3801ICT
Numerical Algorithms
Assignment 1

Candice Smith
S5168666

Question 1

The given problem is to use the centred difference approximation to estimate the first derivative of the function $f(x) = -0.1x^4 - 0.15x^3 - 0.5x^2 - 0.15x + 1.2$. The centred difference approximation with round off errors is defined as:

$$f(x_i)' = \frac{f(x_{i-1}) - f(x_{i+1})}{2h} + \frac{e_{i+1} - e_{x-1}}{2h} - \frac{f^{(3)}(\xi)}{6}h^2$$

$$True\ Value = Finite\ Difference\ Approx. + Round\ off\ error - Truncation\ error$$

Given this formula, it implies that there exists an optimal value for h (h-opt) which will result in the true value. To find h-opt, varying values for h were used from 1.0 to -1.0 and when the error is zero the h-opt value is found. The given problem proposes the follow formula for h-opt.

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

$$Where\ \varepsilon = machine\ epsilon$$

$$and\ M = f(\varepsilon)'''$$

Plugging the values in for 16-bit, 32-bit and 64-bit computers gives a h-opt value of:

16-bit h-opt = -0.143066

32-bit h-opt = -0.00709813

64-bit h-opt = $-8.73348 * 10^{-6}$

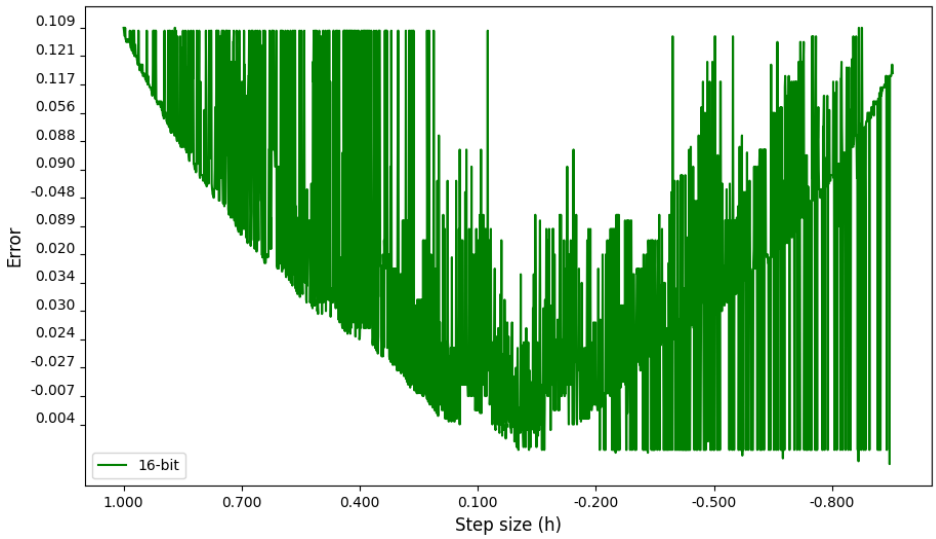
There is quite a lot of noise in the 16-bit graphs but there is still a curved trend; as h approaches zero the total error decreases. It is more evident in the 32-bit and 64-bit graphs, due to less noise, at exactly zero the error increases again. This shows that h-opt is very close to zero but not exactly zero. This trend is reflected in the calculated values for h-opt which are very small but not zero. The h-opt decreases from 16-bit to 32-bit to 64-bit, which shows that 64-bit is the most accurate (produces the closest estimate to the true value) and has the smallest round-off and truncation error.

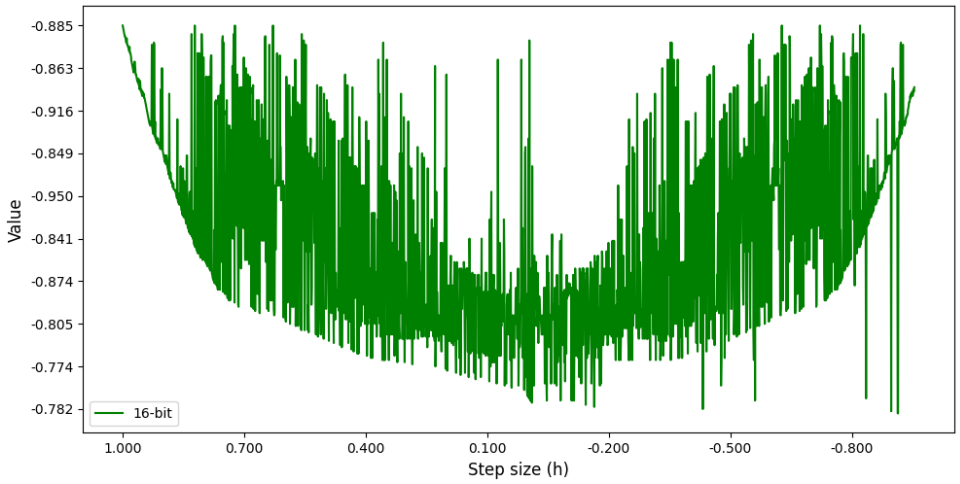
16-bit	
When h ≅ -0.143066 (16-bit h-opt)	$f(x)' = -0.778809$; Error = 0.038788
When $f(x)' \cong -0.8125$ (true value)	h = -0.050079; Error = 0.004513
32-bit	
When h ≅ -0.00709813 (32-bit h-opt)	$f(x)' = -0.660480$; Error = 0.000005
When $f(x)' \cong -0.8125$ (true value)	h = -0.757984; Error = 0.049567

64-bit	
When h ≅ -8.73348 * 10^{-6} (64-bit h-opt)	$f(x)'$ = -0.660471; Error = 0.000000
When $f(x)' \cong -0.8125$ (true value)	h = -0.758000; Error = 0.049566

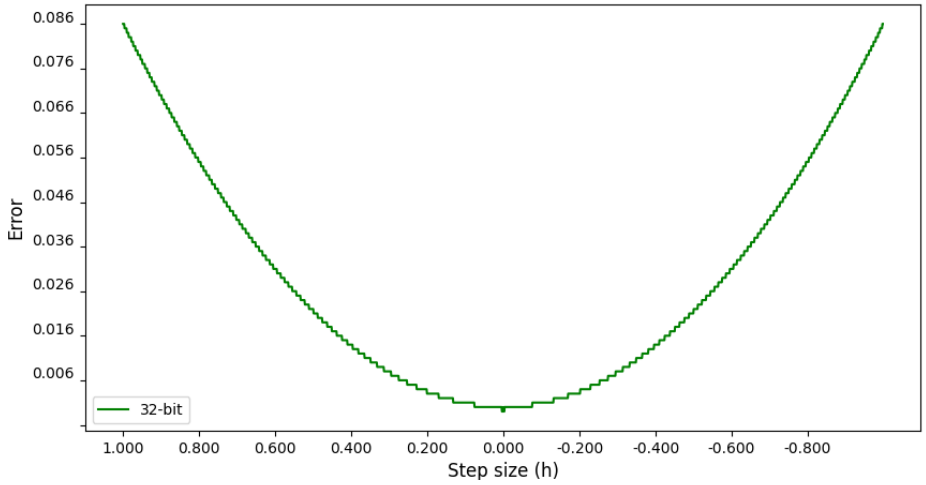
Results:

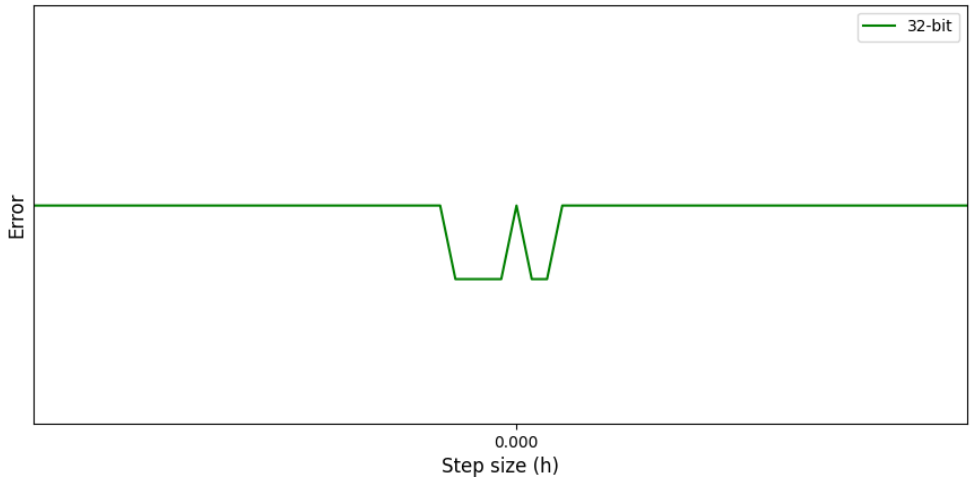
16-bit

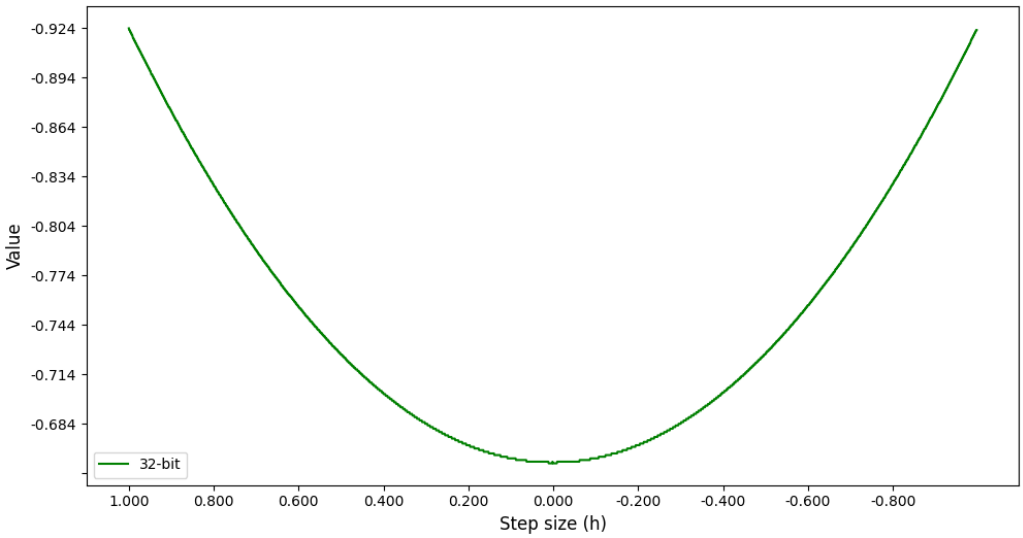




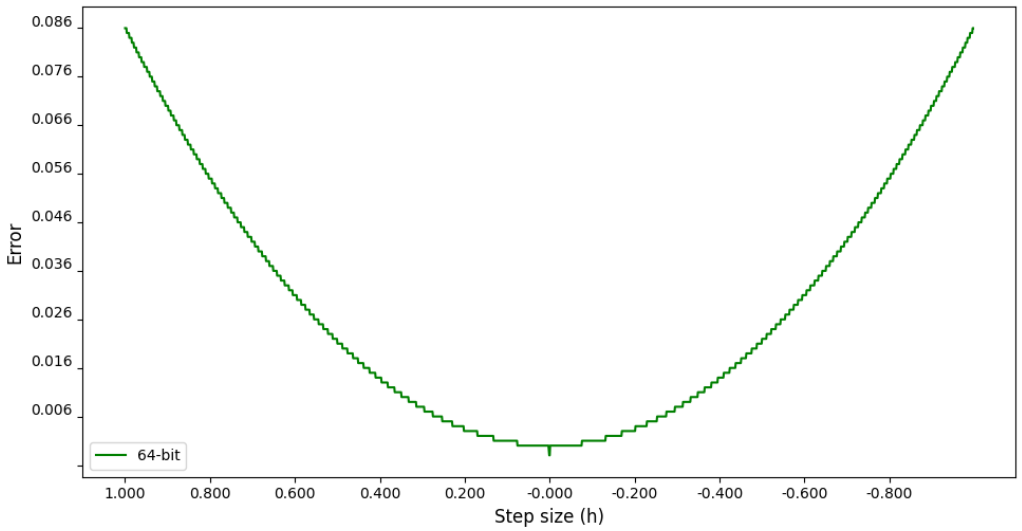
32-bit

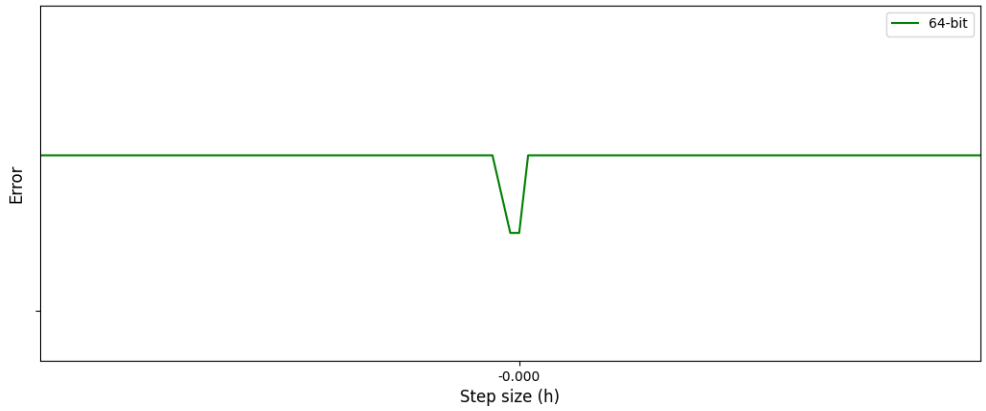


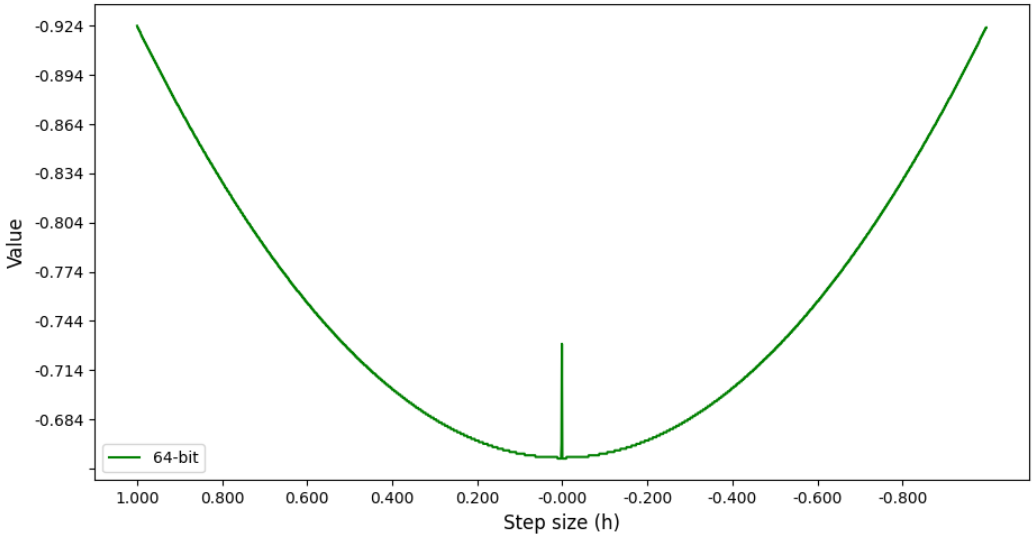

32-bit
Error
0.000
Step size (h)


-0.924
-0.894
-0.864
-0.834
-0.804
-0.774
-0.744
-0.714
-0.684
Value
32-bit
1.000
0.800
0.600
0.400
0.200
0.000
-0.200
-0.400
-0.600
-0.800
Step size (h)

64-bit


0.086
0.076
0.066
0.056
0.046
0.036
0.026
0.016
0.006
Error
64-bit
1.000
0.800
0.600
0.400
0.200
-0.000
-0.200
-0.400
-0.600
-0.800
Step size (h)





Question 2

T, s	200	202	204	206	208	210
θ, rad	0.75	0.72	0.70	0.68	0.67	0.66
R, m	5120	5370	5560	5800	6030	6240

The problem given was to calculate, using the values in the table above, the velocity and acceleration in vector form of an aircraft at the given intervals of time. To calculate the velocity and acceleration the position of the aircraft at each time interval is needed. The coordinates of the position need to be calculated. This is done using $x = R \times \cos(\theta)$ and $y = R \times \sin(\theta)$ where $R =$ distance from radar in meters and $\theta =$ sweep angle of radar.

The coordinates were then used to calculate the velocity of each time interval using the first order derivative.

$$vx = |(x_{i+1} - x_{i-1})/2h|$$

$$vy = |(y_{i+1} - y_{i-1})/2h|$$

The acceleration was calculated for each time interval using the second order derivative.

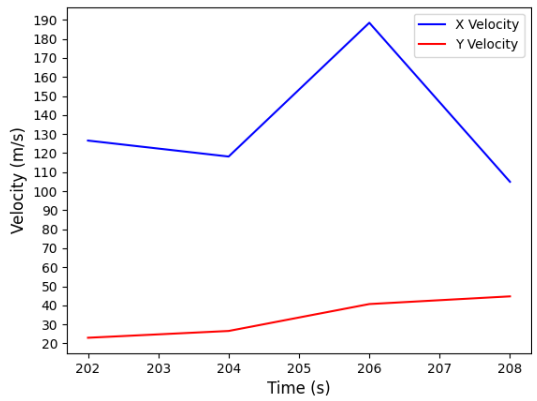
$$ax = (x_{i+1} - 2x_i + x_{i-1})/h^2$$

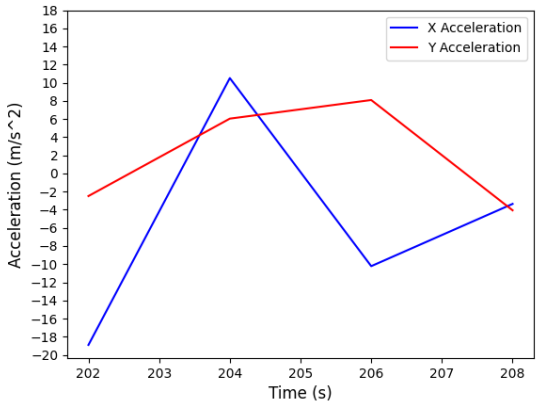
$$ay = (y_{i+1} - 2y_i + y_{i-1})/h^2$$

The x_i and y_i values represent the x and y coordinates at each time interval. The h value represents the step size of each interval. This is the difference between time intervals which is always 2.

Results:

Time (sec)	x Coordinate	y Coordinate	Velocity x	Velocity y	Acceleration x	Acceleration y
200	3746.25	3489.99	-	-	-	-
202	4037.2	3540.9	126.56889	22.964972	-18.90599	-2.4876462
204	4252.5226	3581.8503	118.18125	26.525963	10.518353	6.0486364
206	4509.9218	3646.9995	118.48052	40.67379	-10.219083	8.0991911
208	4726.446	3744.5455	104.90744	44.712404	-3.3539993	-4.060577
210	4929.5515	3825.8492	-	-	-	-





The X Velocity is the change in speed in the x direction (forward movement). The Y Velocity is the change in speed in the y direction (up movement). The X Acceleration is the change in velocity in the x direction. The Y Acceleration is the change in velocity in the y direction.

Question 3

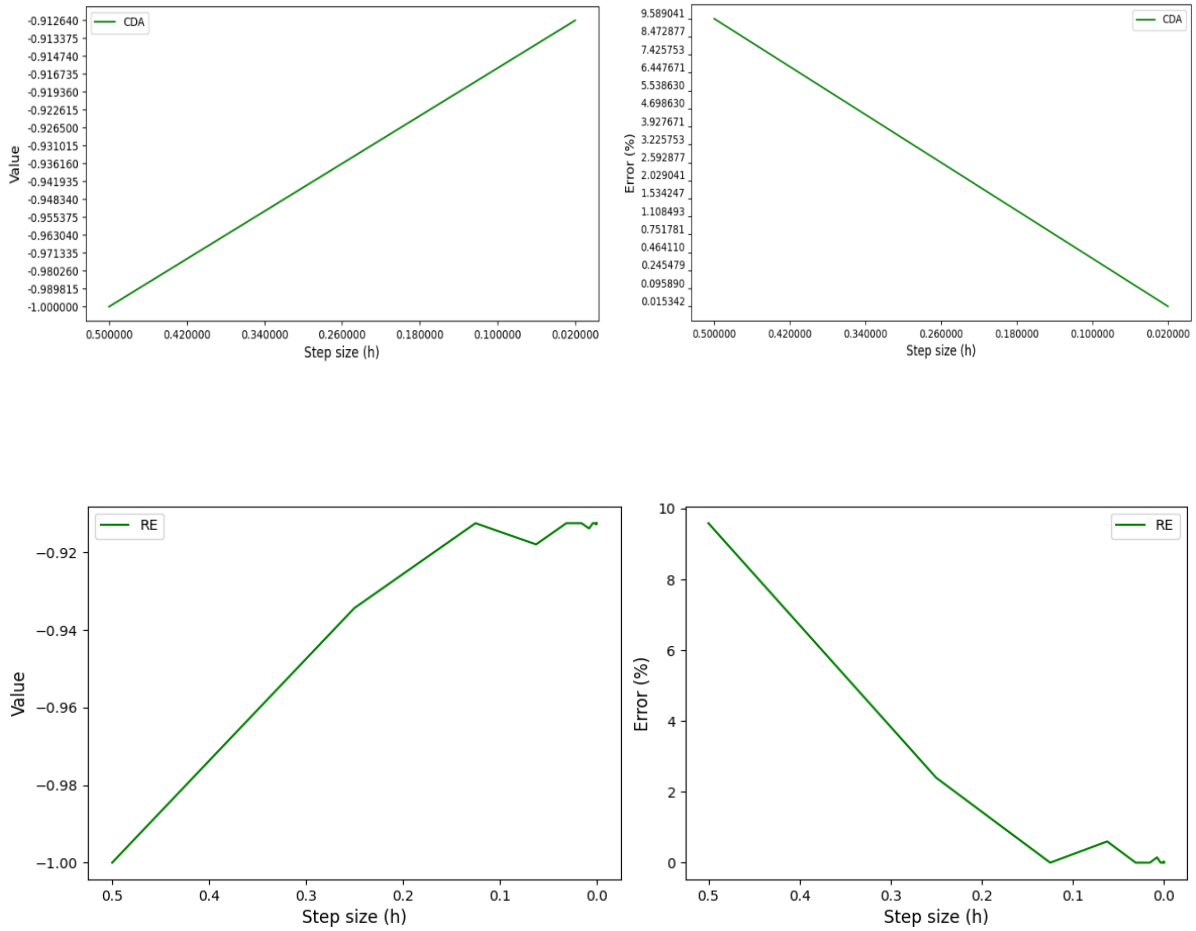
The problem given was to compare the Central Difference Approximation (CDA) and Richardson Extrapolation (RE) methods for finding the first derivative of $f(x) = -0.1x^4 - 0.15x^3 - 0.5x^2 - 0.25x + 1.2$ at $x = 0.5$. The CDA method uses the formula $f(x)' = \frac{f(x+h)-f(x-h)}{2h}$. An initial h value of 0.5 was used then it was decreased by 0.01 until h = 0.01. The RE method uses the same formula as CDA for finding $f(x)'$, however it has an additional extrapolation method to move closer to the real value. The values were recorded and compared to the real value of $f(x)'$ to test the effectiveness of each of the methods.

Pseudocode:

RE ← array of extrapolated values
h ← 0.5
procedure RICHARDSON-EXTRAPOLATION(RE, h)
    for i ← 0 to i < 5 **do**
        RE[i][0] ← $\frac{f(x+h)-f(x-h)}{2h}$
        for j ←1 to j ≤ i **do**
            RE[i][j] ← RE[i][j – 1] + (RE[i][j – 1] – RE[i – 1][j – 1]) / j⁴ – 1

```
            end for
            h ← h / 2
        end for
end procedure
```

Results:



The shape in the graphs for both methods show a similar trend in the value and error as h decreases. As h decreases the value calculated decreases, moving closer to the real value of -0.9125. Conversely as h decreases the error decreases, which shows that it is moving closer to the real value.

Question 4

The given problem was using Romberg Integration find the area of the cross-section of a river using the measurements in the table below. Romberg Integration must be used with a stopping criterion of 1%.

X, m	0	2	4	6	8	10	12	14	16
H, m	0	1.9	2	2	2.4	2.6	2.25	1.12	0

Pseudocode:

```
RI ← array of interpolated values
h ← 0.5
n ← 7
acc ← 1.0
procedure ROMBERG-INTEGRATION(RI, h)
        RI[0][0] ← [f(a) + 2 ∑_{i=1}^{n−1} f(a + ih) + f(b)] h/2
        for i ← 1 to i < n do
                for j ←1 to j ≤ i do
                        if j = 1 do
                                RI[i][j] ← [f(a) + 2 ∑_{i=1}^{n−1} f(a + ih) + f(b)] h/2
                                h ← h / 2
                        else
                                RI[i][j] ← RI[i][j – 1] + (RI[i][j – 1] – RI[i – 1][j – 1]) / (j – 1)⁴ – 1
                                if |(RI[i][j] - RI[i][j - 1]) / RI[i][j])| * 100 < acc do
                                        return RI[i][j]
                        end if
                end for
        end for
end procedure
```

Results:

0.000000		
19.200000	25.600000	
26.600000	29.066667	29.297778

When stopping criteria = 1% Romberg Integration = 29.297778m^2.

0.000000				
19.200000	25.600000			
26.600000	29.066667	29.297778		
28.540000	29.186667	29.194667	29.193030	
28.540000	28.540000	28.496889	28.485813	28.483040

When stopping criteria = 0% Romberg Integration = 28.54m^2.

Calculating the integral by hand gives a total area of 28.54m^2. The implementation of Romberg Integration used the composite trapezoidal rule to approximate the integral of each iteration. The first iteration approximates the area of one segment from a to b (see Figure 1). The second iteration approximates 2 segments from a to b/2 and from b/2 to b. It continues in this fashion until it reaches a stopping criterion, or the value cannot be interpolated anymore. Overall Romberg Integration found the real value of the integral and came close to it with a stopping criterion of 1%.

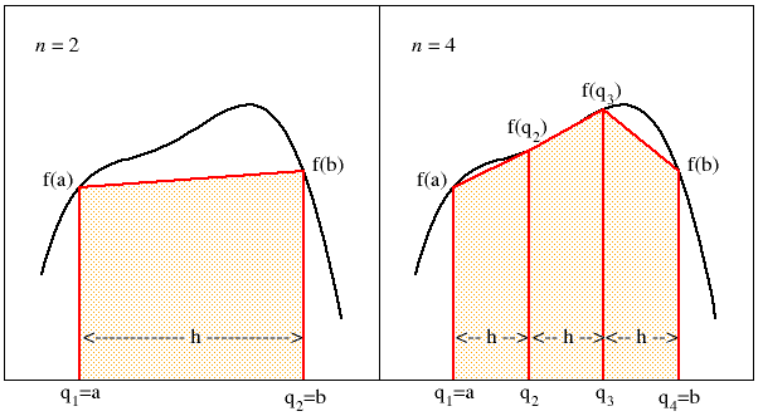


Figure 1. Trapezoidal Rule (www.pamoc.it/tpc_num_int.html)

Question 6

The problem given was to compare the performance of the Bisection, Newton-Raphson and Secant methods in estimating the root of $f(x) = e^{-x} - x$ starting with the initial values of $x_{-1} = 0$ and $x_0 = 1.0$.

Pseudocode:

```
x0 ← 1.0
x1 ← 0
procedure BISECTION(x0, x1)
        u ← e^(-x1) − x1
        v ← e^(-x0) − x0
        for i ← 0 to i < 20 do
                c ← (x1−x0)/2
                fc ← e^(-c) − c
                if u * fc < 0 do
                        b ← c
                        v ← fc
                else
                        a ← c
                        u ← fc
                end if
        end for
        return |fc|
end procedure
```

```
x0 ← 1.0
x1 ← 0
procedure NEWTON-RAPHSON(x1)
        do while |fz| > 0.00001
                z ← (x1−e^(-x1)−x1)/(−e^(-x1)−1)
                fz ← e^(-z) − z
                x1 ← z
        end do while
        return z
end procedure
```

```
x0 ← 1.0
x1 ← 0
procedure SECANT(x0, x1)
        x2 ← 0
        fx1 ← e^(−x1) − x1
        fx0 ← e^(−x0) − x0
        while |x0 – x1| ≥ 0.00001 do
                x2 ← (fx0×x1−fx1×x0) / (fx0−fx1)
                x1 ← x0
                fx1 ← fx0
                x0 ← x2
                fx0 ← e^(−x0) − x0
        end while
        return x2
end procedure
```

All the methods found the same answer. The difference is how many iterations it took for them to find it. The Bisection method took 20 iterations, Newton-Raphson method took 3 iterations and Secant method found it in 4 iterations. Even though Newton-Raphson's method is the fastest, it does require the knowledge of the derivative of $f(x)$. Secant does not require the derivative and produced the same result as Newtons in only 2 more iterations. Bisection method is overall the slowest, but it still produces a correct result.

Results:

Newton-Raphson Method		
i	**Root**	**Error**
0	0.500000	0.106531
1	0.566311	0.001305
2	0.567143	0.000000

Secant Method		
i	**Root**	**Error**
0	0.567143	0.070814
1	0.563838	0.005182
2	0.567170	0.000042
3	0.567143	0.000000

Bisection Method		
i	**Root**	**Error**
0	0.500000	0.106531
1	0.750000	0.277633
2	0.625000	0.089739
3	0.562500	0.007283
4	0.593750	0.041498
5	0.578125	0.017176
6	0.570312	0.004964
7	0.566406	0.001155
8	0.568359	0.001905
9	0.567383	0.000375
10	0.566895	0.000390
11	0.567139	0.000007
12	0.567261	0.000184
13	0.567200	0.000088
14	0.567169	0.000041
15	0.567154	0.000017
16	0.567146	0.000005
17	0.567142	0.000001
18	0.567144	0.000002
19	0.567143	0.000000