# 3801ICT Numerical Algorithms
# Assignment Part One

Ethan Leet
s5223103
ethan.leet@griffithuni.edu.au

April 23, 2022

# Contents

# 0 Introduction

## 0.1 Overview

This assignment involved writing c++ code to solve six Numerical Algorithm based problems. Experiments were then conducted to prove the validity of each of the six problems. Each question is broken down into several sub sections throughout this report. The sub sections include; problem statement, algorithm description, complexity, and experiments and results.

## 0.2 Compilation and Runtime

All programs included in this submission compile without errors nor warnings using clang++ and the c++ standard library 14. Each program can be compiled using the following command from the question folder, using the question number in place of x.

**clang++ main.cpp -std=c++14 -o questionx -Ofast**

Alternatively, each question also contains a make file which runs the above command. Finally, there is also a make file in the parent folder which will make every program in this submission and place the binaries in the respective question folders. To use the make command, CMake must be installed and then the following command can be executed from either the parent or child folder.

**make**

After successful compilation programs can then be executed with the following command, using the question number in place of x. Note that in order to execute the following command you must be inside the question folder.

**./questionx**

# 1 Centred Difference Approximation

## 1.1 Problem Statement

A centered difference approximation of the first derivative can be written as:

$$\underset{\text{True value}}{f'(x_i)} = \underset{\substack{\text{Finite-difference}\\\text{approximation}}}{\frac{f(x_{i+1}) - f(x_{x-1})}{2h}} \underbrace{-\frac{f^{(3)}(\xi)}{6}h^2}_{\text{Truncation error}}$$

However, as we are using a computer, the function values in the numerator of the finite-difference approximation include round-off errors as follows:

$$f(x_{i-1}) = \tilde{f}(x_{i-1}) + e_{i-1}$$
$$f(x_{i+1}) = \tilde{f}(x_{i+1}) + e_{i+1}$$

Substituting these values we get:

$$\underset{\text{True value}}{f'(x_i)} = \underbrace{\frac{\tilde{f}(x_{i+1}) - \tilde{f}(x_{x-1})}{2h}}_{\substack{\text{Finite-difference}\\\text{approximation}}} + \underbrace{\frac{e_{i+1} - e_{e-1}}{2h}}_{\text{Round-off error}} - \underbrace{\frac{f^{(3)}(\xi)}{6}h^2}_{\text{Truncation error}}$$

Assuming that the absolute value of each component of the round-off error has an upper bound of $\epsilon$, the maximum possible value of the difference $e_{i+1} - e_{i-1}$ will be $2\epsilon$. Further, assume that the third derivative has a maximum absolute value of M. An upper bound on the absolute value of the total error can therefore be represented as:

$$\text{Total Error} = \left| \tilde{f}(x_i) - \frac{\tilde{f}(x_{i+1}) - \tilde{f}(x_{x-1})}{2h} \right| \leq \frac{\epsilon}{h} + \frac{h^2 M}{6}$$

An optimal step size can be determined by differentiating this equation, setting the result equal to zero and solving to give:

$$h_{opt} = \sqrt[3]{\frac{3\epsilon}{M}}$$

Given:

$$x = 0.5, f(x) = -0.1x^4 - 0.15x^3 - 0.5x^2 - 0.15x + 1.2$$

use a centred-difference approximation to estimate the first derivative of this function with varying values of h to demonstrate the validity of the analysis above and the impact of both round-off and truncation errors.

## 1.2 Algorithm Description

It was observed that the numerical optimal h as well as the effect of both round-off and truncation errors would be different for different data types. As such, this program was templated to work for three different data types; double precision, single precision and half precision floating point types.

The first step in solving this problem was to predict the numerical optimal h value. In order to do this, numerous h values were tested for each data type.

The templated predict function created a loop over different h values with varying ranges and increments. This function works out the derivative of the given function at the different h values by using the Central Difference formula:

$$f'(x) = \frac{f(x+h) - f(x-h)}{2h} + O(h^2)$$

It is proven that the Central Difference formula approaches the derivative as $h^2$ tends to zero.

The $\log_{10}$ of both the derivative and true error are then saved for graphing and further experimentation. The function returns the predicted optimal h based on the lowest true error.

The next step in this problem is finding the theoretical optimal h based on machine epsilon for each of the three datatypes:

$$h_{opt} = \sqrt[3]{\frac{3\epsilon}{M}}$$

Next, the theoretical optimal h was then passed to the Central Difference formula (above) and the derivative at that h calculated.

Finally, the graphs created by predicting the numerical optimal h values are produced and line equations calculated using Python. These line equations are then solved to confirm the predicted optimal h value found above.

## 1.3 Complexity

### 1.3.1 Space Complexity

The program only holds one best h value at a time meaning that its space complexity at runtime is of order O(1). The program does export data to a text file which holds all predicted h values for the three different data types. This file holds N data points three times which equates to an order of O(n) where n is the range of the upper and lower limits of the loop in respect to the increment passed. Overall, the program has a space complexity of O(n).

### 1.3.2 Time Complexity

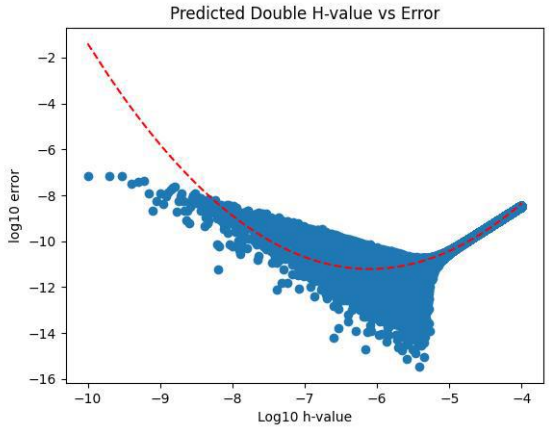
The main work of the program is done in the predict_h function. This function is executed n times for each different data type. Where n is the range of the upper and lower limits of the loop in respect to the increment passed. This means that all other functions which run in O(1) are executed n times, giving a total program time complexity of O(n).

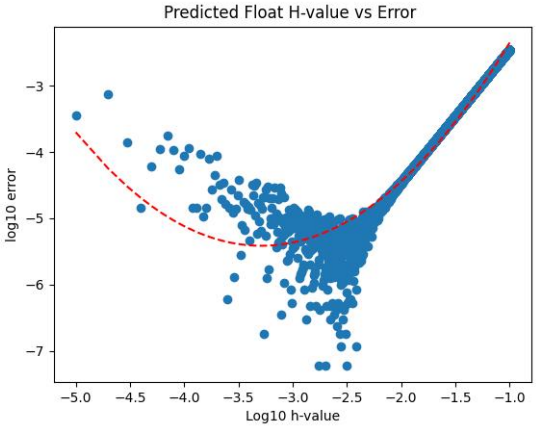
## 1.4 Experiments and Results

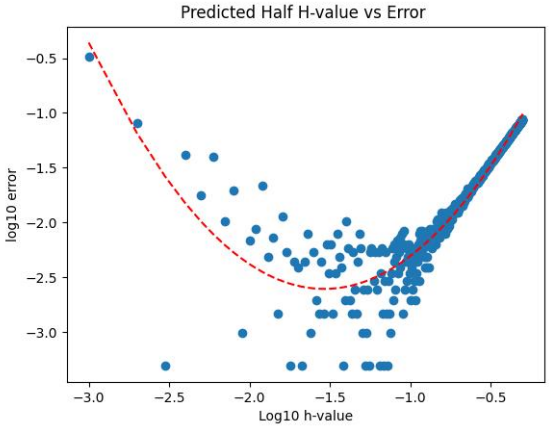
To assess the validity of the problem statement analysis the following h values were tested:

| Type | Lower | Upper | Increment | Predicted h |
|---|---|---|---|---|
| Double | $1 \times 10^{-10}$ | $1 \times 10^{-4}$ | $1 \times 10^{-10}$ | $3.87 \times 10^{-6}$ |
| Single | $1 \times 10^{-5}$ | 0.1 | $1 \times 10^{-5}$ | 0.00237 |
| Half | $1 \times 10^{-3}$ | 0.5 | $1 \times 10^{-3}$ | 0.0582 |

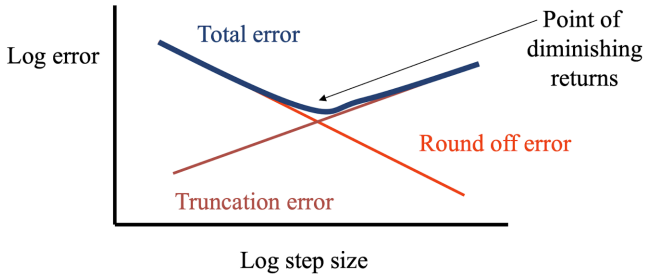
The issue with using this predicted h value is that it is exposed to the effects of round-off and truncation errors. To combat this exposure to error, all h values within the above ranges and the true error of each of these h values were output to a file and the $log_{10}$ of their values graphed and a trendline fitted:






The trendline for each graph is the total error of each predicted h value. At the turning point (when it turns from negative to positive), is our point of diminishing return, and where our predicted optimal h value truly lies. An example of a graph showing total error is sourced from lecture material and can be seen below:




Upon examination of this example error graph, the effect of truncation and round-off errors in the above three graphs for double, single and half precision is evident. The impact of these errors is most easily seen in the double precision graph. Almost all data points surrounding the trendline are effected by both truncation and round-off errors. The impact is slightly less for single precision types however still noticeable. The truncation error impact is negligible for half precision except for some outliers and the round-off error for half precision has much less of an impact compared to single and double precision. In order to gain a deeper understanding of truncation and round-off errors, further experimentation is needed in regards to the half precision type and even smaller types such as quarter precision floating point numbers.

In order to confirm the predicted numerical optimal h value the trendline for the above graphs were calculated:

$$double : y = 0.3577x^2 - 9.2 \times 10^{-7}x + 2.3 \times 10^{-11}$$

$$single : y = 0.3525x^2 - 2.9 \times 10^{-4}x + 7 \times 10^{-6}$$

$$half : y = 0.5029x^2 + -0.0951x + 0.0128$$

Upon solving the above trendline formulae the below predicted numerical optimal h values are produced:

| Type | Predicted h |
|---|---|
| Double | $1.3 \times 10^{-6}$ |
| Single | $4.2 \times 10^{-4}$ |
| Half | 0.0945 |

It is worth noting that these predicted numerical optimal h values are different to those originally predicted by the prediction function. This is due to the function returning the optimal h based on lowest true error. As seen above, true error is susceptible to round-off and truncation error.

Next, the theoretical optimal h was calculated based on machine epsilon:

| Type | Theoretical h |
|---|---|
| Double | $6.8 \times 10^{-6}$ |
| Single | $5.5 \times 10^{-3}$ |
| Half | 0.1117 |

Finally, the derivative values were calculated using predicted and theoretical optimal h:

| Type | Predicted | Theoretical |
|---|---|---|
| Double | -0.8125 | -0.8125 |
| Single | -0.812468 | -0.812511 |
| Half | -0.819336 | -0.81687 |

Interestingly, the Double precision derivative value is closest to the analytical derivative of -0.7125. This is due to the optimal h value being captured by both the double precision theoretical and predicted calculations. Even though it is most susceptible to round-off and truncation errors it has the precision to be able to capture more values, and in this case, the optimal h values needed to produce the closest result to the analytical derivative.

# 2 Multiple-Application Simpson's Rule

## 2.1 Problem Statement

Use the Multiple-Application Simpson's Rule to evaluate the distance travelled where the velocity v as a function of time is as follows:

$$\begin{aligned} v &= 11t^2 - 5t & 0 \leq t \leq 10 \\ v &= 1100 - 5t & 10 \leq t \leq 20 \\ v &= 50t + 2(t-20)^2 & 20 \leq t \leq 30 \end{aligned}$$

In addition use numerical differentiation to develop graphs of acceleration $\frac{dv}{dt}$ and rate of change of acceleration for $t = 0$ to $t = 30$ .

## 2.2 Algorithm Description

The first part of this problem involves using the Multiple-Application Simspon's 1/3 Rule to evaluate distance travelled (integral of the function) where the velocity is a function of time (seen above). The Multi-Application Simpson's 1/3 Rule is described as:

$$I = \frac{h}{3}\left(f(x_0) + 4\sum_{i=1,3,5\ldots}^{n-1} f(x_i) + 2\sum_{j=2,4,6\ldots}^{n-2} f(x_j) + f(x_n)\right)$$

In this formula, h is described as the step size. The step size for this problem is described as:

$$h = \frac{b-a}{t}$$

Where $a$ and $b$ are the lower and upper limits of the function respectively and $t$ is the current culminative time segment. As an example, if $t = 6$, $a = 0$ and $b = 10$.

The Simpson's 1/3 Rule evaluates the integral as some point $t$ by summing the odd number of points and even number of segments separately, ranging from 1 to $t - 1$ for odd $t$ and 2 to $t - 2$ for even $t$.

It can be seen that there are three different functions of velocity dependant on time. The Multi-Application Simpson Rule function keeps track of the current time value in order to appropriately evaluate the function in the correct time period.

The second part of this problem involved creating graphs of acceleration and rate of change of acceleration from time = 0 to time = 30. It is given that acceleration is the derivative of velocity in respect to time and it is proven that rate of change of acceleration is the second derivative of velocity in respect to time.

In order to work out the first derivative of the given functions of time the Richardson Extrapolation technique was used. This technique involves using a matrix D where

each columns $0_{th}$ entry is evaluated using the Central Difference Formula (seen below) and each other entry is evaluated using the following formula where n and m are the row and column entries of the matrix D respectively. The technique is repeated within a loop until the calculation produced is within some defined error tolerance. The technique is as follows:

$$D(n,m) = D(n,m-1) + \frac{1}{4^m - 1}\left[D(n,m-1) - D(n-1,m-1)\right]$$

The Central Difference Formula is described as:

$$f'(x) = \frac{f(x+h) - f(x-h)}{2h} + O(h^2)$$

It is proven that the Central Difference formula approaches the derivative as $h^2$ tends to zero. It can be seen that as the Central Difference formula takes a $x+h$ and $x-h$ value, where $x$ is the current time value, the time can be pushed over or under the appropriate time value for this velocity. As an example, if t = 9.8 and h = 0.5, $x - h$ will be in the first velocity function whereas $x + h$ will be in the second velocity function. To handle this special case a percentage of each function the new value belongs to was applied before working out the Central Difference. In the previous example, 40% lies in the $t \leq 10$ velocity function and 60% lies in the $t \leq 20$ velocity function. Therefore, 40% of $x + h$ is evaluated in the $t \leq 10$ velocity function and 60% is evaluated in the $t \leq 20$ velocity function. As $x - h$ remains in the current time value it is left as normal for this particular time segment.

In order to calculate the rate of change of acceleration the following formula was used to evaluate the second derivative:

$$f"(x) = \frac{f(x+h) - 2f(x) + f(x-h)}{h^2}$$

Similarly to the Central Difference calculation a percentage of each $x + h$ and $x - h$ function was also calculated to handle the special overflow and underflow cases.

## 2.3 Complexity

### 2.3.1 Space Complexity

For the first part of this problem three vectors were used to hold the segment calculations for each respective time value calculation, giving a complexity of O(n) where n = the maximum t value.

In order to calculate the Richardson Extrapolation a $n \times m$ matrix was created dynamically, meaning that $m = n$ which gives a space complexity of O($n^2$). In this program the matrix grows by n rows and columns depending on the error tolerance given. Through experimentation it was proven that for the given problem $n = 3$ for an error tolerance of 6 significant figures. The error tolerance is described below where s in the amount of significant figures (6 in this program):

$$\epsilon_s = (0.5 \times 10^{2-s})\%$$

All other functions in this program operate in constant space and do not contribute to the complexity of this program. At first glance the program appears to be O($n^2$) due to the Richardson Extrapolation function, however, as it was observed that $n = 3$ for Richardson Extrapolation, the program has a total space complexity of O(n) as time(t) = 30 > 3.

### 2.3.2 Time Complexity

The time complexity for the Multi-Application Simpson's Rule is O(n) where n is the amount of segments used.

The Richardson's Extrapolation technique complexity is dependant on the error tolerance above. Therefore, it can be seen that a worst case time complexity for this function is O(n) where n is the size of the matrix. However, as it was previously observed that $n = 3$ a more accurate complexity can be described in constant time, $\Theta(1)$.

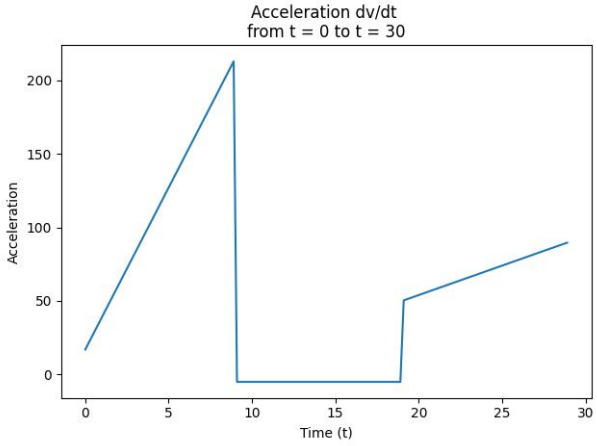
As all other function simply return a value, meaning they run in constant time, an overall time complexity for this program is O(n).

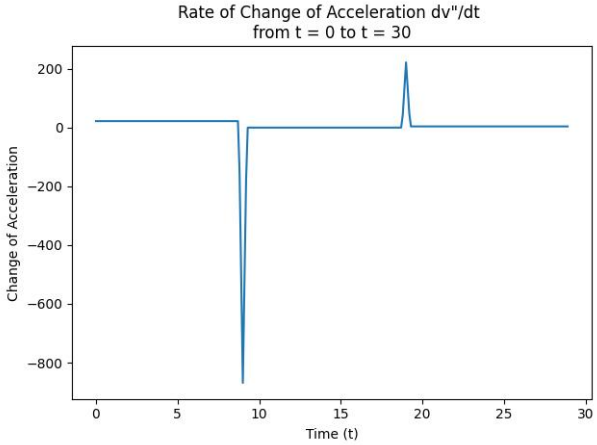
## 2.4 Experiments and Results

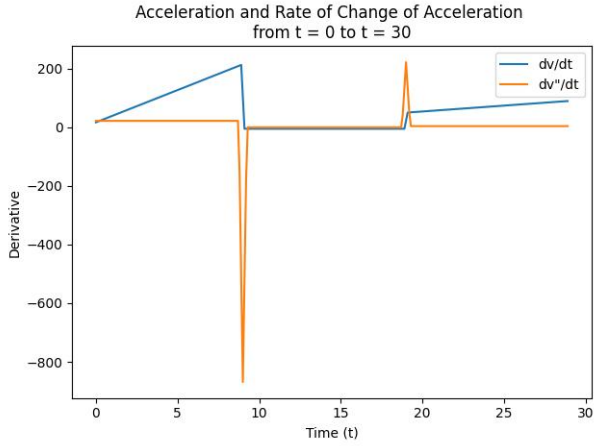
In order to evaluate the distance travelled where the velocity is a function of time, ten segments were used for thirty different time values (1 to 30). Each segment consists of three time values. As an example, segment 1 holds the result of the Multi-Application Simpson's Rule at t = 1, t = 11 and t = 21, segment 2 holds t = 2, t = 12 and t = 22. The Multiple-Application Simpson's Rule is limited to cases where values are equispaced, meaning that there are an even number of segments and and odd number of points. This limitation can be seen in the results tabulated below, every even segment produced the analytical answer of 26,833.33 whereas the odd segments produced varying incorrect answers.

| | |
|---|---|
| Segment 1 | 25,211.46 |
| Segment 2 | 26,833.33 |
| Segment 3 | 23,555.71 |
| Segment 4 | 26,833.33 |
| Segment 5 | 24,054.73 |
| Segment 6 | 26,833.33 |
| Segment 7 | 24,468.51 |
| Segment 8 | 26,833.33 |
| Segment 9 | 24,772.99 |
| Segment 10 | 26,833.33 |

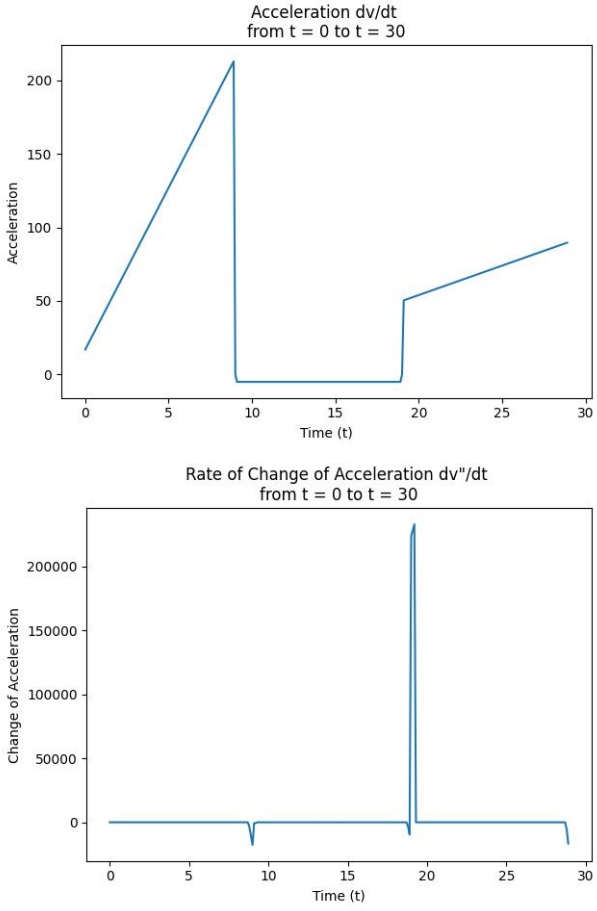
When graphs of acceleration and rate of change of acceleration were developed large peaks and discontinuities can be seen. These discontinuities can be seen below and are evident because of the function changes when $x + h$ or $x - h$ (as described in section 2.2).


Acceleration dv/dt
from t = 0 to t = 30
Acceleration
Time (t)
200
150
100
50
0
0
5
10
15
20
25
30


Rate of Change of Acceleration dv"/dt
from t = 0 to t = 30
Change of Acceleration
Time (t)
200
0
−200
−400
−600
−800
0
5
10
15
20
25
30


Acceleration and Rate of Change of Acceleration
from t = 0 to t = 30
Derivative
Time (t)
dv/dt
dv"/dt
200
0
−200
−400
−600
−800
0
5
10
15
20
25
30

In order to combat these discontinuities and try to smooth out the function changes a percentage calculation (described in section 2.2) was performed. A slight change can be seen at t = 9 where the acceleration starts to curve towards the function change at t = 10, but for the most part this made no visual change. However, when viewing the outputed text file, each function change has multiple values for that point rather than just one value as per not performing the percentage check. This means that although a completely smooth function free of discontinuities could not be achieved based on the number of segments used, this technique did start to smooth the function. Further tests would need to be conducted with exponentially higher segments in order to rid this function of all discontinuities.



# 3 Centered Difference Approximation and Richardson Extrapolation

## 3.1 Problem Statement

Compare the Central Difference and Richardson Extrapolation methods for finding the value of the first derivative of $f(x) = -0.2x^4 - 0.30x^3 - 1.5x^2 - 0.45x + 2.6$ at x = 0.5. Explore the effect of using different values of h.

## 3.2 Algorithm Description

To solve this problem the Central Difference formula (below) was used across 100 different h values to evaluate the derivative.

$$f'(x) = \frac{f(x+h) - f(x-h)}{2h} + O(h^2)$$

It is proven that the Central Difference formula approaches the derivative as $h^2$ tends to zero.

The Richardson Extrapolation technique (below) was then used to evaluate the derivative. This technique involves using a matrix D where each columns $0_{th}$ entry is evaluated using the Central Difference Formula (seen above) and each other entry is evaluated using the following formula where n and m are the row and column entries of the matrix D respectively. The technique is repeated within a loop until the calculation produced is within some defined error tolerance.

$$D(n,m) = D(n,m-1) + \frac{1}{4^m - 1}\left[D(n,m-1) - D(n-1,m-1)\right]$$

## 3.3 Complexity

### 3.3.1 Space Complexity

The Central Difference function simply returns a value and therefore contributes a constant space complexity. In order to calculate the Richardson Extrapolation a $n \times m$ matrix was created dynamically, meaning that $m = n$ which gives a space complexity of O($n^2$). In this program the matrix grows by n rows and columns depending on the error tolerance given. Through experimentation it was observed that for the given problem $n = 3$ for an error tolerance of 6 significant figures. The error tolerance is described below where s in the amount of significant figures (6 in this program):

$$\epsilon_s = (0.5 \times 10^{2-s})\%$$

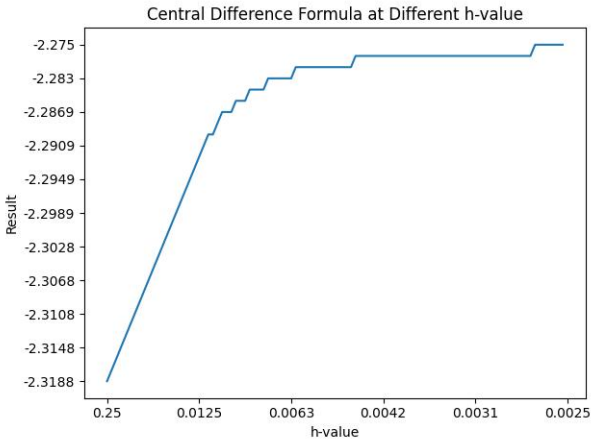
Therefore, a total space complexity is described as O($n^2$), where n is the size of the matrix produced by the Richardson Extrapolation technique. However, as it is proven that $n = 3$ a more accurate space complexity can be described as $\Theta(1)$.

### 3.3.2 Time Complexity

The Central Difference function simply returns a value so its runtime is constant. However, it is called 100 times meaning that this part of the program has a time complexity of O(n) where n is the amount of h values being tested. The Richardson's Extrapolation technique complexity is dependant on the error tolerance above. Therefore, it can be seen that a worst case time complexity for this function is O(n) where n is the size of the matrix. However, as it was previously observed that $n = 3$ a more accurate complexity can be described in constant time, $\Theta(1)$. As such, an overall runtime complexity for this program is described as O(n).

## 3.4 Experiments and Results

As seen in the below graph the Central Difference formula converges towards the analytical result at approximately $h = 0.0028$.




Further experimentation was conducted for a varying different range of h values. It was found that regardless of the range or increment size of h the Central Difference formula converged towards the analytical result at approximately $h = 0.0028$.

Through experimentation of different h values for the Richardson Extrapolation technique a matrix of size $2 \leq size \leq 3$ was always produced for any realistic h value. When $h \leq 100$ was chosen a matrix of size 3 was produced, and when $h = 0.0028$ was chosen a matrix of size 2 was produced. In all test cases an error tolerance (as described in section 3.3.1) was used.

When comparing the two techniques, if an optimal h value is known the complexity of the Central Difference formula is lower than that of the Richardson Extrapolation and should be considered as the preferred technique. However, in most scenarios an optimal h value is not known, as such, the Richardson Extrapolation technique should be used in this situation.

# 4 Romberg Integration

## 4.1 Problem Statement

A variable V is determined by:

$$V = \int_{t_1}^{t_2} P(t)d(t)dt$$

where:

$$P(t) = 9 + 4cos^2(0.4t)$$
$$d(t) = 5e^{-0.5t} + 2e^{0.15t}$$

Evaluate this integral between $t_1 = 2$ and $t_2 = 8$ using Romberg Integration with a tolerance of $0.1\%$.

## 4.2 Algorithm Description

The Romberg integration technique is comprised of successive applications of the trapezoidal rule technique. Similarly to the Richardson Extrapolation technique, Romberg integration involved operations on a matrix to compute the integral and can be seen below:

$$I_{j,k} \cong \frac{4^{k-1} I_{j+1,k-1} - I_{j,k-1}}{4^{k-1} - 1}$$

In this equation, I is a matrix of integral estimations. Each k refers to the Multiple-Application Trapezoidal Rule calculation (seen below) and as such; $k = 1$ implies $O(h^2)$ accuracy, $k = 2$ implies $O(h^4)$ accuracy, $k = 3$ implies $O(h^6)$ accuracy and $k = n$ implies $O(h^{2n})$ accuracy. Index j is used to distinguish between more accurate $(j + 1)$ estimations and less accurate $(j)$ estimations. The Multiple-Application Trapezoidal Rule is as follows:

$$I = \frac{b-a}{2n}\left[f(x_0) + f(x_n) + 2\sum_{i=1}^{n-1} f(x_i)\right]$$

In this equation, I is an integral estimation, a and b are the lower and upper limits of the integral respectively and n is the segment size.

For this program, the Romberg technique was implemented inside a loop which continually updates and checks current error. The loop continues until the error of the Romberg estimation is within $0.1\%$ of the analytical solution. Inside this loop the function calls the Multiple-Application Trapezoidal Rule function on each iteration to produce an updated k value. The parameters for a and b do not change for each successive Multiple-Application Trapezoidal Rule function call whereas the n parameter is multiplied by two on every iteration.

## 4.3 Complexity

### 4.3.1 Space Complexity

For the Romberg Integration function a $j \times k$ matrix is created dynamically meaning that $j = k$. The size of this matrix is determined by the tolerance dictated, in this case, $0.1\%$. For this tolerance level, it was observed that a matrix size of five was produced. The total space complexity for this program is O(j) where j is the size of the matrix and dependant on the tolerance used.

The Multiple-Application Trapezoidal Rule has one saved variable that is continuously updated, as such, it does not contribute to the worst case space complexity of this program.

### 4.3.2 Time Complexity

The time complexity for the Romberg integration technique is dependant on the tolerance used. As stated above, the tolerance level dictates the size of the matrix created, and therefore the depth of the loop used. As such, a worst case estimate for this function can be described as O(j) where j is the size of the matrix and dependant on the tolerance used.

The Multiple-Application Trapezoidal Rule function has a complexity of O(n) where n is the current segment size. This function is executed upon every iteration of the Romberg Integration function. Therefore, a total time complexity for this program can be described as $O(n \times j)$.

## 4.4 Experiments and Results

Using the supplied error tolerance of $0.1\%$ this program evaluated the integral as 322.3471 which is within $0.1\%$ of the analytical solution of 322.3483. Further experiments were conducted using trivial and not trivial integrals to confirm that this program produces the correct result. Experiments were then conducted to assess how the size of the matrix changes as the tolerance changes, the integral value was also noted. The results are as follows:

| Tolerance | Matrix Size | Integral |
|---|---|---|
| $5\%$ | 4 | 322.59572 |
| $2.5\%$ | 4 | 322.59572 |
| $1\%$ | 5 | 322.34571 |
| $0.1\%$ | 5 | 322.34571 |
| $0.01\%$ | 6 | 322.34837 |

As can be seen in the table above, the correct analytical result was a achieved when a tolerance of $0.01\%$ was used and did not require a significant space nor time complexity increase. To conclude, it can be seen that the Romberg Integration technique should be chosen as a preferred numerical integration technique when accuracy, space and time constraints are important.

# 5 Fourth Order Runge-Kutta Approximation

## 5.1 Problem Statement

The deflection of a rod can be moddeled as:

$$\frac{d^2y}{dz^2} = \frac{f}{2EI}(L-z)^2$$

Where f = force, E = modulus of elasticity, L = rod length, and I = moment of inertia. Programmatically in c++ calculate the deflection if y = 0 and dy/dz = 0 at z = 0. Use f = 60, L = 30, E = $1.25 \times 10^8$ and I = 0.05.

## 5.2 Algorithm Description

The Fourth Order Runge-Kutta Approximation method is used to evaluate first order ordinary differential equations. It evaluates the weighted sum of four estimates of change; the first estimate is Euler's method, the second estimate is the Midpoint method using the first estimate, the third estimate is the Midpoint method using the second estimate, and the fourth estimate is Runge-Kutta 4. The formula can be seen below:

$$\text{First Estimate: } k_1 = f(x_i, y_i)$$

$$\text{Second Estimate: } k_2 = f(x_i + \frac{h}{2}, y_i + \frac{k_1 h}{2})$$

$$\text{Third Estimate: } k_3 = f(x_i + \frac{h}{2}, y_i + \frac{k_2 h}{2})$$

$$\text{Fourth Estimate: } k_4 = f(x_i + h, y_i + k_3 h)$$

$$\text{Weighted Sum: } y_{i+1} = \frac{y_i k_1 + 2k_2 + 2k_3 + k4h}{6}$$

The problem statement describes a second order ordinary differential equation. In order to use the Runge-Kutta method to solve this problem the equation first had to be written in first order form as follows:

$$x = \frac{dy}{dx} \Rightarrow \frac{dx}{dz} = \frac{d^2y}{dz^2} = \frac{f}{2EI}(L-z)^2$$

The Runge-Kutta method was then used to evaluate $x = \frac{dy}{dx}$. This evaluation produced a set of datapoints belonging to x. The Runge-Kutta method was then used again, this time on the datapoints belonging to x to produce $\frac{dx}{dz} = \frac{d^2y}{dz^2} = \frac{f}{2EI}(L-z)^2$, the total deflection of the rod.

## 5.3 Complexity

### 5.3.1 Space Complexity

The space complexity for this program is reliant on the length of the rod and the step size used. A step size of 0.25 and 0.5 were used for $x$ and $\frac{dx}{dz}$ respectively which equates to a total space complexity of $(L + L \times h) + (L + L \times h)$. This can be represented in worst case scenario as $O(L + L \times h)$ where L is the length of the rod and h is the step size used. Note that this complexity assumes the step size is less then one (as it is in this program).

### 5.3.2 Time Complexity

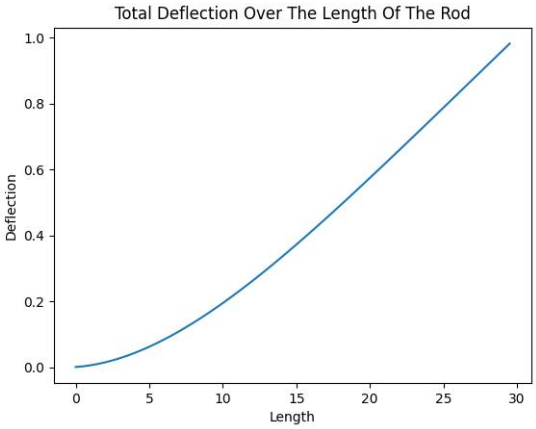
The Runge-Kutta method operates in constant time however it is executed $L + L \times h$ times, where $L$ is the length of the rod and $h$ is the step size. This can be represented in worst case scenario as $O(L + L \times h)$ Note that this complexity assumes the step size is less then one (as it is in this program).

## 5.4 Experiments and Results

For this program a step size of $h = 0.25$ was chosen to evaluate $x = \frac{dy}{dx}$. This gave us twice as many datapoints as $\frac{dx}{dz}$ which used a step size of $h = 0.5$. This two-to-one relationship was chosen due to the nature of the $k_2$ and $k_3$ estimates which divide h by two. Without this relationship, when the second Runge-Kutta method is executed on the datapoints of $x$, $\frac{h}{2}$ may not be a datapoint of $x$ and would therefore be invalid.

Other initial step sizes such as $h = 1$ and $h = 0.1$ were also used in the experimentation process. It was found that $h = 1$ did not produce enough datapoints to adequately model the deflection of the rod. Conversely, $h = 0.1$ produced many datapoints which for this problem did not produce a significantly more accurate answer than an initial value of $h = 0.25$. It can be noted that if the length of the rod was much greater a decreased initial step size of $h = 0.1$ should be considered.

A visual representation of the total deflection of the rod:




Tabular results indicating the deflection at each datapoint:

| Point | Deflection | Point | Deflection | Point | Deflection |
|---|---|---|---|---|---|
| 0 | 0 | 10 | 0.179 | 20 | 0.554 |
| 0.5 | 0.001 | 10.5 | 0.195 | 20.5 | 0.575 |
| 1 | 0.003 | 11 | 0.211 | 21 | 0.596 |
| 1.5 | 0.006 | 11.5 | 0.227 | 21.5 | 0.617 |
| 2 | 0.010 | 12 | 0.244 | 22 | 0.638 |
| 2.5 | 0.015 | 12.5 | 0.262 | 22.5 | 0.659 |
| 3 | 0.21 | 13 | 0.279 | 23 | 0.681 |
| 3.5 | 0.027 | 13.5 | 0.297 | 23.5 | 0.702 |
| 4 | 0.035 | 14 | 0.316 | 24 | 0.723 |
| 4.5 | 0.043 | 14.5 | 0.334 | 24.5 | 0.745 |
| 5 | 0.052 | 15 | 0.353 | 25 | 0.766 |
| 5.5 | 0.062 | 15.5 | 0.372 | 25.5 | 0.788 |
| 6 | 0.073 | 16 | 0.392 | 26 | 0.809 |
| 6.5 | 0.084 | 16.5 | 0.411 | 26.5 | 0.831 |
| 7 | 0.096 | 17 | 0.41 | 27 | 0.853 |
| 7.5 | 0.108 | 17.5 | 0.451 | 27.5 | 0.874 |
| 8 | 0.121 | 18 | 0.471 | 28 | 0.896 |
| 8.5 | 0.135 | 18.5 | 0.492 | 28.5 | 0.917 |
| 9 | 0.149 | 19 | 0.512 | 29 | 0.939 |
| 9.5 | 0.164 | 19.5 | 0.533 | 29 .5 | 0.961 |

The total deflection of the rod at point $30 = 0.98266$.

# 6 Bisection, Newton-Raphson and Secant Methods

## 6.1 Problem Statement

Using a c++ program, compare the performance of the Bisection, Newton-Raphson and Secant methods in estimating the root of $f(x) = e^{-x} - x + x^2$. Start with initial estimates of $x_{-1}$ and $x_0 = 1.0$.

## 6.2 Algorithm Description

In order to compare the three aforementioned methods, each method was first implemented to solve the roots of the function listed above.

In order to use the Bisection Method, an initial interval that is known to contain a root of the function must be used. Upon theoretical inspection of the problem statement, it was noticed that no root existed in the given interval and function. As such, the function and initial estimates were changed (in accordance with the course conveners instructions) so a viable root could be found. The new function used for this question is found below:

$$f(x) = (e^{-x} - x + x^2) - 4, x_0 = -1.0, x_1 = 0.0$$

The Bisection Method operates by systematically reducing the interval of $[x_0, x_1]$ into equal parts based on a midpoint calculation. It then performs a simple test and discards half the function. This process repeats until a root of the function is found. It assumes that the function $f(x)$ is continuous on the limits of $[x_0, x_1]$ and that $f(x_0) \times f(x_1) < 0$. The algorithm can then be described as:

**Algorithm 1** Bisection Method

**loop**
  $MidPoint \leftarrow \frac{x_0+x_1}{2}$
  $f_{MidPoint} \leftarrow f(MidPoint)$
  **if** $f(x_0) \times f_{MidPoint} < 0$ **then**
    $x_1 \leftarrow MidPoint$
  **end if**
  **if** $f(x_0) \times f_{MidPoint} > 0$ **then**
    $x_0 \leftarrow MidPoint$
  **end if**
**end loop**

The Newton-Raphson Method uses an initial guess of the root $x_0$ and information about the function and its derivative at $x_0$ to find a more accurate estimation of the root. It operates under the assumptions that; $f(x)$ is continuous and the first derivative $f'(x)$ is known, and an initial guess $x_0$ such that $f'(x_0) \neq 0$ is supplied.

The method uses a current approximation $x_i$ to find a better approximation $x_{i+1}$ by using the below equation. It will continue to estimate a more accurate guess of the root $x_{i+1}$ until some predefined tolerance is met.

$$f'(x_i) = \frac{f(x_i)}{x_i - x_{i+1}}$$

$$x_{i+1} = x_i - \frac{f(x_i)}{f'(x_i)}$$

The Secant Method uses two initial guesses of the root $x_0$ and $x_1$ to find a more accurate estimation of the root. It operates under the assumption that two initial points are given and that $f(x_0) - f(x_1) \neq 0$.

This final method uses a current approximation $x_{i-1}$ and $x_i$ to find a better approximation $x_{i+1}$ using the below equation. It will continue to estimate a more accurate guess of the root $x_{i+1}$ until some predefined tolerance is met.

$$f'(x_i) = \frac{f(x_i) - f(x_{i-1})}{x_i - x_{i-1}}$$

$$f(x_{i+1}) = f(x_i) + (x_{i+1} - x_i)\frac{f(x_i) - f(x_{i-1})}{x_i - x_{i-1}}$$

A root implies $f(x_{i+1}) = 0$ so,

$$0 = f(x_i) + (x_{i+1} - x_i)\frac{f(x_i) - f(x_{i-1})}{x_i - x_{i-1}}$$

$$x_{i+1}(f(x_i) - f(x_{i-1})) = x_i(f(x_i) - f(x_{i-1})) - f(x_i)(x_i - x_{i-1})$$

$$x_{i+1} = x_i - \frac{f(x_i)(x_i - x_{i-1})}{f(x_i) - f(x_{i-1})}$$

## 6.3 Complexity

### 6.3.1 Space Complexity

Each of the three algorithms simply hold the current best root approximation before returning the result. Individually, their total space complexity can be described as $O(1)$. However, the Newton-Raphson Method uses the Richardson Extrapolation technique to calculate the derivative. As seen in section 3.3.1 this function operates in $O(n^2)$ space where n is the size of the matrix used. As previously derived, $n = 3$, so a more accurate space complexity for this function and in turn the entirety of this program can be described as $\Theta(1)$.

### 6.3.2 Time Complexity

Each three methods execute until some predefined tolerance is met. As such, each program operates in $O(n)$ time where n is the amount of iterations used in order to meet the predefined tolerance. This gives a total running time of $O(n)$ for this program.

## 6.4 Experiments and Results

As previously mentioned, each algorithm operates until some tolerance is met. The following tolerance values were used as experimental data in order to gauge how accurately each algorithm estimated the root of the given function and how many iterations were needed for that estimation.

| Method | Tolerance | Estimate | Iterations |
|---|---|---|---|
| Bisection | 5% | -0.84375 | 5 |
| Newton | 5% | -0.874389 | 1 |
| Secant | 5% | -0.86653 | 3 |

| Method | Tolerance | Estimate | Iterations |
|---|---|---|---|
| Bisection | 1% | -0.868164 | 10 |
| Newton | 1% | -0.867325 | 2 |
| Secant | 1% | -0.867293 | 4 |

| Method | Tolerance | Estimate | Iterations |
|---|---|---|---|
| Bisection | 0.1% | -0.867249 | 14 |
| Newton | 0.1% | -0.867303 | 3 |
| Secant | 0.1% | -0.867303 | 5 |

| Method | Tolerance | Estimate | Iterations |
|---|---|---|---|
| Bisection | 0.01% | -0.867302 | 17 |
| Newton | 0.01% | -0.867303 | 3 |
| Secant | 0.01% | -0.867303 | 5 |

| Method | Tolerance | Estimate | Iterations |
|---|---|---|---|
| Bisection | 0.001% | -0.867303 | 20 |
| Newton | 0.001% | -0.867303 | 3 |
| Secant | 0.001% | -0.867303 | 5 |

It can be seen that both the Newton-Raphson and Secant methods start to diverge on the root at around 0.1% tolerance whereas the Bisection method starts to diverge at 0.01% tolerance. It is interesting to note that even at a 5% tolerance the Secant method is quite close to the analytical root value compared to the other two methods.

The Bisection method can be seen as a reliable however slow method to compute the root of a given function. This method only computes one function evaluation per iteration, meaning each iteration is more efficient then the Newton-Raphson method and on par with that of the Secant method. However, as can be seen above, the amount of iterations needed is much higher making this observation obsolete.

The Newton-Raphson method is extremely fast when compared to the other two methods. It evaluated the root in less iterations at every test case used and diverged upon the root at a reasonable tolerance. This method does however evaluate two functions at each iteration and needs the value of the derivative to do so.

The Secant method is also fast at computing the Root. It diverged towards the root at the same tolerance values of the Newton-Raphson method, however requiring slightly more iterations to do so. This method only computes one function evaluation at each iteration and does not require any knowledge of the derivative to do so.

To conclude, the Bisection method should not be used as a preferred method when calculating the root. Any time the Bisection method can be used, the Secant method can also be used. As seen above, the Secant method is both more accurate and faster than the Bisection method, meaning it should be used in place of the Bisection method any time accuracy and time complexity are important. Although the Newton-Raphson method performed quicker and just as accurate as the Secant method it evaluates two functions upon each iteration instead of one. For the general case where a function is quite large and tedious to compute, this methods complexity would be higher than the of the Secant method even though the iterations needed will be lower. This method also requires knowledge of the derivative which can be passed directly in the case of trivial functions. However, in the general case where the derivative is not known an estimation function would need to be used, again, adding to the Newton-Raphson method complexity. The Newton-Raphson Method does however only require one initial estimate to operate compared to the two needed by the Secant method. As such, it is recommended that any time a function is given with two estimates such that $f(x_0) - f(x_1) \neq 0$ the Secant method should be used. In all other cases where a function is given with only one estimate the Newton-Raphson method should be used.