold on;
106 title("log(rms(error)) against log(step size)");
107 ylabel("log(rms(error))");
108 xlabel("log(step size)");
109 A = plot(A_x, A_yRMS);
110 B = plot(B_x, B_yRMS);
111 C = plot(C_x, C_yRMS);
112 legend([A; B; C], [Aa; Bb; Cc]);
113 hold off;
```

Output Graphs

For a low step number, the methods are unable to provide a useful approximation. With only two steps, all methods tend to a straight-line approximation with gradient being approximately -5 (Figure 16).



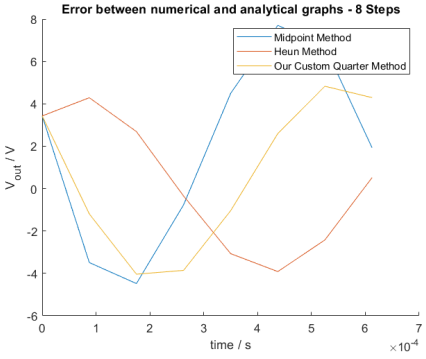
(a) Deviation from Exact Value



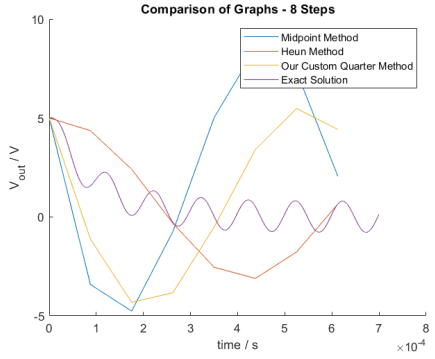
(b) Outputs of Each Method

Figure 16: 2 Total Steps

As the step number increases, discrepancies between methods start to appear (Figure 17). The midpoint method fluctuates with particularly high and low values, while our method and Heun's compute an approximation that tends to the exact solution average value.



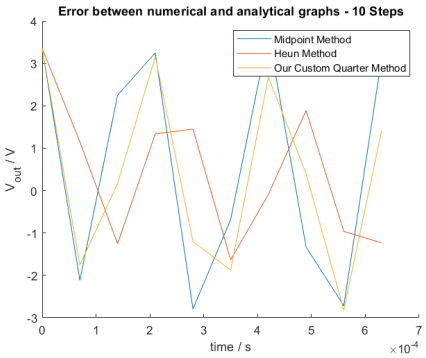
(a) Deviation from Exact Value



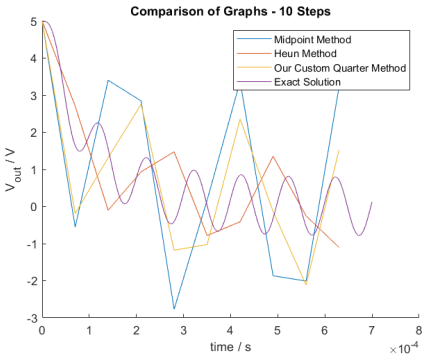
(b) Outputs of Each Method

Figure 17: 8 Total Steps

As the number of steps increases, the error between methods decreases. When the number of steps is 16, we get the first reasonable approximation (Figure 19). The overall trend of a periodic function can be identified, yet the exact values, transient and steady states are smothered by the approximations.

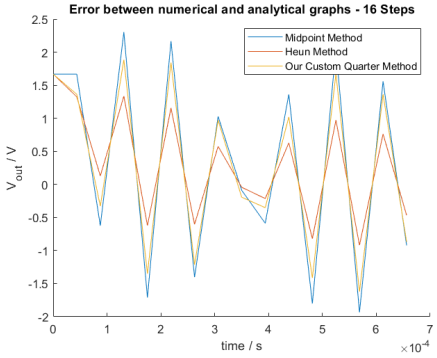


(a) Deviation from Exact Value

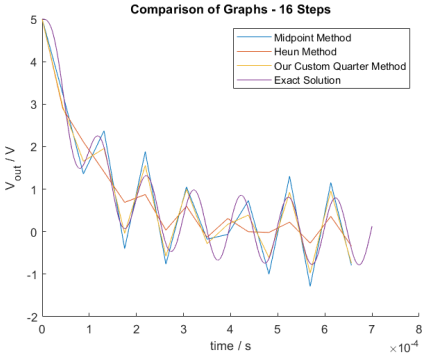


(b) Outputs of Each Method

Figure 18: 10 Total Steps

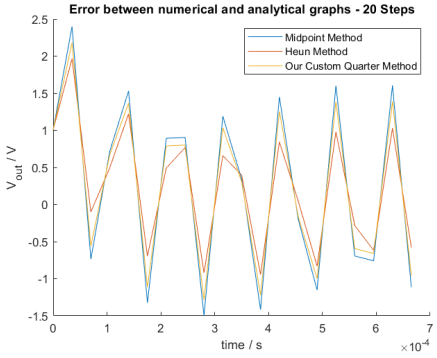


(a) Deviation from Exact Value

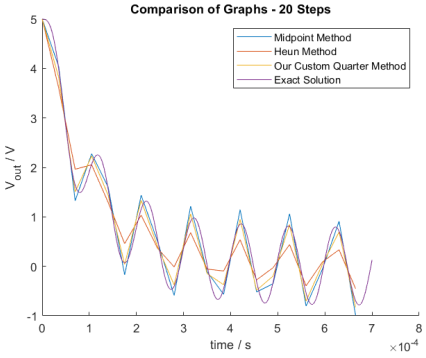


(b) Outputs of Each Method

Figure 19: 16 Total Steps



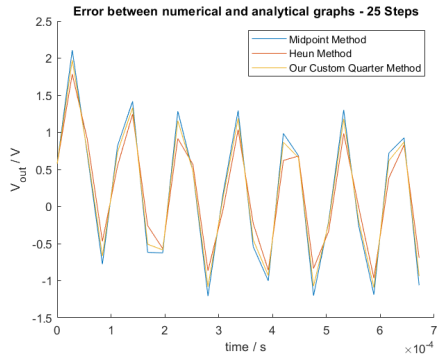
(a) Deviation from Exact Value



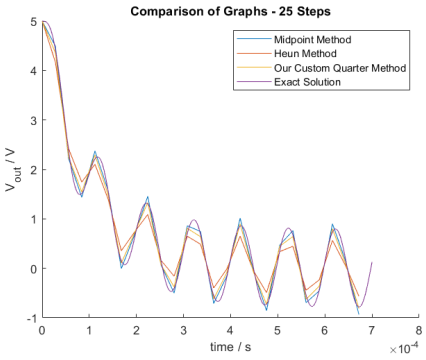
(b) Outputs of Each Method

Figure 20: 20 Total Steps

From number of steps = 25, the approximation becomes precise enough to identify steady state and transient state (Figure 21). The midpoint method has the most fluctuations in

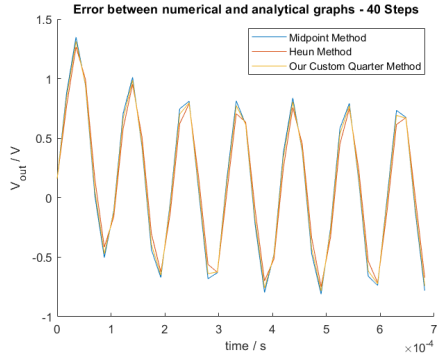


(a) Deviation from Exact Value

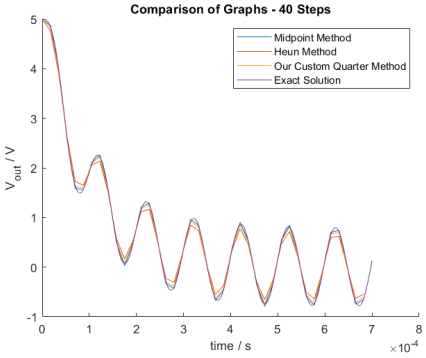


(b) Outputs of Each Method

Figure 21: 25 Total Steps



(a) Deviation from Exact Value



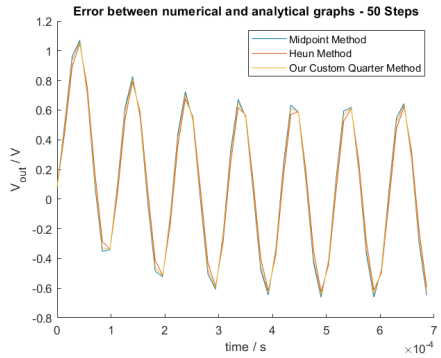
(b) Outputs of Each Method

Figure 22: 40 Total Steps

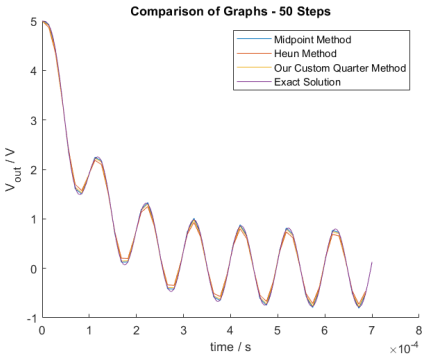
amplitude whilst our method follows the exact solution more efficiently (less amplitude fluctuations).

At number of steps greater or equal to 50, the error follows the exact solution more closely (Figure 21) – this can be seen in the log-log graph as the start of the straight-line section ($10^{1.7} = 50$). For step numbers larger than this it just gets closer and closer to the exact solution.

At number of steps = 200, the graphs are completely stacked on top of each other, and the error gets increasingly smaller after this point (Figure 24). For very large step numbers we see that the graphs are identical (at the current zooming).

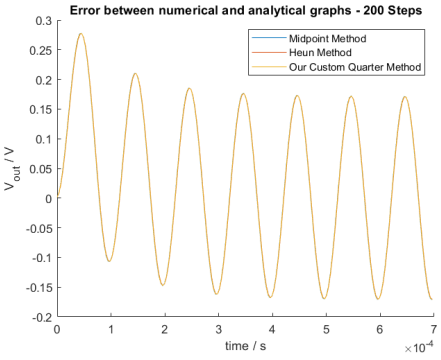


(a) Deviation from Exact Value

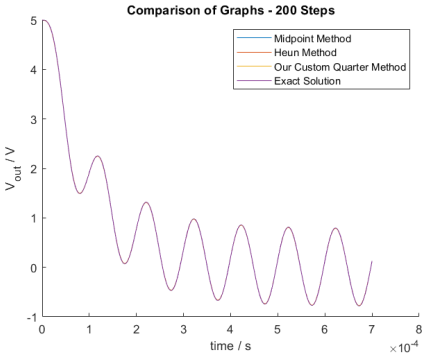


(b) Outputs of Each Method

Figure 23: 50 Total Steps

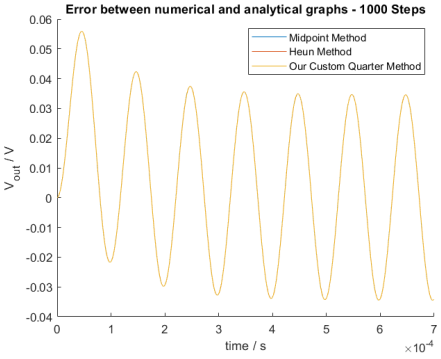


(a) Deviation from Exact Value

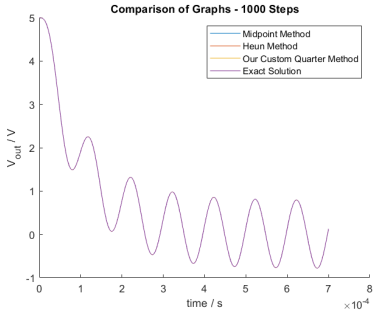


(b) Outputs of Each Method

Figure 24: 200 Total Steps



(a) Deviation from Exact Value



(b) Outputs of Each Method

Figure 25: 1000 Total Steps

By plotting the logarithm of the error against the the logarithm of the number of total steps, a relationship between error and total steps could be found, where the rate of change of error with respect to the size would be m, and the vertical axis intercept would be C.

$$\log(Error(t)) = m\log(steps\_size) + C, \tag{26}$$

where Error(t) is the deviation of the estimated output from the exact output at time t.

However, since the error is a function of time, it means that the error changes with each function for each value of time. Without plotting a three-dimensional graph, some changes have to be made to accurately represent the error with respect to step size.

One method that was discussed was to attribute the error as the average error with respect to time.

$$\log(average\_error) = m\log(step\_size) + C, \tag{27}$$

$$average\_error = \frac{1}{total\_time}\sum_{step=1}^{total\_steps} Error\Big[(step)(step\_size)\Big] \tag{28}$$

By plotting the logarithm of the average error with respect to the step size, a linear trend could be identified (Figure 26).

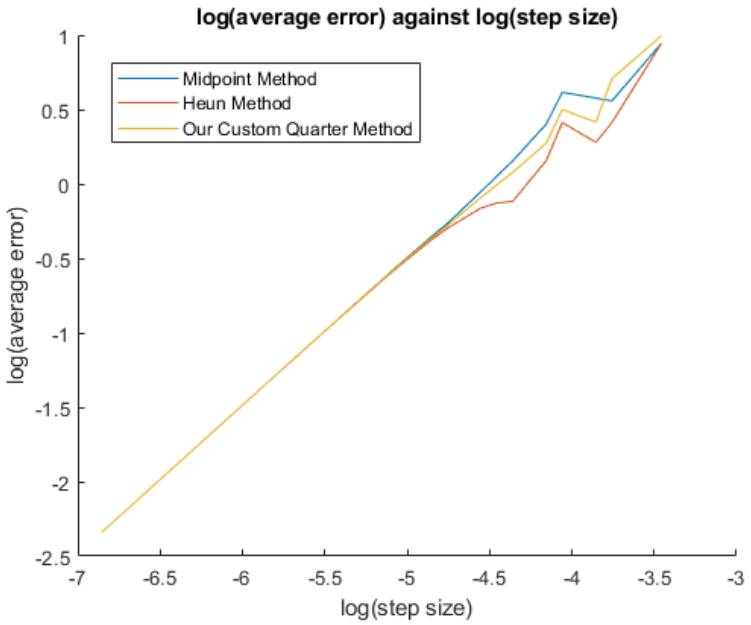


Figure 26: Graph Showing log(average_error) Against log(step_size)

However, as the input is a cosine function, the error is also sinusoidal. It is believed to be insufficient to get the average with respect to time as sinusoidal nature of the error function causes the positive and the negative errors to cancel out with one another.

As such, another method is necessary to more accurately represent the error with respect to step number. Root mean square values are therefore thought to be more precise in achieving this, and is utilised over the method of simple averages.

$$\log(rms(Error)) = m\log(step\_size) + C, \tag{29}$$

$$rms(Error) = \sqrt{\frac{1}{total\_time}\sum_{step=1}^{total\_steps}\left\{Error\Big[(step)(step\_size)\Big]\right\}^2} \tag{30}$$

By plotting the logarithm of the rms error with respect to the total steps, the linear trend is believed to be most accurately portrayed (Figure 27).

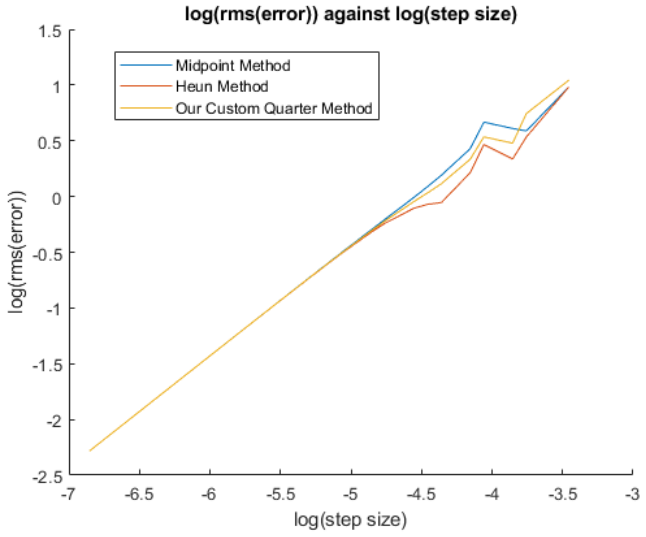


Figure 27: Graph Showing log(rms_error) Against log(step_size)

From the graph, it is observed that all three methods tend towards a well defined gradient of -1 and a y-intercept of 4.56. This gradient represents that the logarithm of the rms of the error is directly proportional to the logarithm of the step size.

Also, it can be noted that log(rms(error)) becomes more unpredictable and deviates more from the gradient as log(step size) increases. This is due to more inaccurate predictions made by the individual Runge-Kutta methods as a result of the diminishing number of total steps, causing the estimated output to look almost totally different from the exact output.

2 Part 2 - RLC Circuit

2.1 Background

Exercise 3. *Write a matlab script called RLC script.m and a matlab function called RK4.m. In the script you should set up the two coupled first order ODEs in q and q̇ to solve the RLC second order ODE (1) for q. The script will include a call to RK4.m, the matlab function you will write to implement the classic fourth-order Runge-Kutta algorithm for any system of two coupled first order equations.*

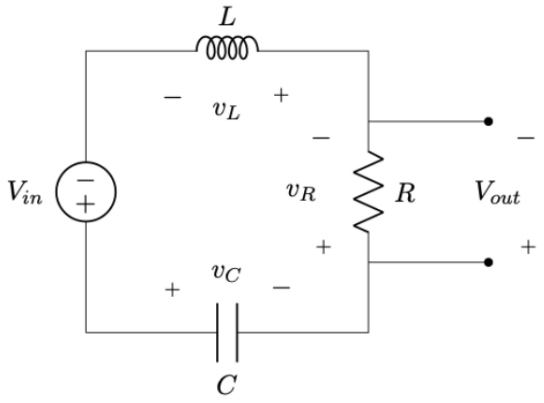


Figure 28: RLC Circuit used in this section

The following ODE can be derived from this circuit

$$V_{in}(t) = v_L(t) + v_R(t) + v_C(t) = \frac{Ldi_L(t)}{dt} + Ri_L(t) + \frac{1}{2\pi C}\int_0^\infty i_L(t)dt \tag{31}$$

$$V_{in}(t) = L\frac{d^2(q_C(t))}{dt2} + R\frac{d(q_C(t))}{dt} + \frac{q_C(t)}{C} \tag{32}$$

With :

1. State $q_C(t)$
2. Input $V_{in}(t)$
3. Output $V_{out}(t) = V_R = R\frac{d(q_C(t))}{dt}$

2.2 Circuit Analysis

The transfer function of the RLC circuit is :

$$\frac{Y}{X} = \frac{j\omega RC}{1 + j\omega RC + (j\omega)^2 LC} \tag{33}$$

X corresponds to the input voltage V_{in}, and Y is the voltage across the resistor. *Using Nodal Analysis at X, find the transfer function* $\frac{Y}{X}$

$$\begin{aligned} YR - XR &= -Y(\frac{1}{j\omega C} + j\omega L) \\ Yj\omega RC + Xj\omega RC &= -Y(1 + (j\omega)^2 LC) \\ Y(1 + j\omega RC + (j\omega)^2 LC) &= Xj\omega RC \end{aligned} \tag{34}$$

Therefore:

$$\frac{Y}{X} = \frac{j\omega RC}{1 + j\omega RC + (j\omega)^2 LC}$$

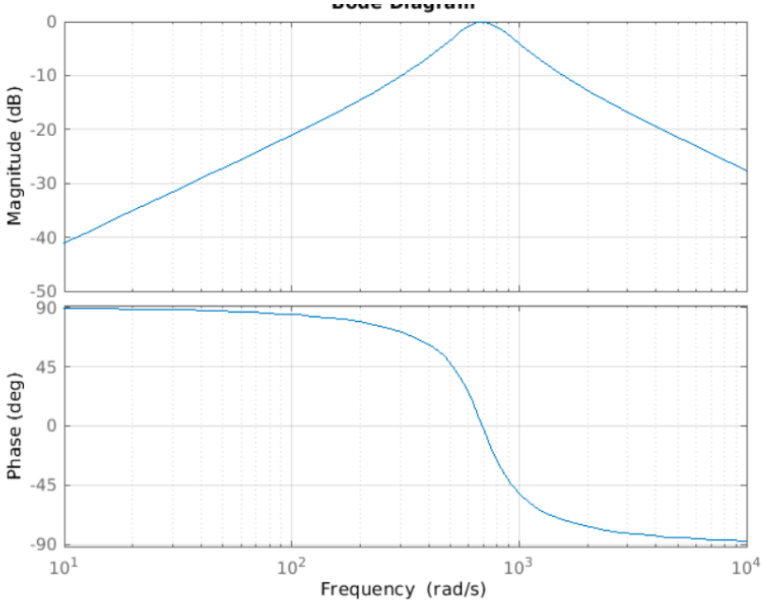


Figure 29: Bode Plot for the RLC circuit

1. Resonance frequency $\frac{1}{\sqrt{LC}} = 690.07rad = 109.83Hz$
2. Damping factor $\zeta = \frac{b*sgn(a)}{4ac} = \frac{R}{2}\frac{\sqrt{C}}{\sqrt{L}} = 0.302$

The damping factor $|\zeta| < 1$. This means that the quadratic on the denominator cannot be factorized, and the system is **underdamped**. Taking the inverse Laplace transform

for the input $\frac{1}{s}$ yields the exact solution for a unit step input and hence, the transient response. The exact solution will include an exponential envelope (see graphs below) and some sine terms.
For **critical damping**, $\zeta = 1$. To achieve this we chose $R = 2000\Omega, C = 300nF, L = 300mH$. Resonant frequency $= 530.5Hz$

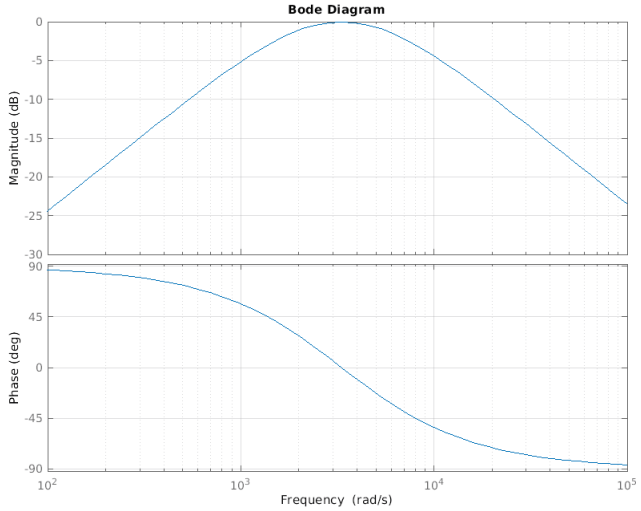


Figure 30: Bode Plot for the RLC circuit - critical damping

For **overdamping** $\zeta > 1$. To achieve this we chose $R = 2000\Omega, C = 3uF, L = 800mH$. $\zeta = 1.94$. Resonant frequency $= 102.7Hz$.

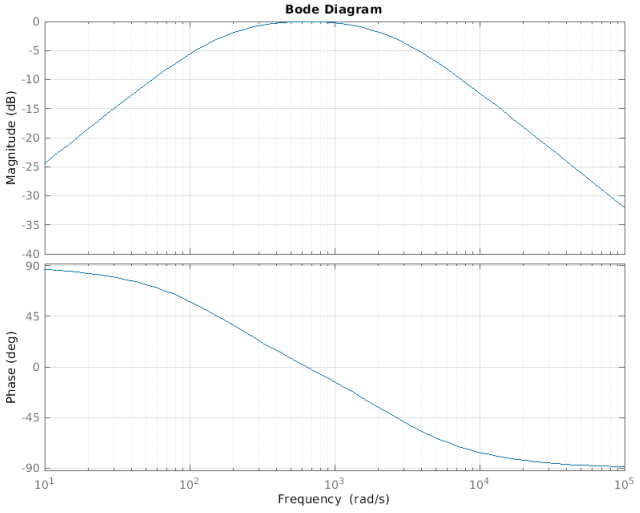


Figure 31: Bode Plot for the RLC circuit - over damping

From this we can see that critical damping plot has a symmetrical magnitude response

whereas the under and over damping plots are skewed slightly. For the overdamping magnitude response we see that the peak is flattened, the gradient is slower, meaning all frequencies are attenuated less. In practice this means that lots of frequencies are being allowed through the system, so the system takes a long time to stabilise. For the underdamping graph (fig 30) the peak is narrower, and the gradient is faster meaning that frequencies are attenuated more. In practice this means that only a few frequencies are being allowed through the system, usually close to the resonant frequency, meaning that the system will oscillate more before coming to a steady state.

Converting the 2^{nd} order ODE to a system of two coupled 1^{st} order ODEs

The fourth order Runge-Kutta method requires the second order ODE to be expressed as a system of two coupled first order ODEs. In order to do so, we start by substituting $y(t) = q_C(t)$ and $z(t) = q'_C(t)$. This allows us to write:

$$Lz'(t) + Rz(t) + y(t) = V_{in}(t) \tag{35}$$

Since $z(t)$ is the derivative of $y(t)$ we get the coupled system of equations as:

$$\begin{aligned} y'(t) &= z(t) \\ z'(t) &= \frac{V_{in}(t) - Rz(t) + \frac{1}{C}y(t)}{L} \end{aligned} \tag{36}$$

Which are both functions of $x(t)$, $y(t)$ and $z(t)$ as required.

Physically, $y(t)$ and $z(t)$ represent the charge in the capacitor and the rate of change of charge i.e. the current through the capacitor respectively.

2.3 MATLAB Script for Fourth Order Runge Kutta

The goal of this exercise is to solve any system of two coupled first order differential equations of the form $z' = f_1(x, y, z)$; $y' = f_2(x, y, z)$ In order to implement the solver, we proceed similarly to the previous exercise, applying the following formulas.

$$\begin{aligned} y_{(i+1)} &= y_i + \frac{1}{6}(k_0 + 2k_1 + 2k_2 + k_3) \\ z_{(i+1)} &= z_i + \frac{1}{6}(l_0 + 2l_1 + 2l_2 + l_3) \end{aligned} \tag{37}$$

The values of k_i and l_i are calculated as:

1. $k_0 = hf(x_i, y_i, z_i)$
2. $l_0 = hg(x_i, y_i, z_i)$
3. $k_1 = hf(x_i + \frac{1}{2}h, y_i + \frac{1}{2}k_0, z_i + \frac{1}{2}l_0)$
4. $l_1 = hg(x_i + \frac{1}{2}h, y_i + \frac{1}{2}k_0, z_i + \frac{1}{2}l_0)$
5. $k_2 = hf(x_i + \frac{1}{2}h, y_i + \frac{1}{2}k_1, z_i + \frac{1}{2}l_1)$
6. $l_2 = hg(x_i + \frac{1}{2}h, y_i + \frac{1}{2}k_1, z_i + \frac{1}{2}l_1)$
7. $k_3 = hf(x_i + h, y_i + k_2, z_i + l_2)$
8. $l_3 = hg(x_i + h, y_i + k_2, z_i + l_2)$

For each step these parameters are recalculated. They not only depend on the current values of x_i, y_i and z_i but also on each other, leading to a similar notion of predictors (k_0, l_0) and correctors (k_{1-3}, l_{1-3}) as for the second order Runge Kutta method.

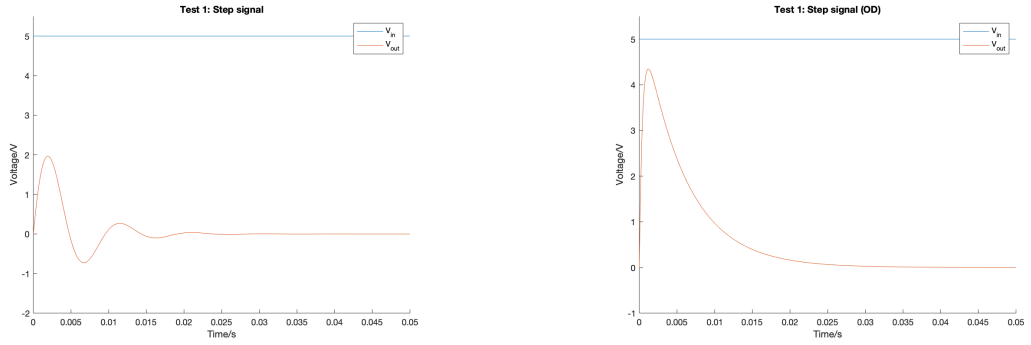
Matlab Code for Fourth Order Runge Kutta

```
function [x, y, z] = RK4(ODE_y, ODE_z, h, final_val, xi, yi, zi)
    % ODE Coupled second order ODE solver
    % @param ODE_y The ODE to be solved in the form y' = f(x, y, z)
    % @param ODE_z The ODE to be solved in the form z' = g(x, y, z)
    % @param step_size aka h The distance on the x-axis between two
    % consecutive steps
    % @param final_val The x-value up to which the ODE is evaluated
    % @param yi The initial y-value
    % @param zi The initial z-value


    % Calculate the number of steps
    N = round(final_val / h);

    % Initialise output arrays
    x = zeros(1, N);
    y = zeros(1, N);
    z = zeros(1, N);
    % xi is defined to be 0
    x(1) = xi;
    y(1) = yi;
    z(1) = zi;


    for i = 1 : N - 1
        % Calculate the coefficients
        k0 = h * ODE_y( x(i), y(i), z(i));
        l0 = h * ODE_z(x(i), y(i), z(i));
        k1 = h * ODE_y(x(i) + (0.5 * h), y(i) + (0.5 * k0), z(i)
            + (0.5 * l0));
        l1 = h * ODE_z(x(i) + (0.5 * h), y(i) + (0.5 * k0), z(i)
            + (0.5 * l0));
        k2 = h * ODE_y(x(i) + (0.5 * h), y(i) + (0.5 * k1), z(i)
            + (0.5 * l1));
        l2 = h * ODE_z(x(i) + (0.5 * h), y(i) + (0.5 * k1), z(i)
            + (0.5 * l1));
        k3 = h * ODE_y(x(i) + h, y(i) + k2, z(i) + l2);
```

```
34          l3 = h * ODE_z(x(i) + h, y(i) + k2, z(i) + l2);
35
36          x(i + 1) = x(i) + h;
37          y(i + 1) = y(i) + ((1.0 / 6.0) * (k0 + 2*k1 + 2*k2 + k3)
                );
38          z(i + 1) = z(i) + ((1.0 / 6.0) * (l0 + 2*l1 + 2*l2 + l3)
                );
39      end
40  end
```

2.4 Output

Step Signal Input for RK4

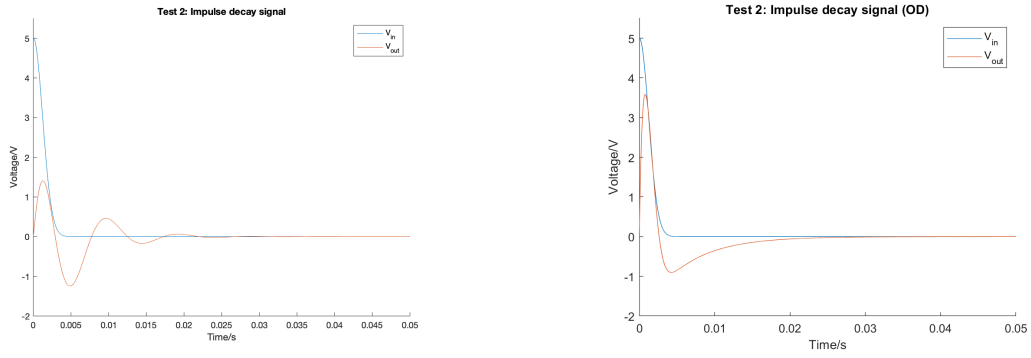


(a) Step signal response - underdamped (b) Step signal response - overdamped

Figure 32: Step Signal Response

The rising edge of the Heaviside function contains an infinite spectrum of frequencies which excite the circuit. In the underdamped case, the circuit starts oscillating at its natural (resonance) frequency, with the oscillations dying down until the steady state is reached. The overdamped response rises up to almost 4.5V before exponentially decaying to back to 0V. Since the steady state response is 0, the observed output is the transient only.

Impulsive signal and decay input for RK4



(a) Impulsive signal response - underdamped (b) Impulsive signal response - overdamped

Figure 33: Impulse Signal Responses

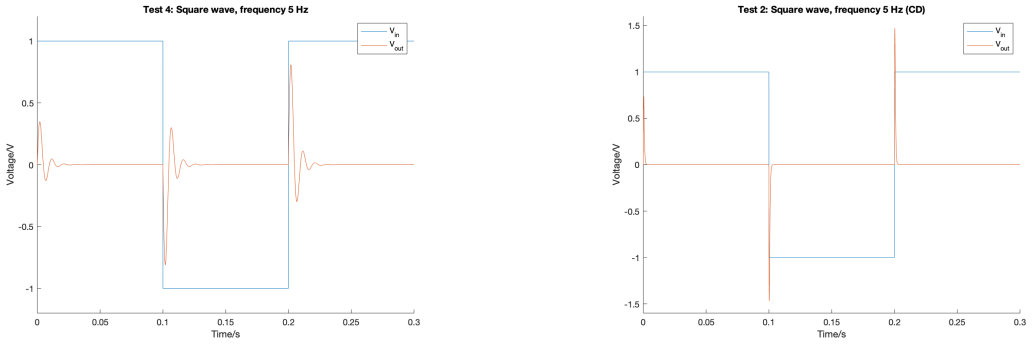
The response is very similar to that of the step input with the steady state being reached at a similar time (around 0.02s). The initial peak in the output is considerably lower (halved) while the following negative peak is more pronounced for both systems. This

becomes especially obvious from the overdamped response where a second (negative) peak has been introduced that was not present in the step response.

From these observations, the gradient of the output and speed at which it reaches the steady-state response has a relation with the gradient of the input signal.

Square wave input for RK4

Square Waves at Low Frequencies (5Hz)



(a) 5Hz underdamped (b) 5 Hz critically damped

Figure 34: Low Frequency Square Wave Response

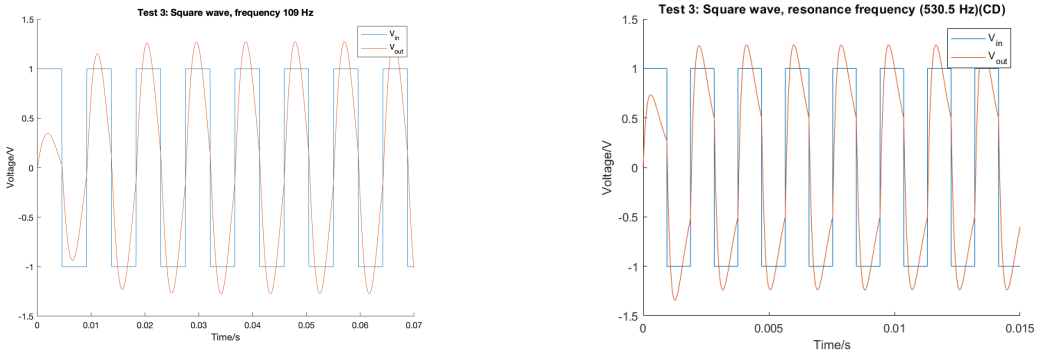
At low frequencies, the period of the wave is much longer than the transient response of the circuit meaning that all transients will have died down before the next rising or falling edge. Since the steady state response for any constant input voltage is 0V, in the case of a square wave, the output decays to 0V after each edge. In the critically damped case, the transients decay much faster, as the transient response for the critically damped circuit decays about 10 times faster, lasting around 0.002s

For both the underdamped and critically damped cases, the initial peak are about half the magnitude of the subsequent ones. This is due to the initial change in input voltage being halve as big - from 0V to 1V rather than from 1V to -1V (or -1V to 1V). We further observe that for a negative input gradient the response appears the same but mirrored.

Square Waves at the Resonant Frequency

In contrast to the low frequency case above, the period of the input wave is too short for the transient to decay fully (and reach steady state) before the next rising or falling edge, leading to a steady state response as the sum of all the individual responses to the frequencies that make up the square wave. This is true independent of the input frequency but for low frequencies it may be of interest to look at the individual edge responses.

Even though the square wave has an infinite frequency spectrum, its frequency spectrum is dominated by the fundamental frequency i.e. the frequency of the square wave itself. At resonance, the transfer function $\frac{Y}{X} = 1$ leading to a high output amplitude. We observe the output amplitude $A_{out} > 1$ due to constructive interference with the responses due to the less prominent frequency components contained in the input square wave. We notice further that the amplitude is higher in the critically damped case. This is since, as seen



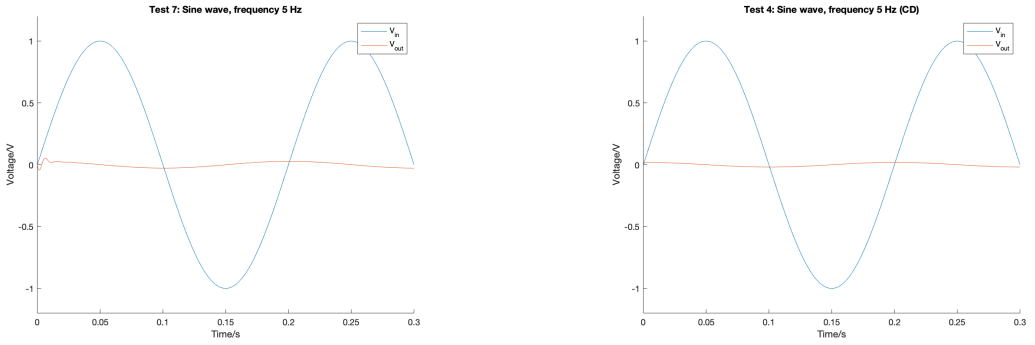
(a) 109Hz underdamped (b) 530.5Hz critically damped

Figure 35: Resonant Frequency Square Wave Response

in fig 32, the transfer function peak around the resonance frequency is higher, meaning that frequencies around the resonance are less attenuated.

Sine wave input for RK4

Sine Waves at Low Frequencies (5Hz)

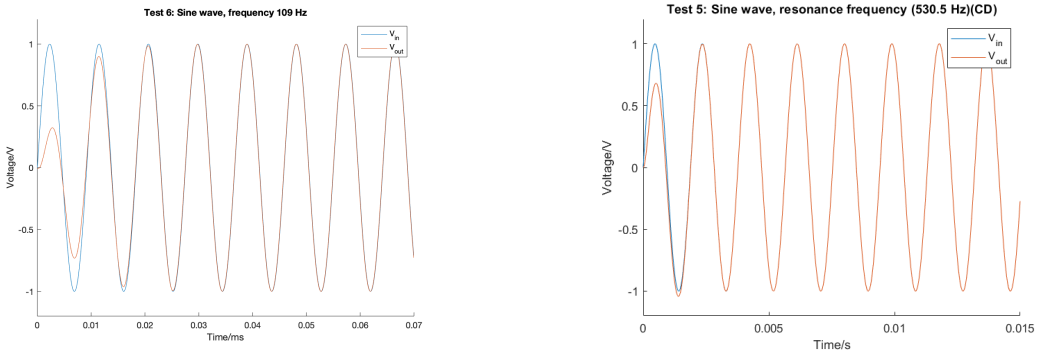


(a) 5Hz underdamped (b) 5Hz critically damped

Figure 36: Low Frequency Sine Wave Response

Compared to the square waves, the transient for sine wave inputs are less prominent since there is no instant rising edge as there is for the square wave or step input. The output is starkly attenuated making it difficult making it difficult to see the phase shift. Since $\frac{Y}{X}(5 \cdot 2\pi) = 7.582 \times 10^{-4} + 0.0275j$ in the steady state we get a sinusoid with amplitude 0.0275V and phase 88.42°.

Sine Waves at the Resonant Frequency

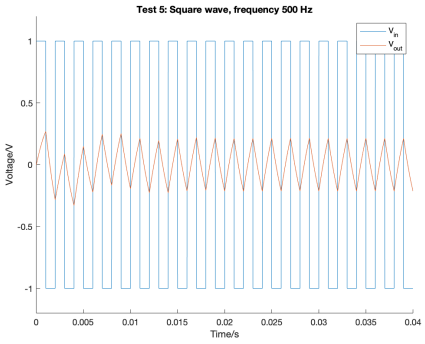


(a) 109Hz underdamped (b) 530.5Hz critically damped

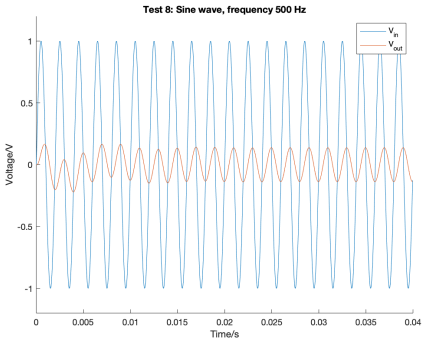
Figure 37: Resonant Frequency Sine Wave Response

At the resonant frequency (109Hz) we can observe that, similarly, the output voltage slowly reaches a steady state. The graphs above clearly show an output signal amplitude of 1V and phase shift of 0° as is expected at the resonance frequency (see transfer function plots - figs 30, 31, 32)

2.5 High Frequency waves



(a) Square wave at 500Hz



(b) Sine wave at 500Hz

Figure 38: High Frequency Waves

For frequencies much higher than the resonance frequency, we observe a much lower output amplitude as well as a phase shift. This is expected as $\frac{Y}{X}(500 \cdot 2\pi) = 0.0190 - 0.1367j$ with magnitude 0.1380 and phase $-82.07°$.

The envelope of the output wave itself follows the shape of a transient before reaching a steady state. This can be seen for both the square and sine waves at high frequencies. However, as the square wave (unlike the sine wave) has discontinuities, the output amplitude of the square wave is higher than that of the sine wave; this is due to the response to the square wave's high frequency components, as described above. This can be seen in the graphs by the peak amplitude of 0.20V for the square wave, compared to 0.14V for the sine wave.

2.6 Analysis

Observation

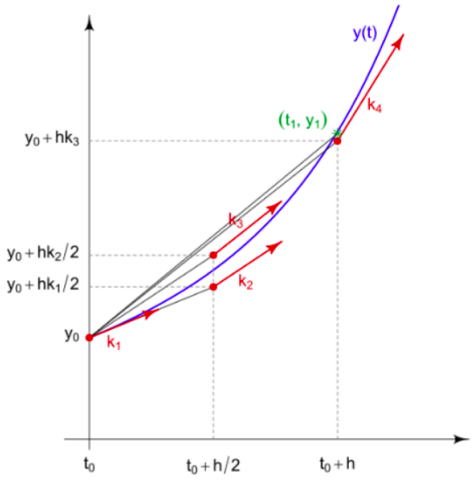


Figure 39: Visualising each coefficient vector for Fourth order Runge Kutta method [3]

In the figure above, the blue line represents the exact output we are trying to estimate with only the gradient at the point $t0 = y'(0)$ being known exactly to solver.

Similar to how the introduction of a single corrector for the second order Runge Kutta (RK2) methods reduces the error in the approximation compared to Euler's method, the fourth order method (RK4) outlined above yields a further improvement to the RK2 methods. It does this by replacing the single corrector with 3 correctors (k_2-k_4), where k_2 depends on the predictor k_1, k_3 on k_2 and k_4 on k_3 respectively. Note that this applies for solving a **single first order ODE** using the RK4 method.

As we are interested in solving a system of two **coupled first order ODEs**, the method is required to produce an output for both ODEs. In order to do so, a second set of predictors (l_1) and correctors (l_2-l_4) is introduced, with each k_{i+1} and l_{i+1} depending on both k_i and l_i.

In the specific case of solving a second order differential equation, we know that $y'(t) = z(t)$ i.e. one of the output functions will be the gradient of the other.

3 Part 3 – Relaxation

3.1 Background

Exercise 4. *Write a matlab script called relaxation.m to implement the relaxation method for Laplace's equation on the unit square, with simple boundary conditions.*

Proof. To solve the Laplace equation : $\frac{\delta^2 u}{\delta y^2} + \frac{\delta^2 u}{\delta y^2} = 0$ We approximate the derivatives using the central difference.

$$\begin{aligned} \frac{\delta^2 u(x_y, y_j)}{\delta x^2} &= \frac{u(x_{i+1}, y_j) - 2u(x_i, y_j) + u(x_{i-1}, y_j)}{h^2} = \frac{U_{i+1}^j - 2U_i^j + U_{i-1}^j}{h^2} \\ \frac{\delta^2 u(x_y, y_j)}{\delta y^2} &= \frac{u(x_i, y_{j+1}) - 2u(x_i, y_j) + u(x_i, y_{j-1})}{h^2} = \frac{U_i^{j+1} - 2U_i^j + U_i^{j-1}}{h^2} \end{aligned} \tag{38}$$

Substituting into the original equation we get:

$$U_{i+1}^j + U_{i-1}^j + U_i^{j+1} + U_i^{j-1} - 4U_i^j = 0 \tag{39}$$

The Laplace equation is on a grid with $(n-1)(m-1)$ squares of length h, making the limits $a = nh, b = mh \rightarrow 0 \leq x \leq a, 0 \leq y \leq b$.

We are given u at the boundaries as:

$$\begin{aligned} \phi_1(x) &= u(x, 1) = x \\ \phi_2(y) &= u(1, y) = y \\ \phi_3(x) &= u(x, 0) = x \\ \phi_4(y) &= u(0, y) = y \end{aligned} \tag{40}$$

which gives us the values for the outside lines of the grid. We set all points on the inside of the grid to an initial value $k = 0$. Then take each point as the average of its four nearest neighbours. We decided to start with 0 instead of $k = avg(AllPoints)$ as it gives better results when the expected output is closer to 0.

For each interior point

$$(U_i^j)_{new} = (U_i^j)_{old} + r_i^j \tag{41}$$

where r is equal to $r_i^j = \frac{(U_{i+1}^j + U_{i-1}^j + U_i^{j+1} + U_i^{j-1} - 4U_i^j)}{4}$
We set a desired accuracy and stop when every residual is **absolutely less** than this - i.e.

$$|r_i^j| < accuracy \tag{42}$$

□

3.2 MATLAB script for relaxation method

One thing to note is that after attempting to use the method above, we found it was easier (and necessary for SOR) to update the new value directly, rather than adding it to the grid of residual at the end of the two inner loops, meaning that the equation for the interior point becomes:

$$(U_i^j)_{new} = \frac{1}{4}(U_i^j)_{old} \tag{43}$$

and the test for convergence is therefore:

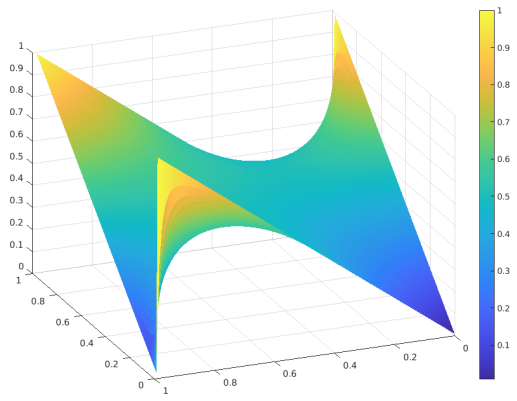
$$accuracy > |(U_i^j)_{old} - (U_i^j)_{new}| \tag{44}$$

```
function [grid] = relaxation(grid_size, b1, b2, b3, b4, e)
    % The grid is a square of size grid_size * grid_size
    % b1 to b4 are the boundary functions called phi in the
       slides
    % b1(0, y); b2(1, y); b3(x, 0); b4(x, 1)
    % e is the required accuarcy
    x_array = 1 : grid_size;
    x_array = x_array / grid_size;
    b1_vals = arrayfun(b1, x_array);
    b2_vals = arrayfun(b2, x_array);
    b3_vals = arrayfun(b3, x_array);
    b4_vals = arrayfun(b4, x_array);
    k = mean(b1_vals + b2_vals + b3_vals + b4_vals);

    % Create the grid and set the boundary values,
    grid = repmat(k, grid_size, grid_size);
    grid(1,:) = b1_vals;
    grid(grid_size,:) = b2_vals;
    grid(:,1) = b3_vals;
    grid(:,grid_size) = b4_vals;

    % Keep averaging until the required accuray is achieved
    done = false;
    residuals = grid;
    while ~done
        done = true;
        for j = 2 : grid_size - 1
            for i = 2 : grid_size - 1
                residuals(i, j) = 0.25 * (grid(i+1, j) + grid(i
                   -1, j) + grid(i, j+1) + grid(i, j-1));
```

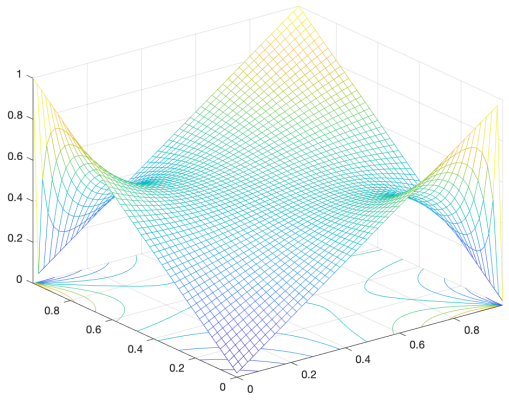
```
29                      r = abs(grid(i,j)-residuals(i, j));
30                      if r >= e
31                          done = false;
32                      end
33                      grid(i, j) = residuals(i, j);
34                  end
35              end
36          end
37      end
```

3.3 Output graphs

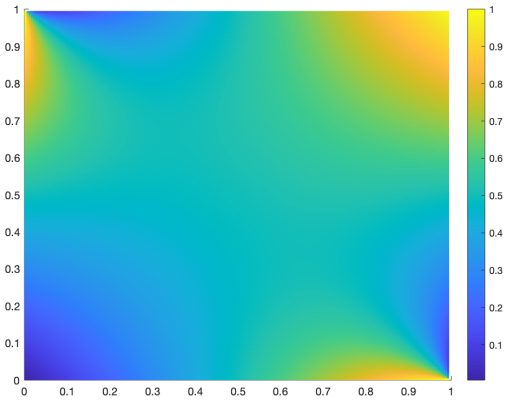
Initially the grid size was set to 100, required accuracy to 0.000001. We obtain the following:



(a) Shaded Side Grid Plot



(b) Front Grid Plot



(c) 2D Contour Plot

Figure 40: Relaxation Output Plots with Initial Values

This is to be expected given the initial conditions. The sides of the grid show the straight line equations; we also notice at the (0,1) boundary the discontinuity where one of the boundary conditions = 1, and the other = 0. The centre of the plot is about 0.5, which is to be expected as the functions average to 0.5.

Varying grid size

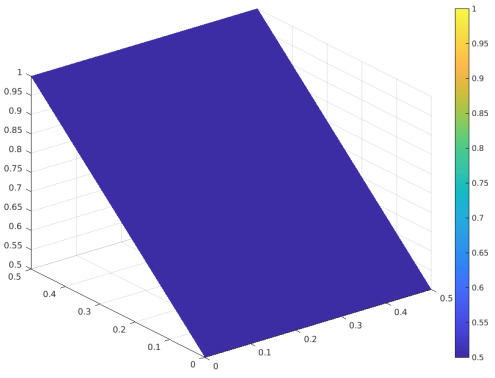


Figure 41: Relaxation method, grid size = 2

At grid size 2, there are no interior points, and therefore we just get a graph showing the boundary conditions for ϕ_i written in 3.1.

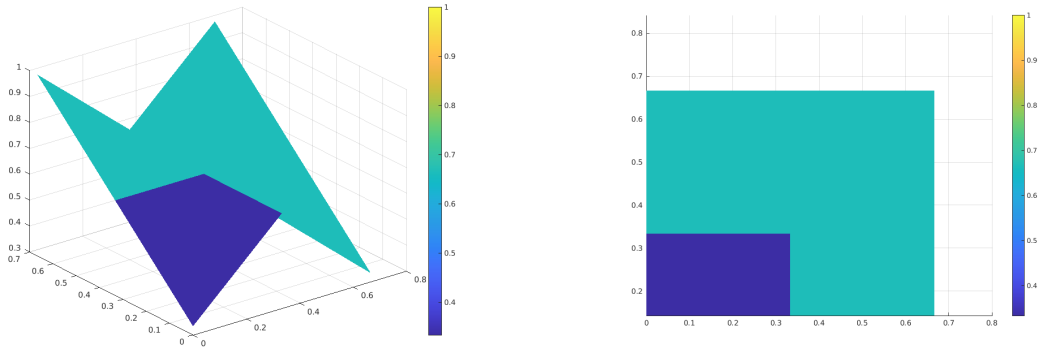


Figure 42: Relaxation method, grid size = 3

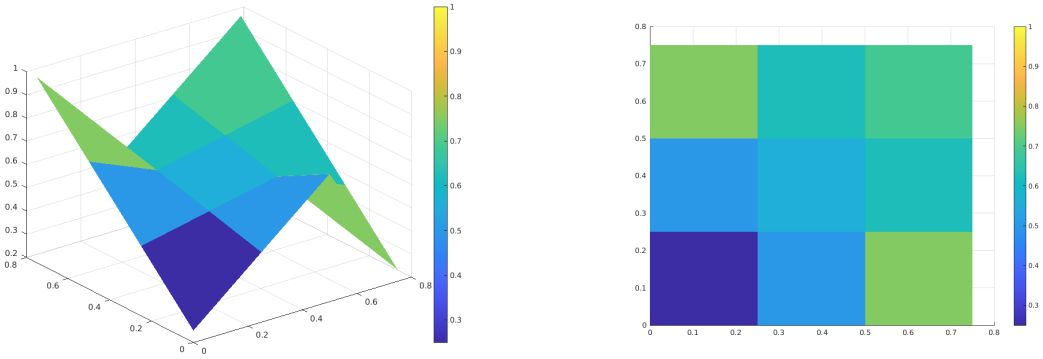


Figure 43: Relaxation method, grid size = 4

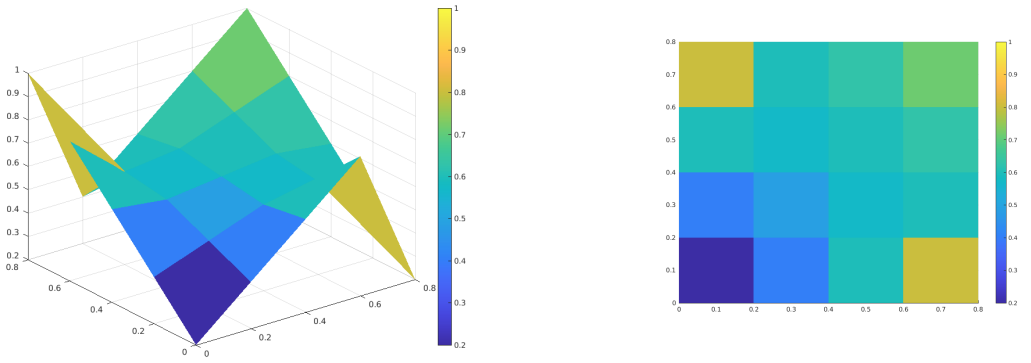


Figure 44: Relaxation method, grid size = 5

At grid size = 5, the shape becomes mostly correct. After this point the plot just becomes more accurate. At grid size 50 most of the 'squares' are no longer visible, except at the curves/edges.

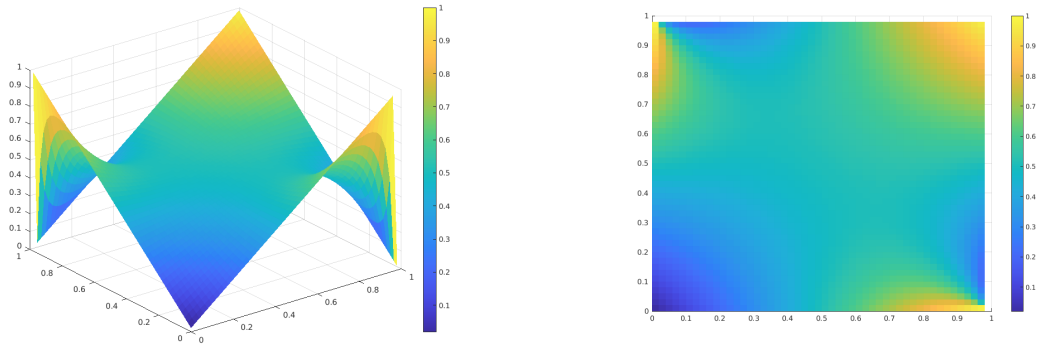


Figure 45: Relaxation method, grid size = 50

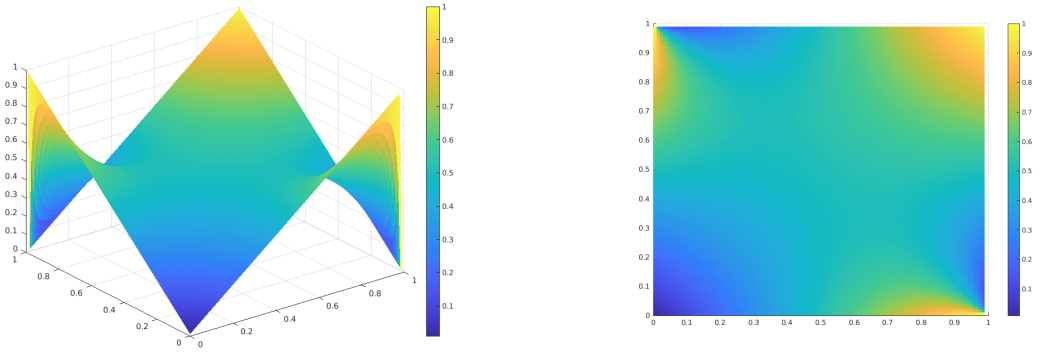


Figure 46: Relaxation method, grid size = 100

At 1000 all 'squares' in the graph have gone and the graph is completely smooth.

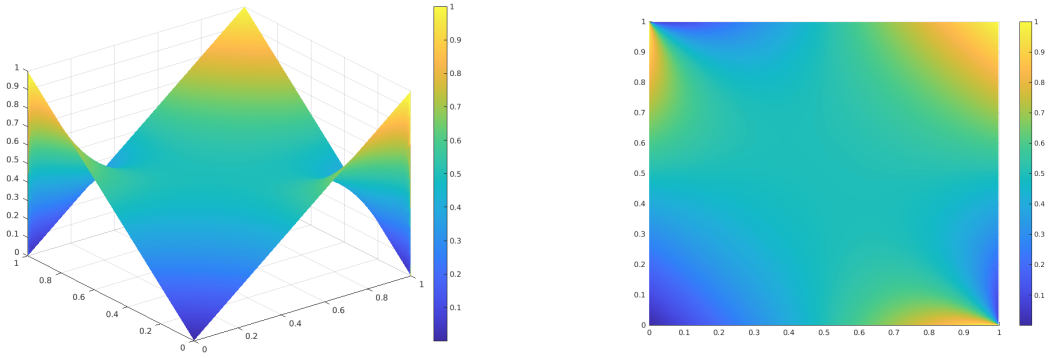


Figure 47: Relaxation method, grid size = 1000

Varying boundary conditions

Keeping the grid size constant (100), we chose different boundary conditions to see how the output would change.
Test 1 The boundary conditions were set to:

1. $\phi_1(y) = (0, y) = 1$
2. $\phi_2(y) = (1, y) = 1$
3. $\phi_3(x) = (x, 0) = 0$
4. $\phi_4(x) = (x, 1) = 0$

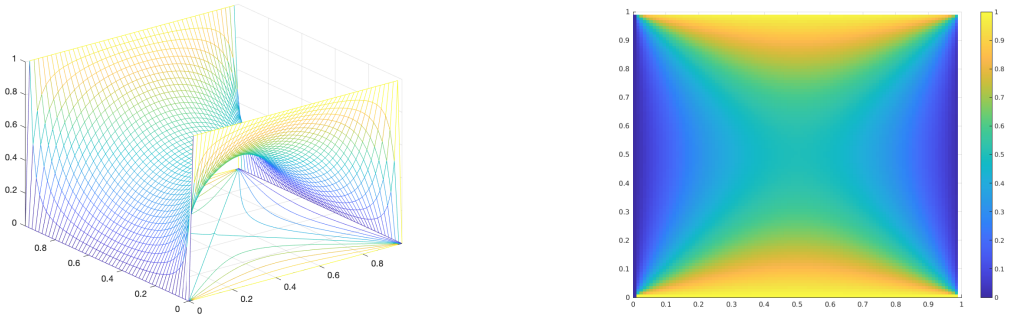


Figure 48: Relaxation method, varying boundary conditions, test 1

From the boundary conditions, we expect to get a 'tunnel' shape, as the two y-sides are set to 1, and the two x-sides are set to 0. This means from the residual calculations, the centre of the surface should be around 0.5, where it increases on the two parallel sides tending to 1, and decreases on the two sides perpendicular to these tending to 0. This description can quite clearly be seen in the heat map (top view) of the surface in figure 49.

Test 2 The boundary conditions were set to:

1. $\phi_1(y) = (0, y) = 0$
2. $\phi_2(y) = (1, y) = 1$
3. $\phi_3(x) = (x, 0) = 0$
4. $\phi_4(x) = (x, 1) = 0$

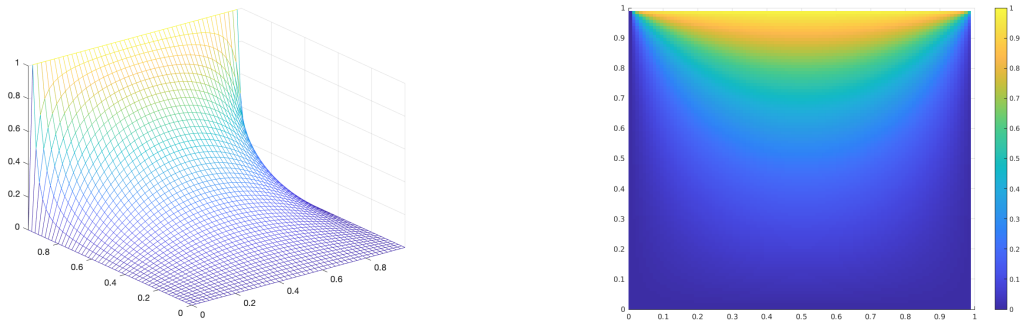


Figure 49: Relaxation method, varying boundary conditions, test 2

From this set of boundary conditions, we have three boundary sides of 0 value, and just boundary 2 of value 1. This means we should (and looking at figure 51 we do) get a single sided slope. The centre value should be about 0.25, as it will be the average of all the sides $= \frac{1}{4}$. There is also a slight curvature from the centre of the shape to the 0 boundaries on the left and right sides.

Test 3 The boundary conditions were set to:

1. $\phi_1(y) = (0, y) = y^2$
2. $\phi_1(y) = (1, y) = \frac{y}{2}$
3. $\phi_1(x) = (x, 0) = \frac{x}{2}$
4. $\phi_1(x) = (x, 1) = x^2$;

Here we can see the (1,1) boundary looks abnormal. Turning the picture around we get:

Boundary 2 is a straight line from (1, 0, 0) to (1, 1, 0.5), whereas boundary 4 is a quadratic curve between (0, 1, 0) and (1, 1, 1). This gives a discontinuity at x = 1, y = 1, and the value jumps between 0.5 and 1 in the z-direction. The graph (due to grid size as we found before) does not plot all the way up to (1, 1, z), but produces the points (0.98, 0.98, 1) and (0.96, 0.98, 0.49) and joins them with a non-vertical line segment. Interestingly at this point $\frac{d\phi}{dy}$ is undefined as the y value is constant for those two points.

Test 4 The boundary conditions were set to:

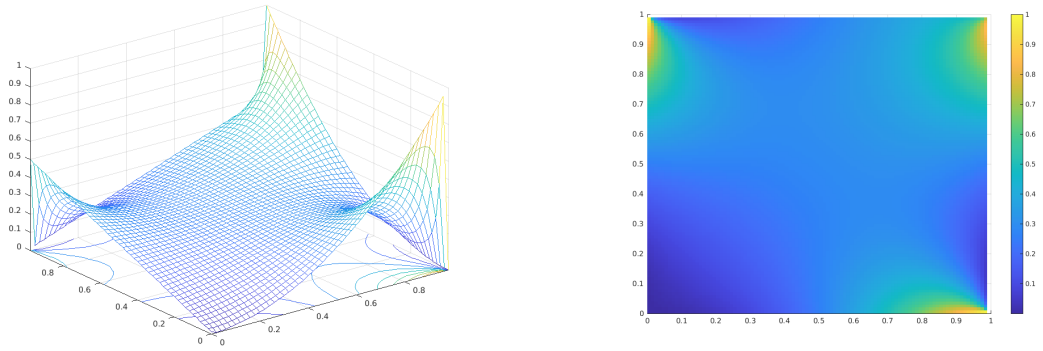


Figure 50: Relaxation method, varying boundary conditions, test 3

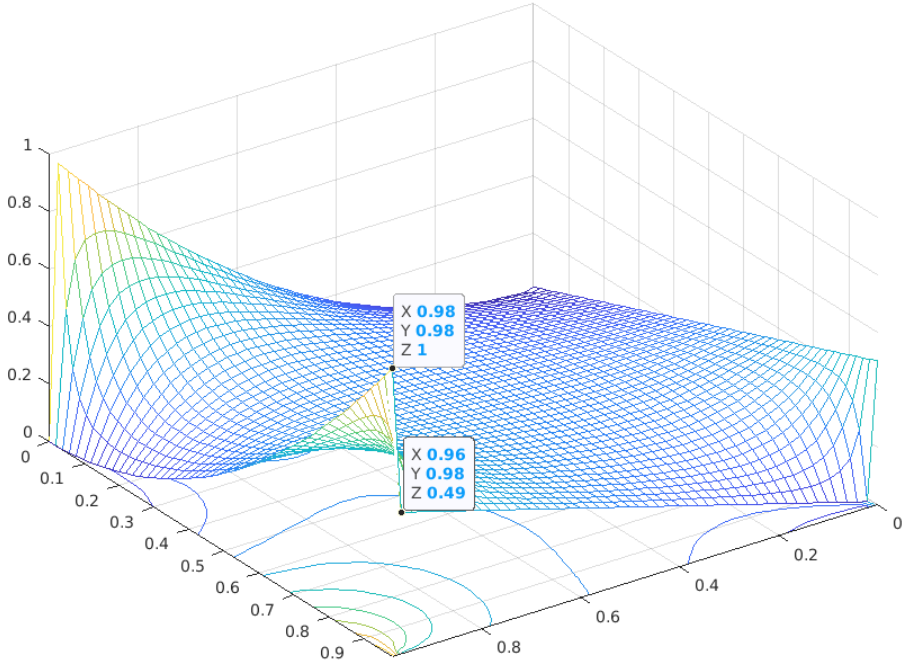


Figure 51: Relaxation method, varying boundary conditions, test 3, discontinuity

1. $\phi_1(y) = (0, y) = 2cos(4\pi y)$
2. $\phi_2(y) = (1, y) = 2cos(2\pi y)$
3. $\phi_3(x) = (x, 0) = 2cos(4\pi x)$
4. $\phi_4(x) = (x, 1) = 2cos(2\pi x)$

When all the boundary conditions are set to cosines, we expect to get peaks and troughs periodically in some form. Boundaries 1 and 3 have period $\frac{1}{2}$, whereas boundaries 2 and 4 have period 1. From this we can deduce that peaks should be formed (as $cos(\theta) = 1$) at

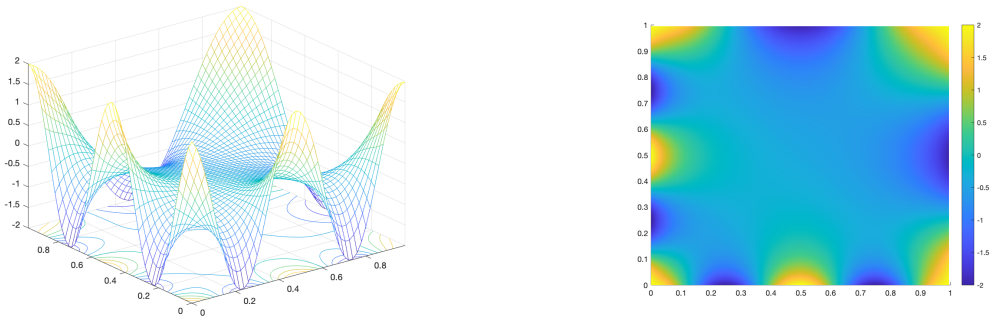


Figure 52: Relaxation method, varying boundary conditions, test 4

0, 0.5 and 1 along boundaries 1 and 3, whereas peaks will be formed at 0 and 1 only on boundaries 2 and 4. The peaks will all be of amplitude = 2, as the formula is $2cos(\theta)$. We can see clearly from the top down (heat) plot on the right of figure 53, that the output is symmetrical about the line y = x. This is to be expected as we set the two boundaries on (0,0) to the same period, and the two boundaries on (1,1) to the same (but different to previous) period. The centre of the plot is about 0, which is to be expected, as $A \cdot cos(\theta)$ averages to 0.

Test 5 The boundary conditions were set to:

1. $\phi_1(y) = (0, y) = 2sin(2\pi y)$
2. $\phi_2(y) = (1, y) = 2sin(2\pi y)$
3. $\phi_3(x) = (x, 0) = 2sin(2\pi x)$
4. $\phi_4(x) = (x, 1) = 2sin(2\pi x)$

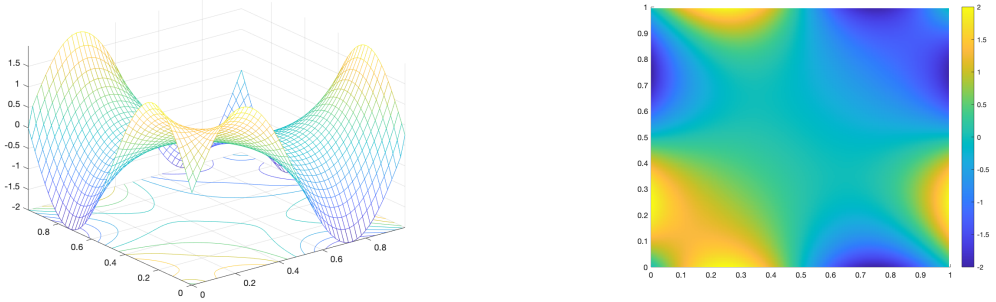


Figure 53: Relaxation method, varying boundary conditions, test 5

For this set of boundary conditions, we changed $cos(\theta)$ to $sin(\theta)$, and all the sine waves are of period 1. Sine is an odd function (unlike cosine which is even), and therefore unlike the previous plot, we expect this output to be asymmetrical along the boundaries. However,

since the boundaries follow the same form as the previous test (in that the periods of the sine waves are the same for the two at (0, 0, z) and (1, 1, z)) we would expect this plot to also be symmetrical along the line $y = x$. This is confirmed by the heat plot in figure 55. From the surface plot in figure 55, we can see that there is only one period on both sides which is what we expect from the boundary condition formulas. The centre of the plot is about 0, which is to be expected, as $A \cdot sin(\theta)$ averages to 0. Additionally, from the heat plot, we can see that the graph is inverted in the line $y = 1 - x$ (i.e. where there is a peak on $y < 1 - x$, there is a trough in the reflection $y > 1 - x$); this is because sine is an odd function.

3.4 Successive Over-Relaxation (SOR)

Exercise 5. *Implement the solutions in Exerise 4, using Successive Over-Relaxation, SOR. Track the computation time for different values of the discretization h, and compare convergence time between relaxation and SOR.*

To do successive over-relaxation we add in a relaxation parameter λ to speed up the convergence of the formula.

$$1 \leq \lambda \leq 2 \tag{45}$$

The optimal value of the relaxation parameter is given by [1]:

$$\lambda = \frac{2}{1 + \frac{\pi}{N}} \tag{46}$$

Where N is grid size.
The new formula for the residual becomes:

$$r_i^j = \frac{\lambda}{4} U_{i+1}^j + U_{i-1}^j + U_i^{j+1} + U_i^{j-1} - 4U_i^j \tag{47}$$

The new formula for the interior points becomes:

$$(U_i^j)_{new} = (1 - \lambda)(U_i^j)_{old} + r_i^j \tag{48}$$

3.5 Matlab Script for SOR method

```
function [grid] = SOR(grid_size, b1, b2, b3, b4, e)
    % same parameters as relaxation method
    x_array = 1 : grid_size;
    x_array = x_array / grid_size;
    b1_vals = arrayfun(b1, x_array);
    b2_vals = arrayfun(b2, x_array);
    b3_vals = arrayfun(b3, x_array);
    b4_vals = arrayfun(b4, x_array);
    k = mean(b1_vals + b2_vals + b3_vals + b4_vals) / 4;
    grid = zeros(grid_size, grid_size); % Create the grid and
       boundary values,
    grid(1,:) = b1_vals;
    grid(grid_size,:) = b2_vals;
    grid(:,1) = b3_vals;
    grid(:,grid_size) = b4_vals;

    h = 1/(grid_size + 1); %create relaxation parameter op
    relax = 2 - (pi*h);
    relax = 1;
    count = 0;
    done = false; % Keep averaging until the required accuray
       is achieved
    residuals = grid;
    while ~done
        done = true;
        for j = 2 :1: grid_size - 1
            for i = 2 :1: grid_size - 1
                residuals(i, j) = ((1 - relax) * grid(i, j)) +
                   (0.25 * relax * (grid(i+1, j) + grid(i-1, j)
                   + grid(i, j+1) + grid(i, j-1)));
                r = abs(residuals(i, j)-grid(i,j));
                if r >= e
                    done = false;
                end
                grid(i, j) = residuals(i, j);
            end
        end
        count = count + 1;
    end
end
```

3.6 Output for Successive Over Relaxation

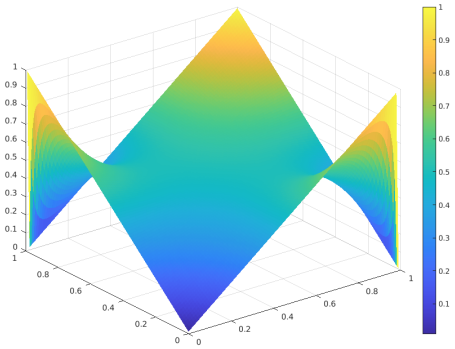
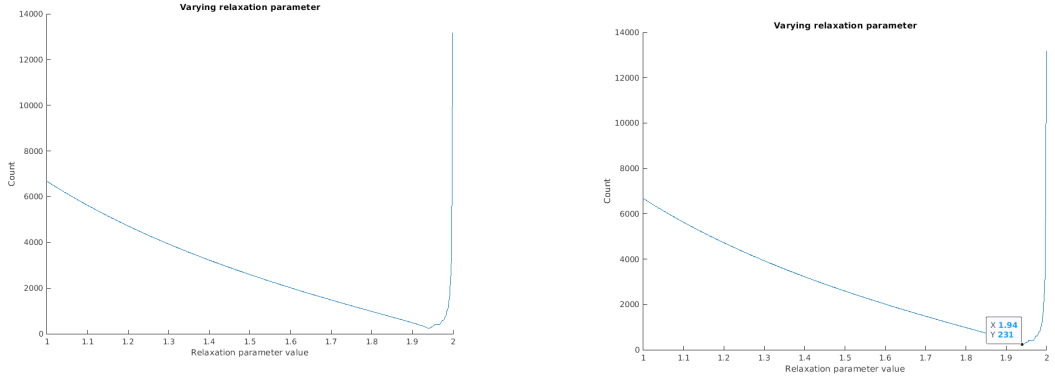


Figure 54: SOR initial output

As we can see, the SOR method outputs exactly the same as the relaxation method. This is what we expected, as SOR is just supposed to speed up the process, not provide a different result.

Comparing values of λ

As previously mentioned, $1 < \lambda < 2$, and the optimal value of $\lambda = \frac{2}{1+\frac{\pi}{N}}$. For a grid size of 100, this would be $\lambda = \frac{2}{1+\frac{\pi}{N}} = 1.9391$. To compare relaxation parameter with number of iterations (count) we ran the SOR script with values of λ incrementing by 0.001 between 1 and 1.999 (as the function does not compute when $\lambda = 2$). The following graph was obtained.



(a) Relaxation Output for different relaxation values

(b) Relaxation Output minimum datapoint

Figure 55: Relaxation Output Graphs

There is a clear trend towards the optimal value we calculated above, with count being largest at the two boundaries (and exponentially large near the 2 boundary); however,

the graph seems to deviate (its curve is no longer smooth) in the minimal section ($1.90 < \lambda < 1.99$). This may be due to the boundary conditions; to investigate this, we ran the code again, but with different boundary conditions.

Test 1 The boundary conditions were set to:

1. $\phi_1(y) = u(0, y) = 1$
2. $\phi_2(y) = u(1, y) = 1$
3. $\phi_3(x) = u(x, 0) = 0$
4. $\phi_4(x) = u(x, 0) = 0$

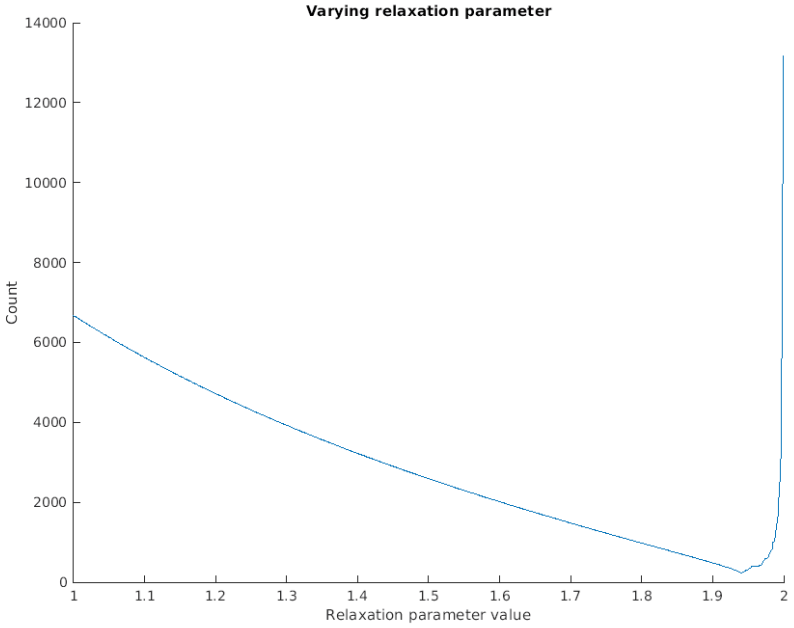


Figure 56: Relaxation method, varying relaxation parameter, test 1

From this (figure 58) we can see that the initial value at $\lambda = 1$ has increased from 5000 to 6700, but overall the shape has stayed the same, and the minimum is in the same place as originally at 1.94.

Test 2 The boundary conditions were set to:

1. $\phi_1(y) = (0, y) = 2cos(4\pi y)$
2. $\phi_2(y) = (1, y) = 2cos(2\pi y)$
3. $\phi_3(x) = (x, 0) = 2cos(4\pi x)$
4. $\phi_4(x) = (x, 1) = 2cos(2\pi x)$

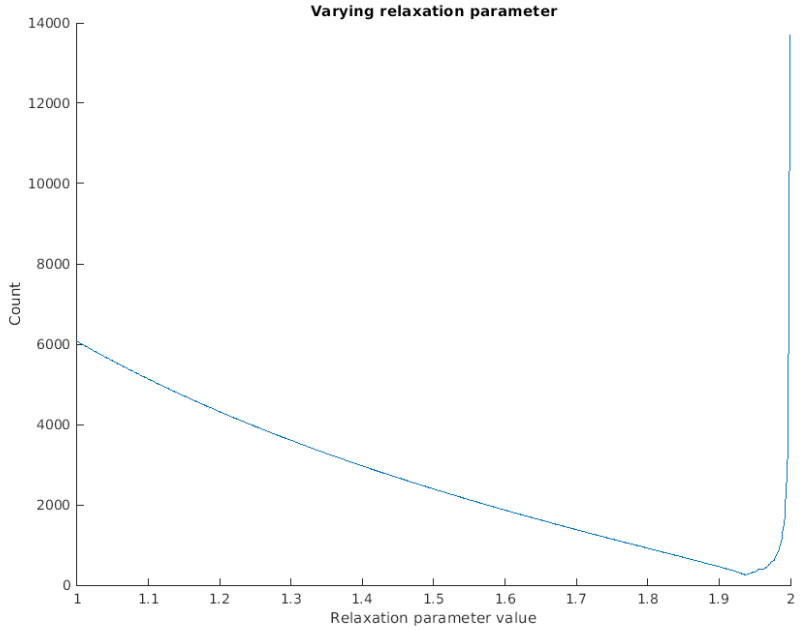


Figure 57: Relaxation method, varying relaxation parameter, test 2

From this (figure 59) we can see that the initial value at $\lambda = 1$ has increased from 5000 to 6000, but overall the shape has stayed the same, and the minimum is in the same place as originally at 1.94.

Test 3 The boundary conditions were set to:

1. $\phi_1(y) = (0, y) = y$
2. $\phi_2(y) = (1, y) = y$
3. $\phi_3(x) = (x, 0) = -x$
4. $\phi_4(x) = (x, 1) = -x$

From this (figure 60) we can see that the initial value at $\lambda = 1$ has decreased from 5000 to 2000, meaning that the initial gradient is less; the minimum has moved fairly significantly to 1.89. With this change we also notice that a change in gradient occurs at around $\lambda = 1.7$, and begins to increase until 1.75, before decreasing again until the minimum point.

In conclusion, we see that the graphs generally have a minimum value in the correct area, but that boundary conditions affect the shape of the graph as well as the minimum value. Although, the feature that boundary conditions mostly affected was the initial count value, and consequently the gradient of its descent. The most deviation was found in the test 3, whereas test 1 and 2 had very similar graphs (as well as the original). However, all of the graphs seemed to have a deviation in expected shape (they become slightly jagged)

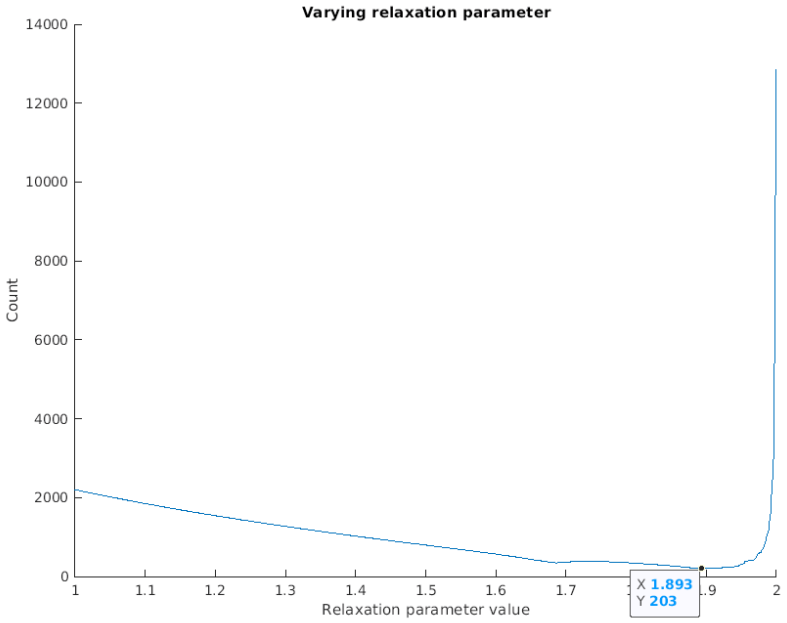


Figure 58: Relaxation method, varying relaxation parameter, test 3

in the area close to our calculated minimum (between 1.90 and 1.99).

Computation time for different values of h and comparison with relaxation method

Grid size is related to h by the following formula:

$$h = \frac{1}{GridSize + 1} \tag{49}$$

Since our function takes in grid size as a parameter rather than h, grid size was varied so we only changed the testbench, not the functions. The relaxation and SOR method were tested for grid sizes 2 – 500, and the following graph was obtained

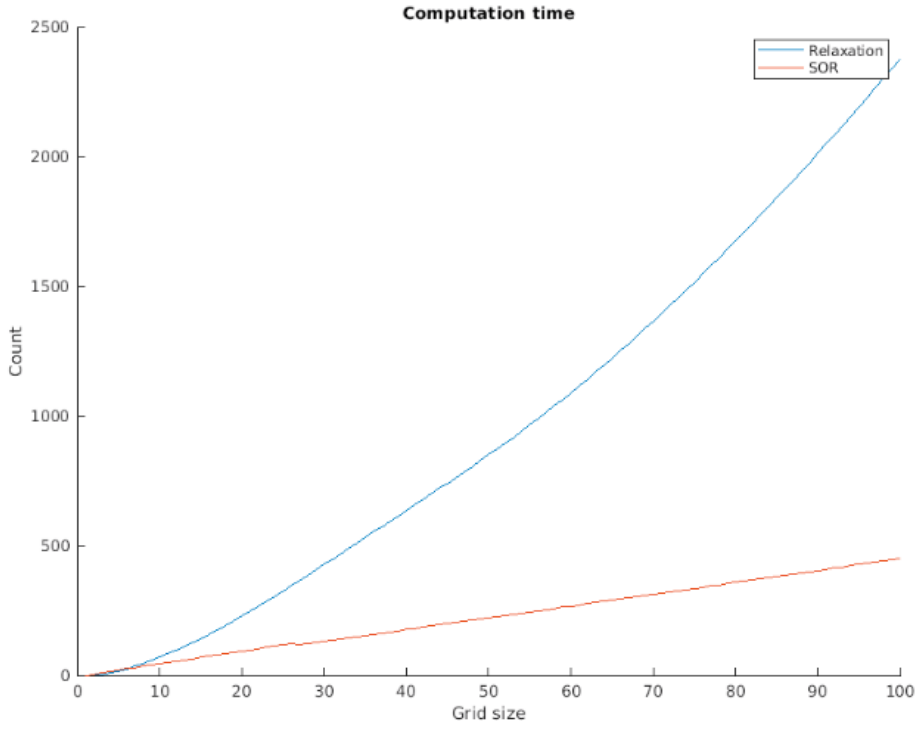


Figure 59: Relaxation vs SOR computation time

From this graph we can see that SOR follows a linear relationship, whereas relaxation is more similar to a quadratic relationship. For larger grid sizes (or smaller h values), the relaxation method therefore takes much longer than the SOR method. The relaxation method can only be used for small grid sizes, whereas SOR can be used even at very large (grid size > 100) without much computation overhead. This means that since SOR gives results that are indistinguishable from the relaxation method, whilst being much more effective are arriving at the solution, it is a much more desirable method to use.

List of Figures

Bibliography

[1] Nicole Nikas. *Using the Relaxation Method to solve Poisson's Equation*. [Online; accessed March 17, 2020]. Oct. 2015. URL: https://storm.cis.fordham.edu/~nicolenikas/CompPhysMidterm-copy%20(2).pdf.

[2] Wikipedia, the free encyclopedia. *Heun's method*. [Online; accessed March 17, 2020]. 2019. URL: https://en.wikipedia.org/wiki/Heun%27s_method#/media/File:Heun's_Method_Diagram.jpg.

[3] Wikipedia, the free encyclopedia. *Runge-Kutta methods*. [Online; accessed March 17, 2020]. 2019. URL: https://en.wikipedia.org/wiki/Runge%E2%80%93Kutta_methods#/media/File:Runge-Kutta_slopes.svg.

Appendices

1 Matlab Code

1.1 RC Circuit

RK2 code (RK2.m)

```
function [x_values, y_values] = RK2(ODE, step_size, final_val,
    xi, yi, RKMETHOD)
    % ODE solver
    % @param ODE The ODE to be solved in the form y' = f(x, y)
    % @param step_size aka h The distance on the x-axis between
        two
    % consecutive steps
    % @param final_val The x-value up to which the ODE is
        evaluated
    % @param xi The initial x-value
    % @param yi The initial y-value

    % Calculate the number of steps
    N = round((final_val - xi) / step_size);
    % Initialise output arrays
    x_values = zeros(1, N);
    y_values = zeros(1, N);
    x_values(1) = xi;
    y_values(1) = yi;

    %%%%%%% EDIT VALUE OF A FOR DIFFERENT METHODS %%%%%%
    % For Dynamic user input:

    a = RKMETHOD;
    % For heun
    % a = 0.5;
    % For Midpoint
    % a = 0;
    % For Ralston
    %a = 1/3;
    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

    b = 1-a;
    p = 1/(2*b);
    q = p;

    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

```
    % Using the formula y_(i+1) = y(i) + h*g(x_i, y_i, h)
    % Where g = a*k_1 + b*k_2
    % And k_1 = f(x_i, y_i)
    % And k_2 = f(x_i + p*h, y_i + q*k_1*h)
    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
    % Run for N - 1 iterations
    for i = 1 : N - 1
        K_one = ODE(x_values(i), y_values(i));
        x_values(i+1) = x_values(i) + step_size;

        K_two = ODE(x_values(i)+(p*step_size), y_values(i)+q*
            K_one*step_size);
        y_values(i+1) = y_values(i)+step_size*(a*K_one+b*K_two);

    end
    % Return the output arrays
end
```

Test script for RC circuit (RK2_script.m)

```
% Test file

% Define an array of ODEs to test (these should be function
    handles)

% For each ODE

% Hold on - make sure to set the graphs to differnt colours
%   For each set of input parameters
%       Run the ODE solver ie plot the graphs

% Todo: Add error calculation for the different inputs (perhaps
    store /
% output to a table


% Just a quick test to check that the solver is working
M1 = "Midpoint Method";
M2 = "Heun's Method";
M3 = "Our Custom Quarter Method";
M4 = "Input V_in";
%%%%%%%%%%TEST 1: Y = 2.5 %%%%%%%%%%%%%%%%%%

ODE = @(x, y) (2.5-y)*10;
figure; hold on;
[out_x1, out_y1] = RK2(ODE, 0.01, 1.5, 0, 5, 0);
a1 = plot(out_x1, out_y1);

[out_x2, out_y2] = RK2(ODE, 0.01, 1.5, 0, 5, 0.5);
a2 = plot(out_x2, out_y2);

[out_x3, out_y3] = RK2(ODE, 0.01, 1.5, 0, 5, 0.25);
a3 = plot(out_x3, out_y3);
input_x = zeros(1, 2001);
input_y = zeros(1,2001);
for i=1:2001
    input_x(i) = 0.00075*(i-1);
    input_y(i) = 2.5;
end
a4 = plot(input_x, input_y);
ylim([2, 5]);
```

```
40
41 legend([a1; a2; a3; a4], [M1; M2; M3; M4]);
42 hold off;
43
44 title('Test 1: Step signal');
45 xlabel('Time/ms');
46 ylabel('Voltage/V');
47
48
49 %%%%%%%%%%TEST 2: Y = Impulse and decay signal 1 %%%%%%%%%%%%%%%%%%
50 ODE = @(x,y) ((2.5)*exp(-(x^2/0.0001))-y)*10;
51 figure; hold on;
52 [out_x4, out_y4] = RK2(ODE, 0.03, 1.5, 0, 5, 0);
53 a4 = plot(out_x4, out_y4);
54
55 [out_x5, out_y5] = RK2(ODE, 0.03, 1.5, 0, 5, 0.5);
56 a5 = plot(out_x5, out_y5);
57
58 [out_x6, out_y6] = RK2(ODE, 0.03, 1.5, 0, 5, 0.25);
59 a6 = plot(out_x6, out_y6);
60
61 for i=1:2001
62     input_x(i) = 0.00075*(i-1);
63     input_y(i) = 2.5*exp(-((input_x(i))^2/0.0001));
64 end
65 a7 = plot(input_x, input_y);
66 ylim([0, 5]);
67
68 legend([a4; a5; a6; a7], [M1; M2; M3; M4]);
69 hold off;
70 title('Test 2: Impulse and decay signal 1');
71 xlabel('Time/ms');
72 ylabel('Voltage/V');
73
74 %%%%%%%%%%TEST 3: Y = Impulse and decay signal 2 %%%%%%%%%%%%%%%%%%
75 ODE = @(x,y) ((2.5)*(exp(-x/0.1)-y)*10);
76 figure; hold on;
77 [out_x4, out_y4] = RK2(ODE, 0.02, 1, 0, 5, 0);
78 a4 = plot(out_x4, out_y4);
79
```

```
80  [out_x5, out_y5] = RK2(ODE, 0.02, 1, 0, 5, 0.5);
81  a5 = plot(out_x5, out_y5);
82
83  [out_x6, out_y6] = RK2(ODE, 0.02, 1, 0, 5, 0.25);
84  a6 = plot(out_x6, out_y6);
85
86  for i=1:2001
87      input_x(i) = 0.0005*(i-1);
88      input_y(i) = 2.5*exp(-(input_x(i)/0.1));
89  end
90  a7 = plot(input_x, input_y);
91  ylim([0, 5]);
92
93  legend([a4; a5; a6; a7], [M1; M2; M3; M4]);
94  hold off;
95  title('Test 3: Impulse and decay signal 2');
96  xlabel('Time/ms');
97  ylabel('Voltage/V');
98
99  %%%%%%%%%%TEST 4: Y = Sine, period 100us %%%%%%%%%%%%%%%%%
100 ODE = @(x,y) ((5*sin(2*pi*(10)*x)-y)*10);
101 figure; hold on;
102 [out_x4, out_y4] = RK2(ODE, 0.001, 0.5, 0, 5, 0);
103 a4 = plot(out_x4, out_y4);
104
105 [out_x5, out_y5] = RK2(ODE, 0.001, 0.5, 0, 5, 0.5);
106 a5 = plot(out_x5, out_y5);
107
108 [out_x6, out_y6] = RK2(ODE, 0.001, 0.5, 0, 5, 0.25);
109 a6 = plot(out_x6, out_y6);
110
111 for i=1:2001
112     input_x(i) = 0.00025*(i-1);
113     input_y(i) = 5*sin(2*pi*(10)*input_x(i));
114 end
115 a7 = plot(input_x, input_y);
116
117 legend([a4; a5; a6; a7], [M1; M2; M3; M4]);
118 hold off;
119 title('Test 4: Sine, period 100us');
120 xlabel('Time/ms');
121 ylabel('Voltage/V');
```

```
122
123 %%%%%%%%%%TEST 5: Y = Sine, period 10us%%%%%%%%%%%%%%%%%%
124 ODE = @(x,y) ((5*sin(2*pi*(10^2)*x)-y)*10);
125 figure; hold on;
126 [out_x4, out_y4] = RK2(ODE, 0.0002, 0.2, 0, 5, 0);
127 a4 = plot(out_x4, out_y4);
128
129 [out_x5, out_y5] = RK2(ODE, 0.0002, 0.2, 0, 5, 0.5);
130 a5 = plot(out_x5, out_y5);
131
132 [out_x6, out_y6] = RK2(ODE, 0.0002, 0.2, 0, 5, 0.25);
133 a6 = plot(out_x6, out_y6);
134
135 for i=1:2001
136     input_x(i) = 0.0001*(i-1);
137     input_y(i) = 5*sin(2*pi*(10^2)*input_x(i));
138 end
139 a7 = plot(input_x, input_y);
140
141 legend([a4; a5; a6; a7], [M1; M2; M3; M4]);
142 hold off;
143 title('Test 5: Sine, period 10us');
144 xlabel('Time/ms');
145 ylabel('Voltage/V');
146
147 %%%%%%%%%%TEST 6: Y = Sine, period 500us%%%%%%%%%%%%%%%%%%
148 ODE = @(x,y) ((5*sin(2*pi*5*x)-y)*10);
149 figure; hold on;
150 [out_x4, out_y4] = RK2(ODE, 0.003, 0.6, 0, 5, 0);
151 a4 = plot(out_x4, out_y4);
152
153 [out_x5, out_y5] = RK2(ODE, 0.003, 0.6, 0, 5, 0.5);
154 a5 = plot(out_x5, out_y5);
155
156 [out_x6, out_y6] = RK2(ODE, 0.003, 0.6, 0, 5, 0.25);
157 a6 = plot(out_x6, out_y6);
158
159 for i=1:2001
160     input_x(i) = 0.0003*(i-1);
161     input_y(i) = 5*sin(2*pi*5*input_x(i));
162 end
163 a7 = plot(input_x, input_y);
```

```
164
165 legend([a4; a5; a6; a7], [M1; M2; M3; M4]);
166 hold off;
167 title('Test 6: Sine, period 500us');
168 xlabel('Time/ms');
169 ylabel('Voltage/V');
170
171 %%%%%%%%%%TEST 7: Y = Sine, period 1000us %%%%%%%%%%%%%%%%%
172 ODE = @(x,y) ((5*sin(2*pi*(1)*x)-y)*10);
173 figure; hold on;
174 [out_x4, out_y4] = RK2(ODE, 0.03, 3, 0, 5, 0);
175 a4 = plot(out_x4, out_y4);
176
177 [out_x5, out_y5] = RK2(ODE, 0.03, 3, 0, 5, 0.5);
178 a5 = plot(out_x5, out_y5);
179
180 [out_x6, out_y6] = RK2(ODE, 0.03, 3, 0, 5, 0.25);
181 a6 = plot(out_x6, out_y6);
182
183 for i=1:2001
184     input_x(i) = 0.0015*(i-1);
185     input_y(i) = 5*sin(2*pi*(1)*input_x(i));
186 end
187 a7 = plot(input_x, input_y);
188
189 legend([a4; a5; a6; a7], [M1; M2; M3; M4]);
190 hold off;
191 title('Test 7: Sine, period 1000us');
192 xlabel('Time/ms');
193 ylabel('Voltage/V');
194
195 %%%%%%%%%%TEST 8: Y = Square, period 100us %%%%%%%%%%%%%%%%%
196 ODE = @(x,y) ((5*square(2*pi*(10)*x)-y)*10);
197 figure; hold on;
198 [out_x4, out_y4] = RK2(ODE, 0.001, 0.5, 0, 5, 0);
199 a4 = plot(out_x4, out_y4);
200
201 [out_x5, out_y5] = RK2(ODE, 0.001, 0.5, 0, 5, 0.5);
202 a5 = plot(out_x5, out_y5);
203
204 [out_x6, out_y6] = RK2(ODE, 0.001, 0.5, 0, 5, 0.25);
205 a6 = plot(out_x6, out_y6);
```

```
206
207 for i=1:2001
208     input_x(i) = 0.00025*(i-1);
209     input_y(i) = 5*square(2*pi*(10)*input_x(i));
210 end
211 a7 = plot(input_x, input_y);
212
213 legend([a4; a5; a6; a7], [M1; M2; M3; M4]);
214 hold off;
215 title('Test 8: Square, period 100us');
216 xlabel('Time/ms');
217 ylabel('Voltage/V');
218
219 %%%%%%%%%%TEST 9: Y = Square, period 10us %%%%%%%%%%%%%%%%%%
220 ODE = @(x,y) ((5*square(2*pi*(10^2)*x)-y)*10);
221 figure; hold on;
222 [out_x4, out_y4] = RK2(ODE, 0.0002, 0.2, 0, 5, 0);
223 a4 = plot(out_x4, out_y4);
224
225 [out_x5, out_y5] = RK2(ODE, 0.0002, 0.2, 0, 5, 0.5);
226 a5 = plot(out_x5, out_y5);
227
228 [out_x6, out_y6] = RK2(ODE, 0.0002, 0.2, 0, 5, 0.25);
229 a6 = plot(out_x6, out_y6);
230
231 for i=1:2001
232     input_x(i) = 0.0001*(i-1);
233     input_y(i) = 5*square(2*pi*(10^2)*input_x(i));
234 end
235 a7 = plot(input_x, input_y);
236
237 legend([a4; a5; a6; a7], [M1; M2; M3; M4]);
238 hold off;
239 title('Test 9: Square, period 10us');
240 xlabel('Time/ms');
241 ylabel('Voltage/V');
242
243 %%%%%%%%%%TEST 10: Y = Square, period 500us %%%%%%%%%%%%%%%%%%
244 ODE = @(x,y) ((5*square(2*pi*5*x)-y)*10);
245 figure; hold on;
246 [out_x4, out_y4] = RK2(ODE, 0.003, 0.6, 0, 5, 0);
247 a4 = plot(out_x4, out_y4);
```

```
248
249 [out_x5, out_y5] = RK2(ODE, 0.003, 0.6, 0, 5, 0.5);
250 a5 = plot(out_x5, out_y5);
251
252 [out_x6, out_y6] = RK2(ODE, 0.003, 0.6, 0, 5, 0.25);
253 a6 = plot(out_x6, out_y6);
254
255 for i=1:2001
256     input_x(i) = 0.0003*(i-1);
257     input_y(i) = 5*square(2*pi*5*input_x(i));
258 end
259 a7 = plot(input_x, input_y);
260
261 legend([a4; a5; a6; a7], [M1; M2; M3; M4]);
262 hold off;
263 title('Test 10: Square, period 500us');
264 xlabel('Time/ms');
265 ylabel('Voltage/V');
266
267 %%%%%%%%%%TEST 11: Y = Square, period 1000us %%%%%%%%%%%%%%%%%
268 ODE = @(x,y) ((5*square(2*pi*(1)*x)-y)*10);
269 figure; hold on;
270 [out_x4, out_y4] = RK2(ODE, 0.03, 3, 0, 5, 0);
271 a4 = plot(out_x4, out_y4);
272
273 [out_x5, out_y5] = RK2(ODE, 0.03, 3, 0, 5, 0.5);
274 a5 = plot(out_x5, out_y5);
275
276 [out_x6, out_y6] = RK2(ODE, 0.03, 3, 0, 5, 0.25);
277 a6 = plot(out_x6, out_y6);
278
279 for i=1:2001
280     input_x(i) = 0.0015*(i-1);
281     input_y(i) = 5*square(2*pi*(1)*input_x(i));
282 end
283 a7 = plot(input_x, input_y);
284
285 legend([a4; a5; a6; a7], [M1; M2; M3; M4]);
286 hold off;
287 title('Test 11: Square, period 1000us');
288 xlabel('Time/ms');
289 ylabel('Voltage/V');
```

```
290
291 %%%%%%%%%%TEST 12: Y = Sawtooth, period 100us
    %%%%%%%%%%%%%%%%%%
292 ODE = @(x,y) ((5*sawtooth(2*pi*(10)*x)-y)*10);
293 figure; hold on;
294 [out_x4, out_y4] = RK2(ODE, 0.001, 0.5, 0, 5, 0);
295 a4 = plot(out_x4, out_y4);
296
297 [out_x5, out_y5] = RK2(ODE, 0.001, 0.5, 0, 5, 0.5);
298 a5 = plot(out_x5, out_y5);
299
300 [out_x6, out_y6] = RK2(ODE, 0.001, 0.5, 0, 5, 0.25);
301 a6 = plot(out_x6, out_y6);
302
303 for i=1:2001
304     input_x(i) = 0.00025*(i-1);
305     input_y(i) = 5*sawtooth(2*pi*(10)*input_x(i));
306 end
307 a7 = plot(input_x, input_y);
308
309 legend([a4; a5; a6; a7], [M1; M2; M3; M4]);
310 hold off;
311 title('Test 12: Sawtooth, period 100us');
312 xlabel('Time/ms');
313 ylabel('Voltage/V');
314
315 %%%%%%%%%%TEST 13: Y = Sawtooth, period 10us %%%%%%%%%%%%%%%%%
316 ODE = @(x,y) ((5*sawtooth(2*pi*(10^2)*x)-y)*10);
317 figure; hold on;
318 [out_x4, out_y4] = RK2(ODE, 0.0002, 0.2, 0, 5, 0);
319 a4 = plot(out_x4, out_y4);
320
321 [out_x5, out_y5] = RK2(ODE, 0.0002, 0.2, 0, 5, 0.5);
322 a5 = plot(out_x5, out_y5);
323
324 [out_x6, out_y6] = RK2(ODE, 0.0002, 0.2, 0, 5, 0.25);
325 a6 = plot(out_x6, out_y6);
326
327 for i=1:2001
328     input_x(i) = 0.0001*(i-1);
329     input_y(i) = 5*sawtooth(2*pi*(10^2)*input_x(i));
330 end
```

```
331 a7 = plot(input_x, input_y);
332
333 legend([a4; a5; a6; a7], [M1; M2; M3; M4]);
334 hold off;
335 title('Test 13: Sawtooth, period 10us');
336 xlabel('Time/ms');
337 ylabel('Voltage/V');
338
339 %%%%%%%%%%TEST 14: Y = Sawtooth, period 500us
      %%%%%%%%%%%%%%%%%%
340 ODE = @(x,y) ((5*sawtooth(2*pi*5*x)-y)*10);
341 figure; hold on;
342 [out_x4, out_y4] = RK2(ODE, 0.003, 0.6, 0, 5, 0);
343 a4 = plot(out_x4, out_y4);
344
345 [out_x5, out_y5] = RK2(ODE, 0.003, 0.6, 0, 5, 0.5);
346 a5 = plot(out_x5, out_y5);
347
348 [out_x6, out_y6] = RK2(ODE, 0.003, 0.6, 0, 5, 0.25);
349 a6 = plot(out_x6, out_y6);
350
351 for i=1:2001
352     input_x(i) = 0.0003*(i-1);
353     input_y(i) = 5*sawtooth(2*pi*5*input_x(i));
354 end
355 a7 = plot(input_x, input_y);
356
357 legend([a4; a5; a6; a7], [M1; M2; M3; M4]);
358 hold off;
359 title('Test 14: Sawtooth, period 500us');
360 xlabel('Time/ms');
361 ylabel('Voltage/V');
362
363 %%%%%%%%%%TEST 15: Y = Sawtooth, period 1000us
      %%%%%%%%%%%%%%%%%%
364 ODE = @(x,y) ((5*sawtooth(2*pi*(1)*x)-y)*10);
365 figure; hold on;
366 [out_x4, out_y4] = RK2(ODE, 0.03, 3, 0, 5, 0);
367 a4 = plot(out_x4, out_y4);
368
369 [out_x5, out_y5] = RK2(ODE, 0.03, 3, 0, 5, 0.5);
370 a5 = plot(out_x5, out_y5);
```

```
371
372 [out_x6, out_y6] = RK2(ODE, 0.03, 3, 0, 5, 0.25);
373 a6 = plot(out_x6, out_y6);
374
375 for i=1:2001
376     input_x(i) = 0.0015*(i-1);
377     input_y(i) = 5*sawtooth(2*pi*(1)*input_x(i));
378 end
379 a7 = plot(input_x, input_y);
380
381 legend([a4; a5; a6; a7], [M1; M2; M3; M4]);
382 hold off;
383 title('Test 15: Sawtooth, period 1000us');
384 xlabel('Time/ms');
385 ylabel('Voltage/V');
```

Error analysis script for RL circuit with cosine input (error_script.m)

```
% Error analysis for the RC circuit with a cosine input
% the numerical solution is compared to the mathematically computed exact solution
% the error is then plotted for each method.

% THE EXACT SOLUTION OF THE ODE IS GIVEN BY:
% y = 5(2pi(2pi(e^(-10000x))+sin(20000pix))+cos(20000pix))/(1+4pi^2)

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% 1. CREATING THE EXACT EQUATION INTO THE MATRIX
% this exact value ranges between x = 0 and 7x10^-4 seconds
% and has 1000 steps. This mean that each step is 0.7us.
exact_x = zeros(1,1000);
exact_y = zeros(1,1000);
for i = 1:10000
    exact_x(i)= 0.00000007*i;
    exact_y(i) = (5*(2*pi*(2*pi*exp(-10000*exact_x(i))+sin(20000*pi*exact_x(i)))+cos(20000*pi*exact_x(i))))/(1+(4*pi*pi));
end
Solution = @(x) (5*(2*pi*(2*pi*exp(-10000*x)+sin(20000*pi*x))+cos(20000*pi*x)))/(1+(4*pi*pi));

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%2. Analytical Solving
ODE =@(x,y) ((5*cos(2*pi*(10^4)*x)-y)*10000);

%%%%%% SETTING UP THE LEGEND %%%%%%
Aa = "Midpoint Method";
Bb = "Heun Method";
Cc = "Our Custom Quarter Method";
Ee = "Exact Solution";

%%%%%% SETTING THE STEP SIZES %%%%%%
steps = [2, 4, 5, 8, 10, 16, 20, 25, 40, 50, 80, 100, 125, 200, 250, 400, 500, 625, 1000, 1250, 2000, 2500, 5000];
stepsize = 10000./steps.*0.00000007;

```

```
%%%%%% Initialising Array Spaces %%%%%%
A_x=zeros(1,23);
A_yRMS=zeros(1,23);
A_y=zeros(1,23);
B_x=zeros(1,23);
B_yRMS=zeros(1,23);
B_y=zeros(1,23);
C_x=zeros(1,23);
C_yRMS=zeros(1,23);
C_y=zeros(1,23);

%%%%%% DO THE LOOP THROUGH ALL FACTORS OF 10000 %%%%%%
for i = 1:23
    [out_x1, out_y1] = RK2(ODE, stepsize(i), 0.0007, 0, 5, 0);
    [out_x2, out_y2] = RK2(ODE, stepsize(i), 0.0007, 0, 5, 0.5);
    [out_x3, out_y3] = RK2(ODE, stepsize(i), 0.0007, 0, 5, 0.25)
        ;
    figure; hold on;
    title(strcat("Comparison of Graphs - ", num2str(steps(i))," 
        Steps"));
    A = plot(out_x1, out_y1);
    B = plot(out_x2, out_y2);
    C = plot(out_x3, out_y3);
    E = plot(exact_x, exact_y);
    legend([A; B; C; E], [Aa; Bb; Cc; Ee]);
    ylabel("V_{out} / V"); xlabel("time / s");
    hold off;
    A_error_y = zeros(1, steps(i));
    B_error_y = zeros(1, steps(i));
    C_error_y = zeros(1, steps(i));
    for j = 1:steps(i)
        A_error_y(j) = abs(exact_y((10000/steps(i))*j) - out_y1
            (j));
        B_error_y(j) = abs(exact_y((10000/steps(i))*j)- out_y2(
            j));
        C_error_y(j) = abs(exact_y((10000/steps(i))*j) - out_y3
            (j));
    end

    figure; hold on;

    title(strcat("Error between numerical and analytical graphs
```

```
        - ", num2str(steps(i)), " Steps"));
73
74      ylabel("V_o_u_t / V");
75      xlabel("time / s");
76      A = plot(out_x1, A_error_y);
77      B = plot(out_x2, B_error_y);
78      C = plot(out_x3, C_error_y);
79      legend([A; B; C], [Aa; Bb; Cc]);
80      hold off;
81      A_x(i) = log10(stepsize(i));
82      A_yRMS(i) = log10(abs(rms(A_error_y)));
83      A_y(i) = log10(mean(A_error_y));
84      B_x(i) = log10(stepsize(i));
85      B_yRMS(i) = log10(abs(rms(B_error_y)));
86      B_y(i) = log10(mean(B_error_y));
87      C_x(i) = log10(stepsize(i));
88      C_yRMS(i) = log10(abs(rms(C_error_y)));
89      C_y(i) = log10(mean(C_error_y));
90      clear A_error_y;
91      clear B_error_y;
92      clear C_error_y;
93  end
94
95  figure; hold on;
96  title("log(average error) against log(step size)");
97  ylabel("log(average error)");
98  xlabel("log(step size)");
99  A = plot(A_x, A_y);
100 B = plot(B_x, B_y);
101 C = plot(C_x, C_y);
102 legend([A; B; C], [Aa; Bb; Cc]);
103 hold off;
104
105 figure; hold on;
106 title("log(rms(error)) against log(step size)");
107 ylabel("log(rms(error))");
108 xlabel("log(step size)");
109 A = plot(A_x, A_yRMS);
110 B = plot(B_x, B_yRMS);
111 C = plot(C_x, C_yRMS);
112 legend([A; B; C], [Aa; Bb; Cc]);
113 hold off;
```

1.2 RLC Circuit

RK4 code (RK4.m)

```
function [x_values, y_values, z_values] = RK4(ODE_y, ODE_z, h,
    final_val, xi, yi, zi)
    % ODE Coupled second order ODE solver
    % @param ODE_y The ODE to be solved in the form y' = f(x, y,
        z)
    % @param ODE_z The ODE to be solved in the form z' = g(x, y,
        z)
    % @param step_size aka h The distance on the x-axis between
       two
    % consecutive steps
    % @param final_val The x-value up to which the ODE is
       evaluated
    % @param yi The initial y-value
    % @param zi The initial z-value


    % Calculate the number of steps
    N = round(final_val / h);

    % Initialise output arrays
    x_values = zeros(1, N);
    y_values = zeros(1, N);
    z_values = zeros(1, N);
    % xi is defined to be 0
    x_values(1) = xi;
    y_values(2) = yi;
    z_values(1) = zi;


    for i = 1 : N - 1
        % Calculate the coefficients
        k0 = h * feval(ODE_y, x_values(i), y_values(i), z_values
            (1));
        l0 = h * feval(ODE_z, x_values(i), y_values(i), z_values
            (1));
        k1 = h * feval(ODE_y, x_values(i) + (0.5 * h), y_values(
            i) + (0.5 * k0), z_values(1) + (0.5 * l0));
        l1 = h * feval(ODE_z, x_values(i) + (0.5 * h), y_values(
            i) + (0.5 * k0), z_values(1) + (0.5 * l0));
```

```
31          k2 = h * feval(ODE_y, x_values(i) + (0.5 * h), y_values(
                i) + (0.5 * k1), z_values(1) + (0.5 * l1));
32          l2 = h * feval(ODE_z, x_values(i) + (0.5 * h), y_values(
                i) + (0.5 * k1), z_values(1) + (0.5 * l1));
33          k3 = h * feval(ODE_y, x_values(i) + h, y_values(i) + k2,
                 z_values(i) + l2);
34          l3 = h * feval(ODE_z, x_values(i) + h, y_values(i) + k2,
                 z_values(i) + l2);
35
36          x_values(i + 1) = x_values(i) + h;
37          y_values(i + 1) = y_values(i) + ((1.0 / 6.0) * (k0 + 2*
                k1 + 2*k2 + k3));
38          z_values(i + 1) = z_values(i) + ((1.0 / 6.0) * (l0 + 2*
                l1 + 2*l2 + l3));
39      end
40  end
```

Test script for RLC Circuit (RLC_script.m)

```
close all;

% Graph labels
IN = "V_{in}";
OUT = "V_{out}";

% Circuit parameters
R = 250;
C = 3.5 * 10^-6;
L = 0.6;
Q0 = 500 * 10^-9;
Q_dash_0 = 0;

%%%%%%%%%%TEST 1: Step signal %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% The input to the system as a function of x
input = @(x) 5;

ODE_y = @(x, y, z) z;
ODE_z = @(x, y, z) ((input(x) - R*z - (1/C)*y) / L);

[out_x, out_y, out_z] = RK4(ODE_y, ODE_z, 0.00003, 0.05, 0, Q0,
    Q_dash_0);

% Calculate V_out as R*z
V_out = R*out_z;

figure; hold on;
input_vals = arrayfun(input, out_x);
p1 = plot(out_x, input_vals);
p2 = plot(out_x, V_out);

title('Test 1: Step signal')
legend([p1; p2], [IN, OUT]);
xlabel('Time/s');
ylabel('Voltage/V');
ylim([-2, 5.5]);

%%%%%%%%%%TEST 2: Y = Impulse and decay signal 1 %%%%%%%%%%%%%%%%%%%%%%
input = @(x) 5*exp(-x^2 / 0.000003);
```

```
39 ODE_z = @(x, y, z) ((input(x) - R*z - (1/C)*y) / L);
40
41 [out_x, out_y, out_z] = RK4(ODE_y, ODE_z, 0.000003, 0.05, 0, Q0,
       Q_dash_0);
42
43 % Calculate V_out as R*z
44 V_out = R*out_z;
45
46 figure; hold on;
47 input_vals = arrayfun(input, out_x);
48 p1 = plot(out_x, input_vals);
49 p2 = plot(out_x, V_out);
50
51 title('Test 2: Impulse decay signal')
52 legend([p1; p2], [IN, OUT]);
53 xlabel('Time/s');
54 ylabel('Voltage/V');
55 ylim([-2, 5.5]);
56
57 %%%%%%%%%%TEST 3: Square wave with frequency f = 109 Hz
       %%%%%%%%%%%%%%%%%
58 input = @(x) square(2*pi*(109)*x);
59 ODE_z = @(x, y, z) ((input(x) - R*z - (1/C)*y) / L);
60
61 [out_x, out_y, out_z] = RK4(ODE_y, ODE_z, 0.00003, 0.07, 0, Q0,
       Q_dash_0);
62
63 % Calculate V_out as R*z
64 V_out = R*out_z;
65
66 figure; hold on;
67 input_vals = arrayfun(input, out_x);
68 p1 = plot(out_x, input_vals);
69 p2 = plot(out_x, V_out);
70
71 title('Test 3: Square wave, frequency 109 Hz')
72 legend([p1; p2], [IN, OUT]);
73 xlabel('Time/s');
74 ylabel('Voltage/V');
75
76 %%%%%%%%%%TEST 4: Square wave with frequency f = 5 Hz
       %%%%%%%%%%%%%%%%%%%
```

```
77 input = @(x) square(2*pi*(5)*x);
78 ODE_z = @(x, y, z) ((input(x) - R*z - (1/C)*y) / L);
79
80 [out_x, out_y, out_z] = RK4(ODE_y, ODE_z, 0.00003, 0.3, 0, Q0,
       Q_dash_0);
81
82 % Calculate V_out as R*z
83 V_out = R*out_z;
84
85 figure; hold on;
86 input_vals = arrayfun(input, out_x);
87 p1 = plot(out_x, input_vals);
88 p2 = plot(out_x, V_out);
89
90 title('Test 4: Square wave, frequency 5 Hz')
91 legend([p1; p2], [IN, OUT]);
92 xlabel('Time/s');
93 ylabel('Voltage/V');
94 ylim([-1.2, 1.2]);
95
96 %%%%%%%%%%TEST 5: Square wave with frequency f = 500 Hz
       %%%%%%%%%%%%%%%%%%
97 input = @(x) square(2*pi*(500)*x);
98 ODE_z = @(x, y, z) ((input(x) - R*z - (1/C)*y) / L);
99
100 [out_x, out_y, out_z] = RK4(ODE_y, ODE_z, 0.000003, 0.04, 0, Q0,
        Q_dash_0);
101
102 % Calculate V_out as R*z
103 V_out = R*out_z;
104
105 figure; hold on;
106 input_vals = arrayfun(input, out_x);
107 p1 = plot(out_x, input_vals);
108 p2 = plot(out_x, V_out);
109
110 title('Test 5: Square wave, frequency 500 Hz')
111 legend([p1; p2], [IN, OUT]);
112 xlabel('Time/s');
113 ylabel('Voltage/V');
114 ylim([-1.2, 1.2]);
115
```

```
116 %%%%%%%%%%TEST 6: Sine wave with frequency f = 109 Hz %%%%%%%%%%%%%%%%%%
117 input = @(x) sin(2*pi*(109)*x);
118 ODE_z = @(x, y, z) ((input(x) - R*z - (1/C)*y) / L);
119
120 [out_x, out_y, out_z] = RK4(ODE_y, ODE_z, 0.00003, 0.07, 0, Q0, Q_dash_0);
121
122 % Calculate V_out as R*z
123 V_out = R*out_z;
124
125 figure; hold on;
126 input_vals = arrayfun(input, out_x);
127 p1 = plot(out_x, input_vals);
128 p2 = plot(out_x, V_out);
129
130 title('Test 6: Sine wave, frequency 109 Hz')
131 legend([p1; p2], [IN, OUT]);
132 xlabel('Time/ms');
133 ylabel('Voltage/V');
134 ylim([-1.2, 1.2]);
135
136 %%%%%%%%%%TEST 7: Sine wave with frequency f = 5 Hz %%%%%%%%%%%%%%%%%%
137 input = @(x) sin(2*pi*(5)*x);
138 ODE_z = @(x, y, z) ((input(x) - R*z - (1/C)*y) / L);
139
140 [out_x, out_y, out_z] = RK4(ODE_y, ODE_z, 0.00003, 0.3, 0, Q0, Q_dash_0);
141
142 % Calculate V_out as R*z
143 V_out = R*out_z;
144
145 figure; hold on;
146 input_vals = arrayfun(input, out_x);
147 p1 = plot(out_x, input_vals);
148 p2 = plot(out_x, V_out);
149
150 title('Test 7: Sine wave, frequency 5 Hz')
151 legend([p1; p2], [IN, OUT]);
152 xlabel('Time/s');
153 ylabel('Voltage/V');
```

```
ylim([-1.2, 1.2]);

%%%%%%%%%%TEST 8: Sine wave with frequency f = 500 Hz
    %%%%%%%%%%%%%%%%%
input = @(x) sin(2*pi*(500)*x);
ODE_z = @(x, y, z) ((input(x) - R*z - (1/C)*y) / L);

[out_x, out_y, out_z] = RK4(ODE_y, ODE_z, 0.00003, 0.04, 0, Q0,
    Q_dash_0);

% Calculate V_out as R*z
V_out = R*out_z;

figure; hold on;
input_vals = arrayfun(input, out_x);
p1 = plot(out_x, input_vals);
p2 = plot(out_x, V_out);

title('Test 8: Sine wave, frequency 500 Hz')
legend([p1; p2], [IN, OUT]);
xlabel('Time/s');
ylabel('Voltage/V');
ylim([-1.2, 1.2]);

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%% -- New circuit 1
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%% critically
    damped
% Circuit parameters
R = 2000;
C = 300 * 10^-9;
L = 0.3;
%Q0 = 500 * 10^-9;
Q0 = 0;
Q_dash_0 = 0;


%%%%%%%%%%TEST 1: Step signal
    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% The input to the system as a function of x
input = @(x) 5;

ODE_y = @(x, y, z) z;
```

```
192 ODE_z = @(x, y, z) ((input(x) - R*z - (1/C)*y) / L);
193
194 [out_x, out_y, out_z] = RK4(ODE_y, ODE_z, 0.00003, 0.02, 0, Q0,
        Q_dash_0);
195
196 % Calculate V_out as R*z
197 V_out = R*out_z;
198
199 figure; hold on;
200 input_vals = arrayfun(input, out_x);
201 p1 = plot(out_x, input_vals);
202 p2 = plot(out_x, V_out);
203
204 title('Test 1: Step signal (CD)')
205 legend([p1; p2], [IN, OUT]);
206 xlabel('Time/s');
207 ylabel('Voltage/V');
208 ylim([-1, 5.5]);
209
210 %%%%%%%%%%TEST 2: Impulse and decay signal
        %%%%%%%%%%%%%%%%%%%%%%%%
211 input = @(x) 5*exp(-x^2 / 0.000003);
212 ODE_z = @(x, y, z) ((input(x) - R*z - (1/C)*y) / L);
213
214 [out_x, out_y, out_z] = RK4(ODE_y, ODE_z, 0.000003, 0.05, 0, Q0,
        Q_dash_0);
215
216 % Calculate V_out as R*z
217 V_out = R*out_z;
218
219 figure; hold on;
220 input_vals = arrayfun(input, out_x);
221 p1 = plot(out_x, input_vals);
222 p2 = plot(out_x, V_out);
223
224 title('Test 2: Impulse decay signal (CD)')
225 legend([p1; p2], [IN, OUT]);
226 xlabel('Time/s');
227 ylabel('Voltage/V');
228 ylim([-2, 5.5]);
229
230 %%%%%%%%%%TEST 3: Square wave at resonance frequency (350.5 Hz)
```

```
    %%%%%%%%%%%%%%%%%%%%
input = @(x) square(2*pi*(350.5)*x);
ODE_z = @(x, y, z) ((input(x) - R*z - (1/C)*y) / L);

[out_x, out_y, out_z] = RK4(ODE_y, ODE_z, 0.000003, 0.03, 0, Q0, Q_dash_0);

% Calculate V_out as R*z
V_out = R*out_z;

figure; hold on;
input_vals = arrayfun(input, out_x);
p1 = plot(out_x, input_vals);
p2 = plot(out_x, V_out);

title('Test 3: Square wave, resonance frequency (350.5 Hz)(OD)')
legend([p1; p2], [IN, OUT]);
xlabel('Time/s');
ylabel('Voltage/V');

%%%%%%%%%%TEST 4: Sine wave with frequency f = 5 Hz
    %%%%%%%%%%%%%%%%%%%%
input = @(x) sin(2*pi*(5)*x);
ODE_z = @(x, y, z) ((input(x) - R*z - (1/C)*y) / L);

[out_x, out_y, out_z] = RK4(ODE_y, ODE_z, 0.00003, 0.3, 0, Q0, Q_dash_0);

% Calculate V_out as R*z
V_out = R*out_z;

figure; hold on;
input_vals = arrayfun(input, out_x);
p1 = plot(out_x, input_vals);
p2 = plot(out_x, V_out);

title('Test 4: Sine wave, frequency 5 Hz (CD)')
legend([p1; p2], [IN, OUT]);
xlabel('Time/s');
ylabel('Voltage/V');
ylim([-1.2, 1.2]);

```

```
269 %%%%%%%%%%TEST 5: Sine wave at resonance frequency (350.5 Hz)
    %%%%%%%%%%%%%%%%%%%%
270 input = @(x) sin(2*pi*(350.5)*x);
271 ODE_z = @(x, y, z) ((input(x) - R*z - (1/C)*y) / L);
272
273 [out_x, out_y, out_z] = RK4(ODE_y, ODE_z, 0.000003, 0.03, 0, Q0,
        Q_dash_0);
274
275 % Calculate V_out as R*z
276 V_out = R*out_z;
277
278 figure; hold on;
279 input_vals = arrayfun(input, out_x);
280 p1 = plot(out_x, input_vals);
281 p2 = plot(out_x, V_out);
282
283 title('Test 5: Sine wave, resonance frequency (350.5 Hz)(OD)')
284 legend([p1; p2], [IN, OUT]);
285 xlabel('Time/s');
286 ylabel('Voltage/V');
287 ylim([-1.2, 1.2]);
288
289 %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%% -- New circuit
    2--
290 %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%% overdamped
291 % Circuit parameters
292 R = 2000;
293 C = 3 * 10^-6;
294 L = 0.8;
295 %Q0 = 500 * 10^-9;
296 Q0 = 0;
297 Q_dash_0 = 0;
298
299 %%%%%%%%%%TEST 1: Step signal
    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
300 % The input to the system as a function of x
301 input = @(x) 5;
302
303 ODE_y = @(x, y, z) z;
304 ODE_z = @(x, y, z) ((input(x) - R*z - (1/C)*y) / L);
305
306 [out_x, out_y, out_z] = RK4(ODE_y, ODE_z, 0.00003, 0.05, 0, Q0,
```

```
    Q_dash_0);
307
308 % Calculate V_out as R*z
309 V_out = R*out_z;
310
311 figure; hold on;
312 input_vals = arrayfun(input, out_x);
313 p1 = plot(out_x, input_vals);
314 p2 = plot(out_x, V_out);
315
316 title('Test 1: Step signal (OD)')
317 legend([p1; p2], [IN, OUT]);
318 xlabel('Time/s');
319 ylabel('Voltage/V');
320 ylim([-1, 5.5]);
321
322 %%%%%%%%%%TEST 2: Impulse and decay signal
    %%%%%%%%%%%%%%%%%%%%%%%%%
323 input = @(x) 5*exp(-x^2 / 0.000003);
324 ODE_z = @(x, y, z) ((input(x) - R*z - (1/C)*y) / L);
325
326 [out_x, out_y, out_z] = RK4(ODE_y, ODE_z, 0.000003, 0.05, 0, Q0,
      Q_dash_0);
327
328 % Calculate V_out as R*z
329 V_out = R*out_z;
330
331 figure; hold on;
332 input_vals = arrayfun(input, out_x);
333 p1 = plot(out_x, input_vals);
334 p2 = plot(out_x, V_out);
335
336 title('Test 2: Impulse decay signal (OD)')
337 legend([p1; p2], [IN, OUT]);
338 xlabel('Time/s');
339 ylabel('Voltage/V');
340 ylim([-2, 5.5]);
341
342 %%%%%%%%%%TEST 3: Square wave at resonance frequency (102.7 Hz)
    %%%%%%%%%%%%%%%%%%%%%
343 input = @(x) square(2*pi*(102.7)*x);
344 ODE_z = @(x, y, z) ((input(x) - R*z - (1/C)*y) / L);
```

```
345
346 [out_x, out_y, out_z] = RK4(ODE_y, ODE_z, 0.000003, 0.03, 0, Q0,
        Q_dash_0);
347
348 % Calculate V_out as R*z
349 V_out = R*out_z;
350
351 figure; hold on;
352 input_vals = arrayfun(input, out_x);
353 p1 = plot(out_x, input_vals);
354 p2 = plot(out_x, V_out);
355
356 title('Test 3: Square wave, resonance frequency (102.7 Hz)(OD)')
357 legend([p1; p2], [IN, OUT]);
358 xlabel('Time/s');
359 ylabel('Voltage/V');
360
361 %%%%%%%%%%TEST 4: Sine wave with frequency f = 5 Hz
        %%%%%%%%%%%%%%%%%%%%
362 input = @(x) sin(2*pi*(5)*x);
363 ODE_z = @(x, y, z) ((input(x) - R*z - (1/C)*y) / L);
364
365 [out_x, out_y, out_z] = RK4(ODE_y, ODE_z, 0.00003, 0.3, 0, Q0,
        Q_dash_0);
366
367 % Calculate V_out as R*z
368 V_out = R*out_z;
369
370 figure; hold on;
371 input_vals = arrayfun(input, out_x);
372 p1 = plot(out_x, input_vals);
373 p2 = plot(out_x, V_out);
374
375 title('Test 4: Sine wave, frequency 5 Hz (OD)')
376 legend([p1; p2], [IN, OUT]);
377 xlabel('Time/s');
378 ylabel('Voltage/V');
379 ylim([-1.2, 1.2]);
380
381 %%%%%%%%%%TEST 5: Sine wave at resonance frequency (102.7 Hz)
        %%%%%%%%%%%%%%%%%%%%
382 input = @(x) sin(2*pi*(102.7)*x);
```

```
383 ODE_z = @(x, y, z) ((input(x) - R*z - (1/C)*y) / L);
384
385 [out_x, out_y, out_z] = RK4(ODE_y, ODE_z, 0.000003, 0.03, 0, Q0,
        Q_dash_0);
386
387 % Calculate V_out as R*z
388 V_out = R*out_z;
389
390 figure; hold on;
391 input_vals = arrayfun(input, out_x);
392 p1 = plot(out_x, input_vals);
393 p2 = plot(out_x, V_out);
394
395 title('Test 5: Sine wave, resonance frequency (102.7 Hz)(OD)')
396 legend([p1; p2], [IN, OUT]);
397 xlabel('Time/s');
398 ylabel('Voltage/V');
399 ylim([-1.2, 1.2]);
```

1.3 Relaxation

Relaxation with boundary conditions code (relaxation.m)

```
% The test script for relaxation

% Set the boundary function b1, b2, b3, b4
b1 = @(y) y;
b2 = @(y) y;
b3 = @(x) x;
b4 = @(x) x;

grid_size = 50;
x = 0:1/grid_size:1-(1/grid_size);
x2 = 0:1/(grid_size*4):1-(1/(grid_size*4));
required_accuracy = 0.000001;

figure;
[grid_out, count] = relax_func(grid_size, b1, b2, b3, b4,
    required_accuracy);
[X, Y] = meshgrid(x);
meshc(X, Y, grid_out);

figure; hold on;
[grid_out, count] = relax_func(grid_size * 4, b1, b2, b3, b4,
    required_accuracy);
[X, Y] = meshgrid(x2);
s2 = surf(X, Y, grid_out);
s2.EdgeColor = 'none';
colorbar;

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%% TEST 1
    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
b1 = @(y) 1;
b2 = @(y) 1;
b3 = @(x) 0;
b4 = @(x) 0;

figure;
[grid_out, count] = relax_func(grid_size, b1, b2, b3, b4,
    required_accuracy);
[X, Y] = meshgrid(x);
meshc(X, Y, grid_out);
```

```
36
37 figure; hold on;
38 [grid_out, count] = relax_func(grid_size * 4, b1, b2, b3, b4,
       required_accuracy);
39 [X, Y] = meshgrid(x2);
40 s2 = surf(X, Y, grid_out);
41 s2.EdgeColor = 'none';
42 colorbar;
43
44 %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%% TEST 2
       %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
45 b1 = @(y) 0;
46 b2 = @(y) 1;
47 b3 = @(x) 0;
48 b4 = @(x) 1;
49
50 figure;
51 [grid_out, count] = relax_func(grid_size, b1, b2, b3, b4,
       required_accuracy);
52 [X, Y] = meshgrid(x);
53 meshc(X, Y, grid_out);
54
55 figure; hold on;
56 [grid_out, count] = relax_func(grid_size * 4, b1, b2, b3, b4,
       required_accuracy);
57 [X, Y] = meshgrid(x2);
58 s2 = surf(X, Y, grid_out);
59 s2.EdgeColor = 'none';
60 colorbar;
61
62 %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%% TEST 3
       %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
63 b1 = @(y) 0;
64 b2 = @(y) 1;
65 b3 = @(x) 0;
66 b4 = @(x) 0;
67
68 figure;
69 [grid_out, count] = relax_func(grid_size, b1, b2, b3, b4,
       required_accuracy);
70 [X, Y] = meshgrid(x);
71 meshc(X, Y, grid_out);
```

```
72
73 figure; hold on;
74 [grid_out, count] = relax_func(grid_size * 4, b1, b2, b3, b4,
       required_accuracy);
75 [X, Y] = meshgrid(x2);
76 s2 = surf(X, Y, grid_out);
77 s2.EdgeColor = 'none';
78 colorbar;
79
80 %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%% TEST 4
       %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
81 b1 = @(y) y^2;
82 b2 = @(y) 0.5*y;
83 b3 = @(x) x^2;
84 b4 = @(x) 0.5*x;
85
86 figure;
87 [grid_out, count] = relax_func(grid_size, b1, b2, b3, b4,
       required_accuracy);
88 [X, Y] = meshgrid(x);
89 meshc(X, Y, grid_out);
90
91 figure; hold on;
92 [grid_out, count] = relax_func(grid_size * 4, b1, b2, b3, b4,
       required_accuracy);
93 [X, Y] = meshgrid(x2);
94 s2 = surf(X, Y, grid_out);
95 s2.EdgeColor = 'none';
96 colorbar;
97
98 %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%% TEST 5
       %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
99 b1 = @(y) 2*cos(4*pi*y);
100 b2 = @(y) 2*cos(2*pi*y);
101 b3 = @(x) 2*cos(4*pi*x);
102 b4 = @(x) 2*cos(2*pi*x);
103
104 figure;
105 [grid_out, count] = relax_func(grid_size, b1, b2, b3, b4,
       required_accuracy);
106 [X, Y] = meshgrid(x);
107 meshc(X, Y, grid_out);
```

```
108
109 figure; hold on;
110 [grid_out, count] = relax_func(grid_size * 4, b1, b2, b3, b4,
      required_accuracy);
111 [X, Y] = meshgrid(x2);
112 s2 = surf(X, Y, grid_out);
113 s2.EdgeColor = 'none';
114 colorbar;
115
116 %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%% TEST 6
      %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
117 b1 = @(y) 2*sin(2*pi*y);
118 b2 = @(y) 2*sin(2*pi*y);
119 b3 = @(x) 2*sin(2*pi*x);
120 b4 = @(x) 2*sin(2*pi*x);
121
122 figure;
123 [grid_out, count] = relax_func(grid_size, b1, b2, b3, b4,
      required_accuracy);
124 [X, Y] = meshgrid(x);
125 meshc(X, Y, grid_out);
126
127 figure; hold on;
128 [grid_out, count] = relax_func(grid_size * 4, b1, b2, b3, b4,
      required_accuracy);
129 [X, Y] = meshgrid(x2);
130 s2 = surf(X, Y, grid_out);
131 s2.EdgeColor = 'none';
132 colorbar;
133
134 %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%% TEST 7
      %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
135 b1 = @(y) y^2;
136 b2 = @(y) y^2;
137 b3 = @(x) x^2;
138 b4 = @(x) x^2;
139
140 figure;
141 [grid_out, count] = relax_func(grid_size, b1, b2, b3, b4,
      required_accuracy);
142 [X, Y] = meshgrid(x);
143 meshc(X, Y, grid_out);
```

```
144
145 figure; hold on;
146 [grid_out, count] = relax_func(grid_size * 4, b1, b2, b3, b4,
    required_accuracy);
147 [X, Y] = meshgrid(x2);
148 s2 = surf(X, Y, grid_out);
149 s2.EdgeColor = 'none';
150 colorbar;
151
152 function [grid, count] = relax_func(grid_size, b1, b2, b3, b4, e
    )
153     % grid_size is the size of the grid. The grid is a square of
          size
154     % grid_size * grid_size
155     % b1 to b4 are the boundary functions called phi in the
        slides
156     % b1(0, y); b2(1, y); b3(x, 0); b4(x, 1)
157     % e is the required accuarcy
158
159     % Calculate the average value of the boundaries
160     x_array = 1 : grid_size;
161     x_array = x_array / grid_size;
162     b1_vals = arrayfun(b1, x_array);
163     b2_vals = arrayfun(b2, x_array);
164     b3_vals = arrayfun(b3, x_array);
165     b4_vals = arrayfun(b4, x_array);
166     k = mean(b1_vals + b2_vals + b3_vals + b4_vals) / 4;
167
168     % Create the grid and set the boundary values, otherwise
        initialise
169     % with k
170     grid = repmat(k, grid_size, grid_size);
171     grid(1,:) = b1_vals;
172     grid(grid_size,:) = b2_vals;
173     grid(:,1) = b3_vals;
174     grid(:,grid_size) = b4_vals;
175
176     count = 0;
177     % Keep averaging until the required accuray is achieved
178     done = false;
179     %residuals = zeros(grid_size, grid_size);
180     residuals = grid;
```

```
181     while ~done
182         done = true;
183         for j = 2 : grid_size - 1
184             for i = 2 : grid_size - 1
185                 residuals(i, j) = 0.25 * (grid(i+1, j) + grid(i
                        -1, j) + grid(i, j+1) + grid(i, j-1));
186                 r = abs(grid(i,j)-residuals(i, j));
187                 if r >= e
188                     done = false;
189                 end
190                 grid(i, j) = residuals(i, j);
191             end
192         end
193         count = count + 1;
194     end
195     count
196 end
```

SOR code and tracking computation times (SOR.m)

```
SORval = zeros(1, 200);
xvals = zeros(1, 200);
n = 1;
for i = 1:0.005:1.995
    % The test script for relaxation and SOR

    % Set the boundary function b1, b2, b3, b4
    b1 = @(y) 2*cos(4*pi*y);
    b2 = @(y) 2*cos(2*pi*y);
    b3 = @(x) 2*cos(4*pi*x);
    b4 = @(x) 2*cos(2*pi*x);

    grid_size = 100;
    required_accuracy = 0.000001;

    [grid_out, count] = SOR_fun(grid_size, b1, b2, b3, b3,
       required_accuracy, i);
    SORval(n) = count;
    xvals(n) = i;
    n = n + 1;
end

figure
hold on
title('Varying relaxation parameter');
xlabel('Relaxation parameter value');
ylabel('Count');
plot(xvals, SORval);


function [grid, count2] = SOR_fun(grid_size, b1, b2, b3, b4, e,
   relaxIn)
    % grid_size is the size of the grid. The grid is a square of
        size
    % grid_size * grid_size
    % b1 to b4 are the boundary functions called phi in the
       slides
    % b1(0, y); b2(1, y); b3(x, 0); b4(x, 1)
    % e is the required accuarcy

    % Calculate the average value of the boundaries
```

```
38      x_array = 1 : grid_size;
39      x_array = x_array / grid_size;
40      b1_vals = arrayfun(b1, x_array);
41      b2_vals = arrayfun(b2, x_array);
42      b3_vals = arrayfun(b3, x_array);
43      b4_vals = arrayfun(b4, x_array);
44      k = mean(b1_vals + b2_vals + b3_vals + b4_vals) / 4;
45
46      % Create the grid and set the boundary values, otherwise
           initialise
47      % with k
48      %grid = repmat(k, grid_size, grid_size);
49      grid = zeros(grid_size, grid_size);
50      grid(1,:) = b1_vals;
51      grid(grid_size,:) = b2_vals;
52      grid(:,1) = b3_vals;
53      grid(:,grid_size) = b4_vals;
54
55      %create relaxation parameter op
56      h = 1/(grid_size + 1);
57      %relax = 2 - (pi*h); %optimal?
58      %relax = 0.2;
59      %relax = 1;
60      relax = relaxIn;
61
62      count2 = 0;
63      % Keep averaging until the required accuray is achieved
64      done = false;
65      residuals = grid;
66      while ~done
67          done = true;
68          for j = 2 :1: grid_size - 1
69              for i = 2 :1: grid_size - 1
70                  %r = ((1 - relax) * grid(i, j)) + (0.25 * relax
                       * (grid(i+1, j) + grid(i-1, j) + grid(i, j+1)
                        + grid(i, j-1) - 4*grid(i, j)));
71                  residuals(i, j) = ((1 - relax) * grid(i, j)) +
                       (0.25 * relax * (grid(i+1, j) + grid(i-1, j)
                       + grid(i, j+1) + grid(i, j-1)));
72                  r = abs(residuals(i, j)-grid(i,j));
73                  if r >= e
74                      done = false;
```

```
75                      end
76                      grid(i, j) = residuals(i, j);
77                  end
78              end
79
80              count2 = count2 + 1;
81          end
82          %disp('SOR');
83          %disp(count2);
84          count2
85      end
```