Question 1

a) The error associated with the lower difference and upper difference of 0.25 are $2.7755575615628914 \times 10^{-17}$ and $5.551115123125783 \times 10^{-17}$ respectively.
The fractional range was determined to be $1.6653345369377348 \times 10^{-16}$, by dividing the mid-point of the lower difference and upper difference by 0.25.

b) The nearest values of 0.25 were determined to be (0.25000000000000006, 0.24999999999999997), by subtracting the lower difference from 0.25 and adding the upper difference to 0.25.

Similarly, the nearest values of the upper bound are determined to be (0.2500000000000001, 0.25) and the nearest values of the lower bound are (0.25, 0.24999999999999994)

The fractional rounding range of the upper bound and lower bound were determined to be $1.1102230246251563 \times 10^{-16}$ and $8.326672684688675 \times 10^{-16}$.

The nearest values found using the numpy.nextafter functions agree with the results are above, and hence these results are validated.

Since my computer uses 64 bits to store numbers, the smallest representable number would be around the order of magnitude of 2^{-52} = $2.220446049250313 \times 10^{-16}$. Therefore, the result from the program is in the correct order of magnitude.

Question 2

a) See computational_functions.py. The function *crouts* can factorise any N×N matrix into a lower diagonal matrix, **L**, and a upper diagonal matrix, **U**, using Crout's algorithm. The *crouts* function was validated by multiplying **L** and **U,** which returned **A,** as expected.

b) By *using* the function from (a), a matrix **A** is factorized into lower diagonal and upper diagonal matrices:

$$\mathbf{A} = \begin{bmatrix} 3 & 1 & 0 & 0 & 0 \\ 3 & 9 & 4 & 0 & 0 \\ 0 & 8 & 20 & 0 & 0 \\ 0 & 0 & -22 & 31 & -25 \\ 0 & 0 & 0 & -35 & 61 \end{bmatrix}$$

$$\mathbf{L} = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & -1.375 & 1 & 0 \\ 0 & 0 & 0 & -0.78212 & 1 \end{bmatrix}$$

$$\mathbf{U} = \begin{bmatrix} 3 & 1 & 0 & 0 & 0 \\ 0 & 8 & 4 & 0 & 0 \\ 0 & 0 & 16 & 10 & 0 \\ 0 & 0 & 0 & 44.75 & -25 \\ 0 & 0 & 0 & 0 & 41.44693 \end{bmatrix}$$

Concisely, a matrix **M** can be expressed with all the elements of **U** and the non-diagonal elements of **L:**

$$\mathbf{M} = \begin{bmatrix} 3 & 1 & 0 & 0 & 0 \\ 1 & 8 & 4 & 0 & 0 \\ 0 & 1 & 16 & 10 & 0 \\ 0 & 0 & -1.375 & 44.750 & -25 \\ 0 & 0 & 0 & -0.782 & 41.447 \end{bmatrix}$$

The determinant of the matrix is calculated to be 712224, by multiplying all the elements of the diagonal of **U,** which agrees with the value of the determinant calculated by the numpy built-in function.

c) See computational_functions.py. The function *solve_lu* can solve the matrix equation $\mathbf{L}\cdot \mathbf{U} \cdot \vec{x} = \vec{b}$**,** using forward substitution and backward substitution.

d) Using this function from (c), the vector $\vec{x}$ is solved to be (0.46168059, 0.61495822, -0.47991643, 0.06786629, 0.18648066).

e) Since $\mathbf{A}^{-1} \cdot \mathbf{A}$ = **I**, where **I** is the identity matrix, the inverse matrix $\mathbf{A}^{-1}$ can be solved column by column as follow:

$$\mathbf{L}\cdot \mathbf{U} \cdot \overrightarrow{a'_j} = \boldsymbol{I}_j$$

where *j* denotes the column to be solved.

The inverse matrix is then solved to be:

$$\mathbf{A}^{-1} = \begin{bmatrix} 0.38071 & -0.04737 & 0.00571 & -0.00343 & -0.00140 \\ -0.14212 & 0.14212 & 0.01712 & 0.01028 & 0.00421 \\ 0.03423 & -0.03424 & 0.03424 & -0.02056 & -0.00842 \\ 0.04522 & -0.04522 & 0.04522 & 0.03289 & 0.01348 \\ 0.02594 & -0.02595 & 0.02595 & 0.01887 & 0.02413 \end{bmatrix}$$

The inverse matrix was calculated with the numpy built-in function as well, which gives the same result, and this validates the *lu_solve* function.

Question 3

a) See computational_functions.py. The function *lagrange_pol* interpolates at an input x value from a set of x and y data and this function was validated with the scipy built-in function.
b) See computational_functions.py. The function *cubic_spline* interpolates at an input x value from a set of x and y data and this function was validated with the scipy built-in function.
c) The Lagrange polynomial interpolation and cubic spline interpolation are plotted on the same graph (see below).

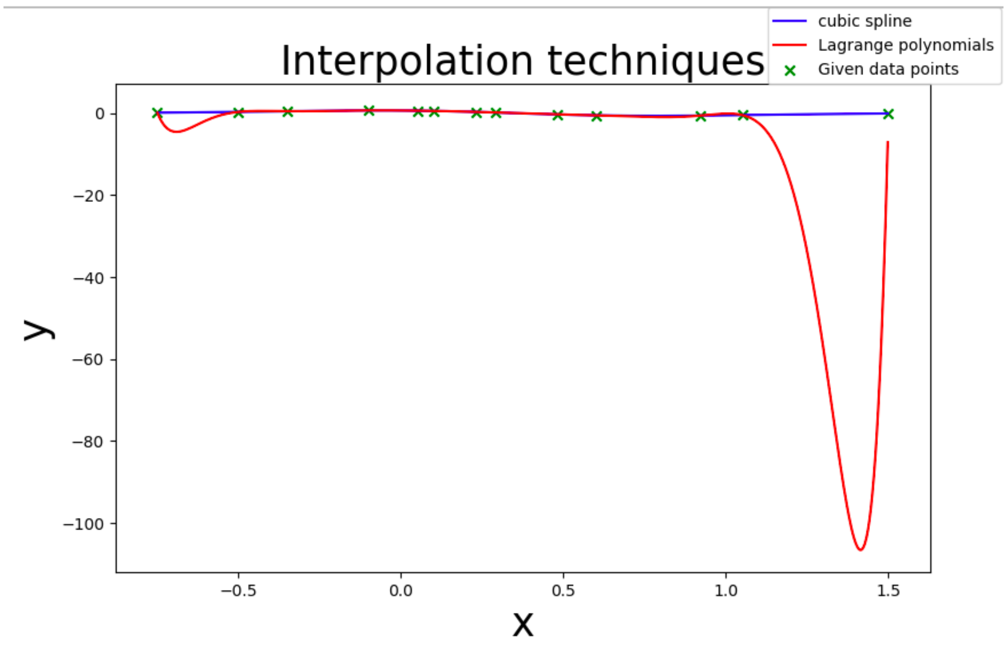

Figure 3.1. The data points, Lagrange polynomial interpolation and cubic spline interpolation are plotted together.

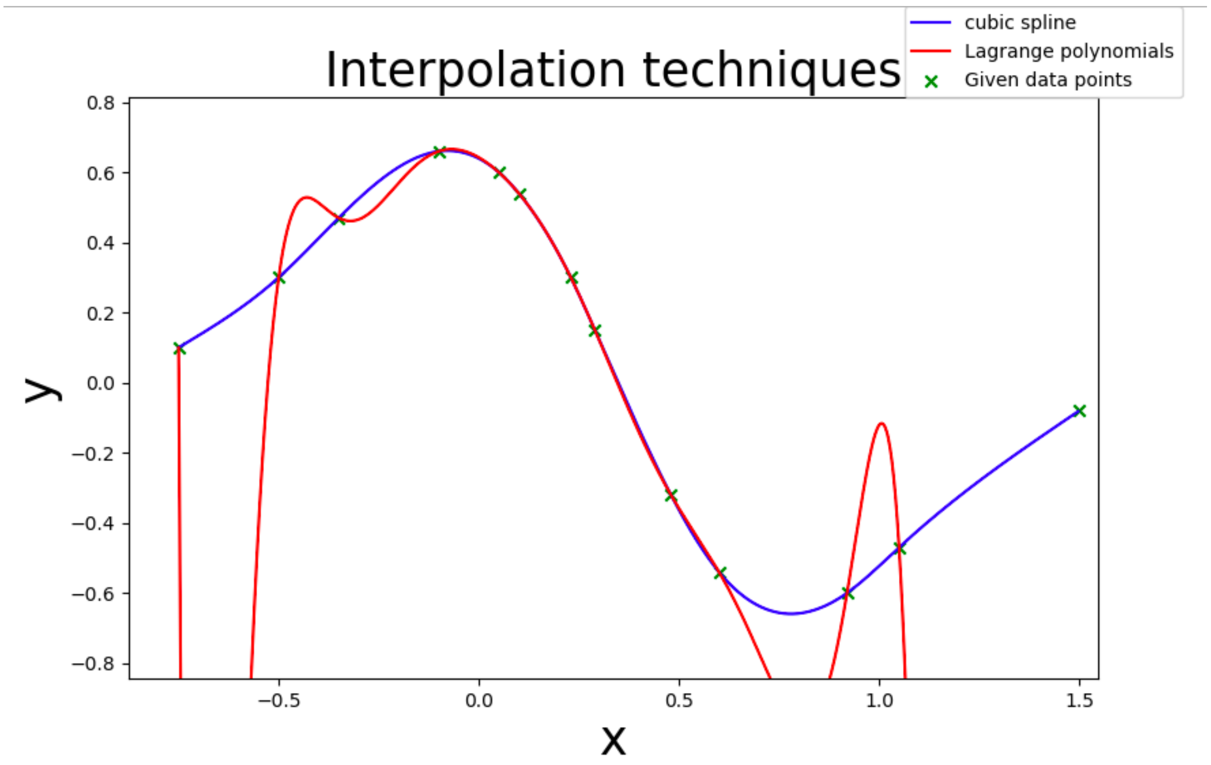

Figure 3.2. Same as Figure 1.1, but enlarged to show the cubic spline polynomials with more detail.

Both curves are smooths and analytical in the given range and both go through all the data points. However, it is clear that the Lagrange polynomial method overfits the data with a wide range of gradients, which is because the it fits one function over the whole range with 12 polynomials. On the other hand, the cubic spline method doesn't overfit the data and seems to be a better fit, because there is a different function in each range, and that the second order derivatives are ensured to be zero.

<u>Question 4</u>

a) First g(t) and h(t) were fourier transformed, then the convolution was obtained by the inverse fourier transform of the product of g(t) and h(t). The sampling was taken from -20 to 20 so that it is symmetric about the origin. Then I added zeros to each end to pad. By zero padding, the frequency spacings were decreased. Since Fast Fourier transform works best with 2^N sampling points, 2^{13} sampling points were used (excluding the zeros). The sampling step size was chosen to be 0.0391, which corresponds to the Nyquist frequency to be 643.4rads^{-1}, which is sufficient to minimize the distortion to the fourier transform due to aliasing and is believed to be the right compensation of computational efficiency and aliasing.

b)

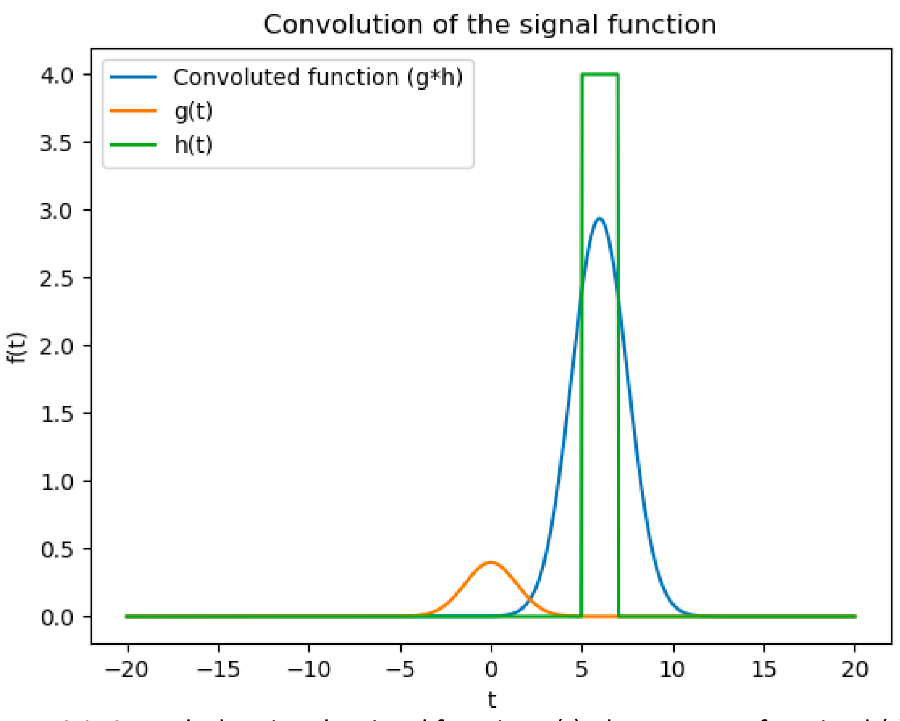

Figure 4.1. A graph showing the signal function g(t), the response function h(t) and the convoluted function g*h.

<u>Question 5</u>

a) By substituting $Q = CV_{out}$, the given first order differential equation can be written as follow,

$$\frac{dV_{out}(t)}{dt} = \frac{V_{in}(t) - V_{out}(t)}{RC}$$

By using a scaled variable t', this can be written as:

$$\frac{dV_{out}(t')}{dt} = V_{in}(t') - V_{out}(t')$$

where $t' = \frac{t}{RC}$.

b) See computational_functions.py. The *ode_methods* can return an approximate solution to any first order differential equations, using Adams-Bashforth method or Runge-Kutta method. The second, third and fouth initial points used in the Adams-Bashforth methods were estimated using the Euler method and leapfrog method.

c) In this task, V_0 was set to be 10 and R and C were both set to 1. Two approximate solutions obtained were plotted on the same graph as the analytic solution.

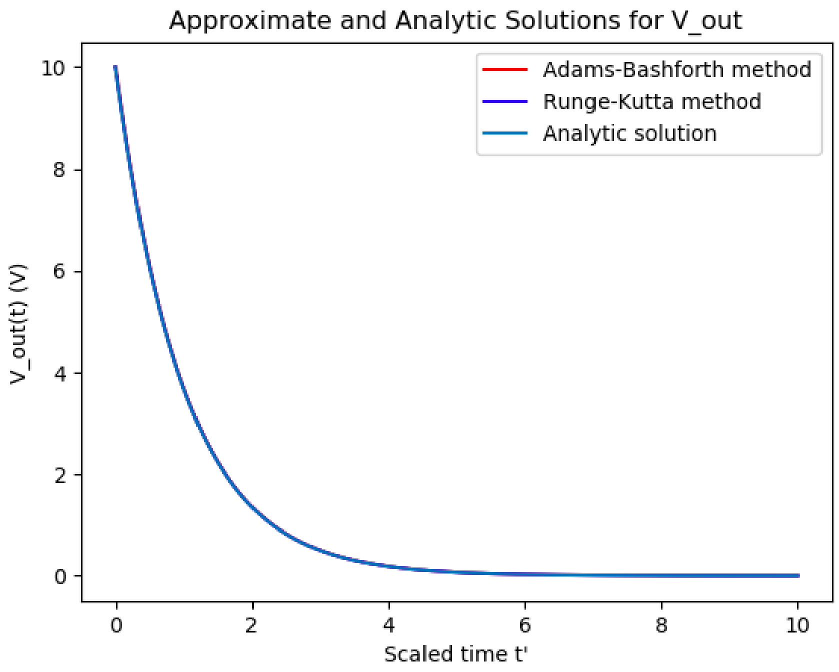

Figure 5.1. A graph showing the analytic solution and the approximate solutions obtained from Adams-Bashforth method and Runge-Kutta.

As seen from figure 5.1, the Adams-Bashforth method and Runge-Kutta method are both very good approximation to the analytic solution, and hence, figure 5.1 seems to be just one line. This in fact also validates the alogorithms implemented for the Adams-Bashforth method and Runge-Kutta method.

d) Errors were determined by finding the difference between the approximated values and the analytical values.

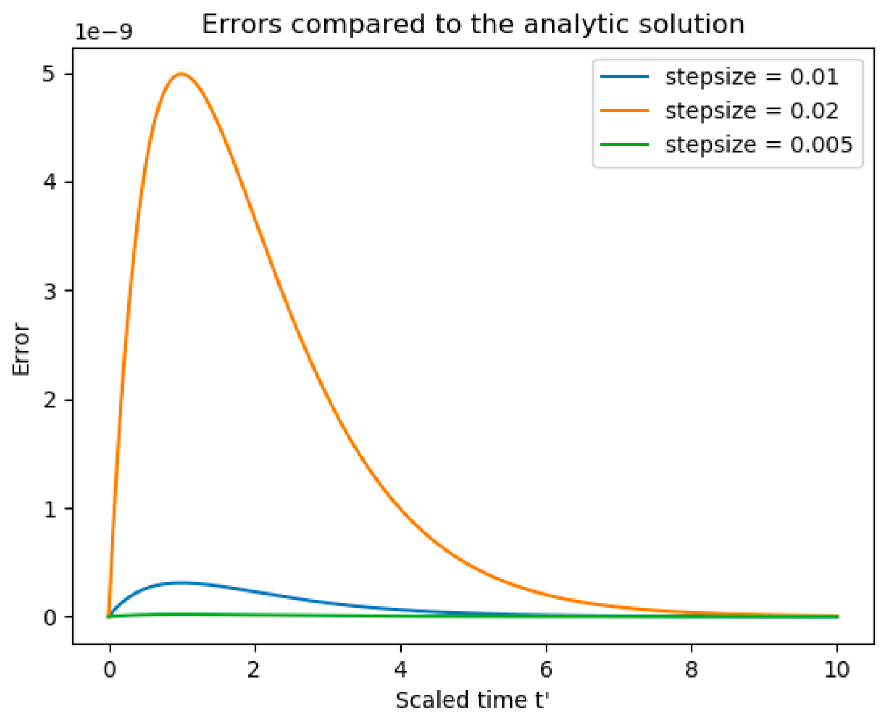

Figure 5.2. A graph showing the error of the Runge-Kutta methods with different step size.

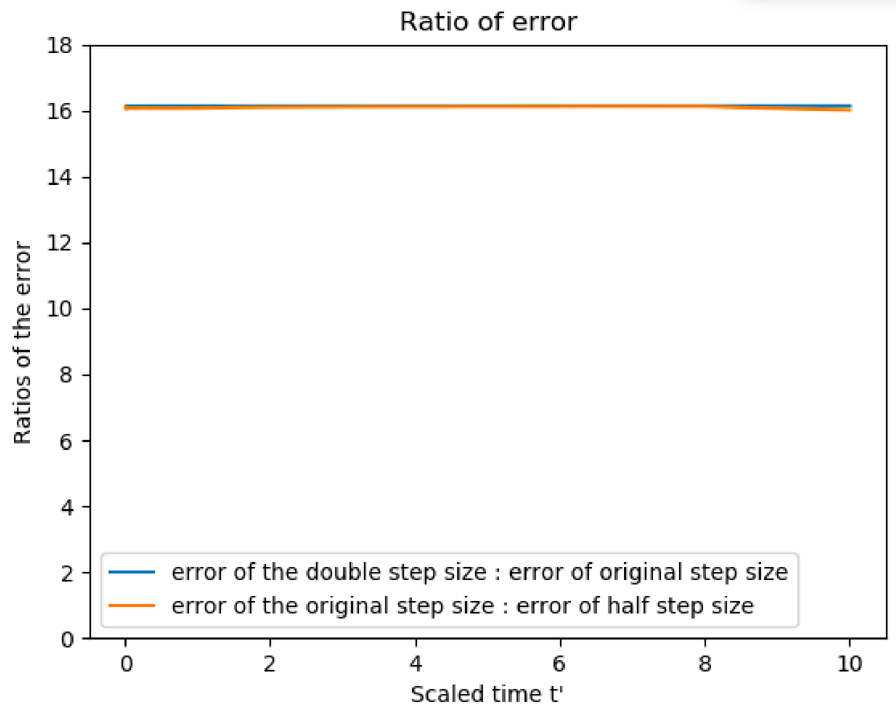

Figure 5.3. The graph shows the ratio of the erros of different step sizes.

As seen from figure 5.2 and 5.3, when the step size is doubled, the error increases by 16 times. This shows the error scales as O(h^4), where h is the step size. This is expected because the Runge-Kutta method is a fourth order approximation.

e) Square waves of different time period were used as the $V_{in}(t)$.

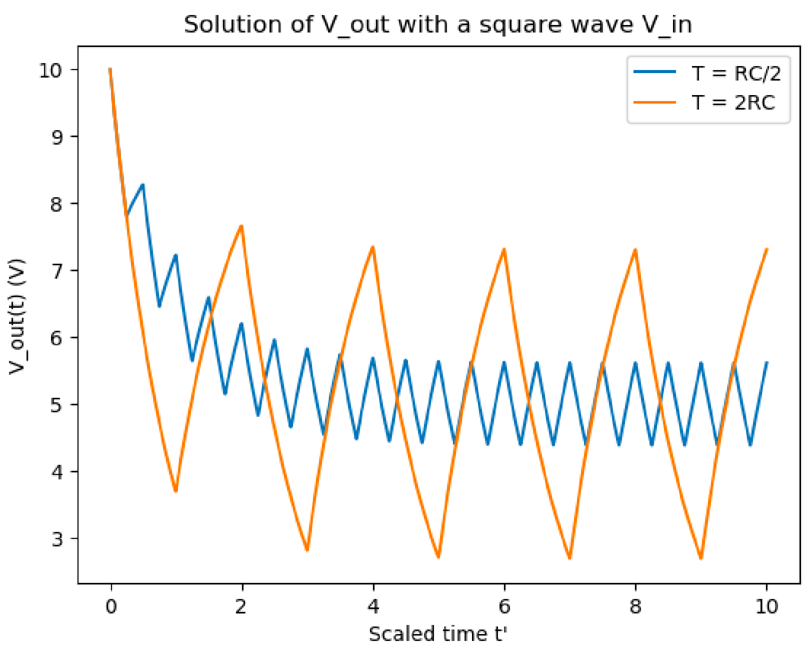

Figure 5.4. This graph shows how $V_{out}(t)$ behaves with square waves as $V_{in}(t)$ of different time period.

It is clear that the amplitudes of both $V_{out}(t)$ decay away exponentially, and that the one of longer period has bigger amplitude. This in fact agrees with the prediction, as a larger time period means, it can store more charge in one period before discharging and hence giving a larger amplitude.