
# Laboratory Report

*Martin Westberg*

*Gustav Sundkvist Olsson*

*Nicolas Kuiper*

*Group 9*

*Numerical Methods with MATLAB, VT 2022*

*Course code: MAA042*

*Instructor: Maksat Ashyralyyev*

# INDEX

# 1. NUMERICAL DIFFERENTIATION

The second and fourth order central difference formulas approximating $f'(x_0)$ are given by

$$(D_0^2 f)(x_0; h) = \frac{f(x_0+h)-f(x_0-h)}{2h} \quad (1)$$

and

$$(D_0^4 f)(x_0; h) = \frac{-f(x_0+2h)+8f(x_0+h)-8f(x_0-h)+f(x_0-2h)}{12h} \quad (2)$$

respectively.

1. Show that the truncation errors in the approximations (1) and (2) are $O(h^2)$ and $O(h^4)$, respectively.
2. Consider the function $f(x) = x^{x^x}$. Approximate $f'(1.2)$ by using the central difference formulas (1) and (2) with $h = h_k = 2^{-k}$, $k = 1, 2, \dots, 15$. Calculate the absolute errors in these two approximations for each step size $h_k = 2^{-k}$, $k = 1, 2, \dots, 15$ and then plot in one frame the errors against the stepsizes $\{h_k\}_{k=1}^{16}$ in log-log scale. What do you observe from the plots? Does this confirm the theory for order of convergence? If not then explain why.

## 1.1 Solution

The Taylor's series of a function $f(x)$ around the point $x_0$ is given by:

$$f(x) = \sum_{n=0}^{n} \left( \frac{f^{(n)}(x_0)}{n!}(x - x_0)^n \right) + \frac{f^{(n+1)}(\xi)}{(n+1)!}(x - x_0)^{n+1}, \; x < \xi < x_0 \quad (3)$$

Where the first term on the right side is the *n*th Taylor polynomial approximation for the function $f(x)$ and the second term is the error introduced by the use of such approximations.

Assuming that $f(x) \in C^{n+1}$ in an interval $[a, b]$, the second order central difference formula given by equation (1) can be derived from Taylor's polynomial of second order (i.e. $n = 2$) by inserting $x = x_0 + h \Leftrightarrow h = x - x_0$ in (3) to obtain:

$$f(x) = f(x_0 + h) = \sum_{n=0}^{2} \frac{f^n(x_0)}{n!}h^n = f(x_0) + f'(x_0)h + \frac{f''(x_0)}{2!}h^2 + \frac{f^{(3)}(\xi_1)}{3!}h^3$$

Solving for $f'(x_0)$ we get:

$$f'(x_0) = \frac{f(x_0+h) - f(x_0)}{h} - \frac{f'(x_0)}{2}h - \frac{f^{(3)}(\xi_1)}{6}h^2,\ x_0 < \xi_1 < x_0 + h \quad (4)$$

Similarly, by inserting $x = x_0 - h \Leftrightarrow -h = x - x_0$ in (3) we get:

$$f(x_0 - h) = f(x_0) - f'(x_0)h + \frac{f''(x_0)}{2!}h^2 - \frac{f^{(3)}(\xi_2)}{3!}h^3$$

And solving for $f'(x_0)$ yields:

$$f'(x_0) = \frac{f(x_0) - f(x_0-h)}{h} + \frac{f''(x_0)}{2}h - \frac{f^{(3)}(\xi_2)}{6}h^2,\ x_0 - h < \xi_2 < x_0 \quad (5)$$

Adding equations (4) and (5) gives:

$$2f'(x_0) = \frac{f(x_0+h) - f(x_0)}{h} - \frac{f''(x_0)}{2}h - \frac{f^{(3)}(\xi_1)}{6}h^2 + \frac{f(x_0) - f(x_0-h)}{h} + \frac{f''(x_0)}{2}h - \frac{f^{(3)}(\xi_2)}{6}h^2$$

$$\Rightarrow f'(x_0) = \frac{f(x_0+h)-f(x_0-h)}{2h} - \frac{(f^{(3)}(\xi_1)+f^{(3)}(\xi_2))}{12}h^2 \quad (6)$$

Since we assumed $f(x) \in C^{n+1}$ in an interval $[a, b]$, we can say that $f^{(3)}(x)$ is continuous in $[a, b]$ and therefore also in $[\xi_1, \xi_2]$, so there must exists a number $\xi \in [\xi_1, \xi_2]$ such that:

$$f^{(3)}(\xi) = \frac{f^{(3)}(\xi_1)+f^{(3)}(\xi_2)}{2}$$

Plugging this result in equation (6) gives:

$$f'(x_0) = \frac{f(x_0+h)-f(x_0-h)}{2h} - \frac{f^{(3)}(\xi)}{6}h^2 \Leftrightarrow \frac{f(x_0+h)-f(x_0-h)}{2h} + O(h^2)$$

We have therefore derived the *second order central difference formula* and proven that the truncation error is of order $O(h^2)$.

In a similar manner, the fourth order central difference formula given by equation (2) can be derived from Taylor's polynomial of fourth order (i.e. $n = 4$) by inserting $x = x_0 + h \Leftrightarrow h = x - x_0$ and $x = x_0 - h \Leftrightarrow -h = x - x_0$ in (3) to obtain:

$$f'(x_0) = \frac{f(x_0+h)-f(x_0)}{h} - \frac{f''(x_0)}{2}h - \frac{f^{(3)}(x_0)}{6}h^2 - \frac{f^{(4)}(x_0)}{24}h^3 - \frac{f^{(5)}(\xi_1)}{120}h^4 \quad (7)$$

$$f'(x_0) = \frac{f(x_0)-f(x_0-h)}{h} + \frac{f''(x_0)}{2}h - \frac{f^{(3)}(x_0)}{6}h^2 + \frac{f^{(4)}(x_0)}{24}h^3 - \frac{f^{(5)}(\xi_2)}{120}h^4 \quad (8)$$

Adding (7) and (8) results in:

$$2f'(x_0) = \frac{f(x_0+h)-f(x_0-h)}{h} - \frac{f'''(x_0)}{3}h^2 - \frac{(f^{(5)}(\xi_1)+f^{(5)}(\xi_2))}{120}h^4$$

$$\Rightarrow f'(x_0) = \frac{f(x_0+h)-f(x_0-h)}{2h} - \frac{f'''(x_0)}{6}h^2 - \frac{(f^{(5)}(\xi_1)+f^{(5)}(\xi_2))}{240}h^4$$

And since we assumed $f(x) \in C^{n+1}$ in an interval $[a, b]$, we can say that $f^{(5)}(x)$ is continuous in $[a, b]$ and also in $[\xi_1, \xi_2]$, so there must exists a number $\xi \in [\xi_1, \xi_2]$ such that:

$$f^{(5)}(\xi) = \frac{f^{(5)}(\xi_1)+f^{(5)}(\xi_2)}{2}$$

and plugging this back we obtain:

$$f'(x_0) = \frac{f(x_0+h)-f(x_0-h)}{2h} - \frac{f'''(x_0)}{6}h^2 - \frac{f^{(5)}(\xi)}{120}h^4 \quad (9)$$

If we take step size $2h$ in equation (9) we get:

$$f'(x_0) = \frac{f(x_0+2h)-f(x_0-2h)}{4h} - \frac{4f'''(x_0)}{6}h^2 - \frac{16f^{(5)}(\xi)}{120}h^4 \quad (10)$$

Taking $4 * (9) - (10)$:

$$3f'(x_0) = \frac{4f(x_0+h)-4f(x_0-h)}{2h} - \frac{4f'''(x_0)}{6}h^2 - \frac{4f^{(5)}(\xi)}{120}h^4 - \frac{f(x_0+2h)-f(x_0-2h)}{4h} + \frac{4f'''(x_0)}{6}h^2 + \frac{16f^{(5)}(\xi)}{120}h^4$$

$$\Rightarrow 3f'(x_0) = \frac{-f(x_0+2h)+8f(x_0+h)-8f(x_0-h)+f(x_0-2h)}{4h} + \frac{f^{(5)}(\xi)}{10}h^4$$

$$\Rightarrow f'(x_0) = \frac{-f(x_0+2h)+8f(x_0+h)-8f(x_0-h)+f(x_0-2h)}{12h} + \frac{f^{(5)}(\xi)}{30}h^4$$

$$\Rightarrow f'(x_0) = \frac{-f(x_0+2h)+8f(x_0+h)-8f(x_0-h)+f(x_0-2h)}{12h} + O(h^4)$$

And we have derived the *fourth order central difference formula* while proving that the truncation error is of order $O(h^4)$.

## 1.2 Solution with MATLAB Algorithm

In order to calculate the absolute errors in the approximation of $f'(1.2)$ for $f(x) = x^{x^x}$ when using second and fourth central order difference formulas, we first need to know the actual value for $f'(1.2)$. We can obtain an expanded form for $f'(x) = \frac{df(x)}{dx}$ from Wolfram Alpha and evaluate it for $x = 1.2$. We then obtain:

$$f'(x) = \frac{df(x)}{dx} = x^{x^x}(x^x ln(x)(ln(x) + 1) + x^{x-1})$$

$$\Rightarrow f'(1.2) = 1.2^{1.2^{1.2}}(1.2^{1.2} ln(1.2)(ln(1.2) + 1) + 1.2^{0.2}) \approx 1.637932983825062173566$$

From the second order central difference formula with $x_0 = 1.2$ we have:

$$f'(1.2) \approx \frac{f(1.2+h)-f(1.2-h)}{2h} = \frac{(1.2+h)^{(1.2+h)^{(1.2+h)}} - (1.2-h)^{(1.2-h)^{(1.2-h)}}}{2h}$$

and for the fourth central difference formula with $x_0 = 1.2$ we have:

$$f'(1.2) \approx \frac{-f(1.2+2h)+8f(1.2+h)-8f(1.2-h)+f(1.2-2h)}{12h}$$

$$\Rightarrow f'(1.2) \approx \frac{-(1.2+2h)^{(1.2+2h)^{(1.2+2h)}} + 8(1.2+h)^{(1.2+h)^{(1.2+h)}} - 8(1.2-h)^{(1.2-h)^{(1.2-h)}} + (1.2-2h)^{(1.2-2h)^{(1.2-2h)}}}{12h}$$

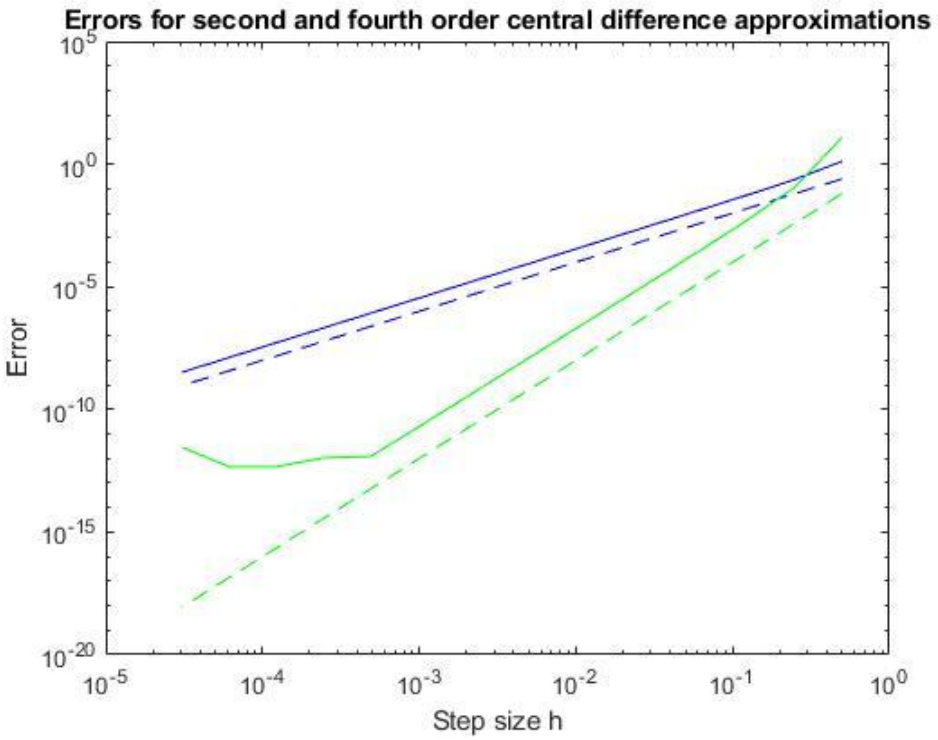
Let us define absolute error as follows:

$$Absolute\ Error = |Actual\ value - Approximated\ value|$$

We can now proceed on writing a MATLAB algorithm to calculate and illustrate the absolute errors when utilizing these two approximations with step sizes $h_k = 2^{-k}$ for $k = 1, 2, ..., 15$; that is:

```
k=linspace(1,15,15);
h=2.^(-k);
CDF2=(((1.2+h).^((1.2+h).^(1.2+h)))-((1.2-h).^((1.2-h).^(1.2-h)))))./(2*h);
CDF4=(-(1.2+2.*h).^((1.2+2.*h).^(1.2+2.*h))+8*(1.2+h).^((1.2+h).^(1.2+h))-8*(1.2-h).^((1.2-h).^(
  1.2-h))+(1.2-2.*h).^((1.2-2.*h).^(1.2-2.*h)))./(12*h);
Correctvalue=1.637932983825062173566;
error1=abs(Correctvalue-CDF2);
error2=abs(Correctvalue-CDF4);
loglog(h,error1,'b',h,h.^2,'b--',h,error2,'g',h,h.^4,'g--')
title('Errors for second and fourth order central difference approximations')
xlabel('Step size h')
ylabel('Error')
```

**ALGORITHM OUTPUT:**



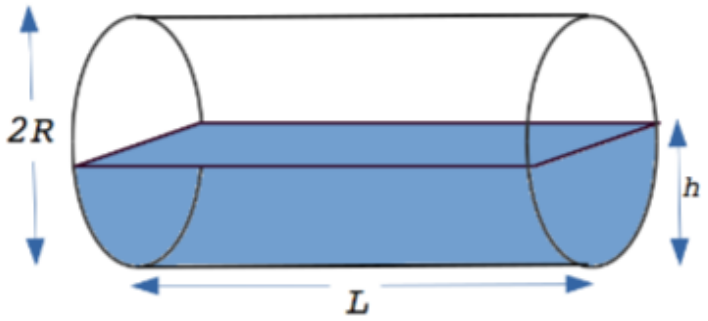

## Analysis of results

The plotted graph above shows the error for the second and fourth order central difference formula (in blue and green respectively) when approximating $f'(1.2)$ for $f(x) = x^{x^x}$. Since the theory remarks that the error for each approximation should be of the same order as the corresponding central difference formula (i.e. $O(h^2)$ for second order central difference and $O(h^4)$ for fourth-), we should expect for the slope of each of the solid lines to match the slope of the same-colored dotted line, which is indeed what is shown in the graph. In the case of the green curve (i.e. fourth order central difference formula), the slope changes drastically for smaller step sizes of $h$ due to an existing trade-off between truncation and rounding errors. This means that too small of a step size leads to a larger rounding error and, conversely, having a small rounding error would be achieved as a consequence of taking a slightly bigger step size.

## 2. SOLVING NONLINEAR ALGEBRAIC EQUATION

The volume of liquid $V$ in a hollow horizontal cylinder of radius $R$ and length $L$ is related to the depth of the liquid $h$ by the formula:

$$V = \left[R^2 \cdot arccos\left(\frac{R-h}{R}\right) - (R - h) \cdot \sqrt{2Rh - h^2}\right] \cdot L \qquad (1)$$




Consider the problem of finding the depth $h$ if $R = 3m$, $L = 8m$ and $V = 100m^3$. It results in a nonlinear algebraic equation with the root in the interval $[0, 6]$.

1. Illustrate graphically that the resulting equation has a unique root in the interval $[0, 6]$.
2. Use the Bisection method to approximate the depth $h$ within an absolute accuracy of $10^{-15}$. How many Bisection iterations are required to achieve the approximation within an accuracy of $10^{-15}$? Store this numerical result. It can be considered as an exact (reference) solution.
3. Use Newton-Raphson method with initial value $p_0 = 1$ to approximate the depth $h$ within a relative precision of $10^{-14}$. How many Newton-Raphson steps are needed to obtain the result? Compare your numerical result with the reference solution obtained from the Bisection method.
4. Use Secant method with initial values $p_0 = 0$ and $p_1 = 6$ to approximate the depth $h$ within a relative precision of $10^{-14}$. How many steps does the Secant method need to obtain the result? Compare your numerical result with the reference solution obtained from the Bisection method.
5. Compare the computational costs of three methods used for our root finding problem.

## 2.1 Graphic Method

Variables $R = 3m$, $L = 8m$ and $V = 100$ are given, so that the only unknown variable is $h$, and from equation (1) we can define a function $F(h)$ so that:

$$F(h) = V \tag{2}$$

Let us now consider a nonlinear algebraic equation:

$$f(h) = F(h) - V = 0$$

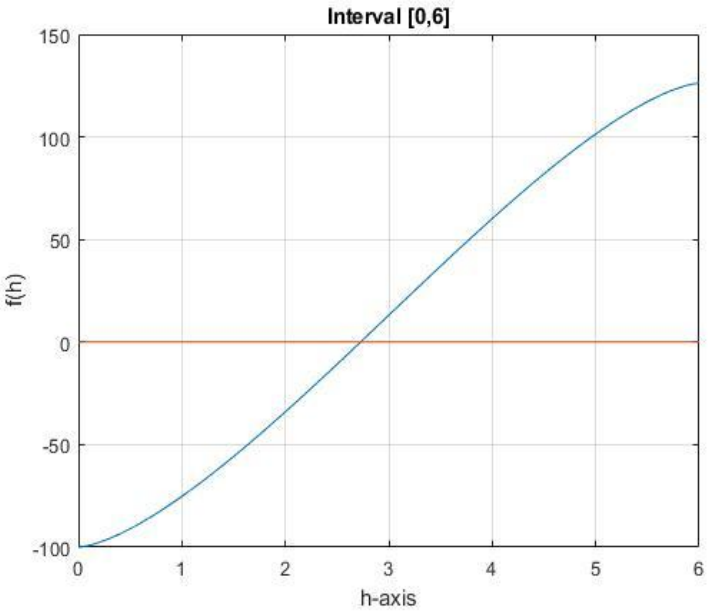
$$\Rightarrow f(h) = \left[R^2 \cdot arccos\left(\frac{R-h}{R}\right) - (R - h) \cdot \sqrt{2Rh - h^2}\right] \cdot L - V = 0$$

And replacing for the values of $R$, $L$ and $V$ we obtain:

$$\Rightarrow f(h) = \left[9 \cdot arccos\left(\frac{3-h}{3}\right) - (3 - h) \cdot \sqrt{6h - h^2}\right] \cdot 8 - 100 = 0 \tag{3}$$

We can illustrate this by writing the MATLAB algorithm below to plot equation (3) for $h \in [0, 6]$; this is:

```
clc; clear
x=linspace(0,6,100);
R=3; L=8; V=100;
f=(R.^2*acos((R-x)./R)-(R-x).*sqrt(2.*R.*x-x.^2)).*L-V;
y=0*x;
plot(x,f,x,y)
grid on
title('Interval [0,6]')
xlabel('h-axis')
ylabel('f(h)')
```




From the algorithm's output on the right, it is clear that the equation $f(h) = 0$ holds for only one value of $h$, meaning that the equation has a unique root in the interval $[0, 6]$.

## 2.2 Bisection Method

We know from the previous solution that equation (3) has a unique root in the interval $[0, 6]$. We can conclude this too by the means of the *Intermediate Value Theorem*; knowing that $f(h)$ is continuous in the interval $[0, 6]$ and since, by observation, $f(a) = f(0) < 0$ and $f(b) = f(6) > 0 \Rightarrow f(a) \cdot f(b) < 0$ then we know for certain that there exists at least one root $p$ of the equation such that $f(p) = 0,\ p \in [0, 6]$; and because $f(h)$ is monotonic in $[0, 6]$, we know that this root is unique.

We can approximate the depth $h = p_n \approx p$ by using the *Bisection method* within the interval $[0, 6]$. To apply the method, let's consider the values $a_1 = 0$ and $b_1 = 6$ – the extremes of the given interval – and a value $p_1$ to be the mean of such interval, i.e. $p_1 = \frac{a_1 + b_1}{2}$. If $f(p_1) = 0$, then we have obtained the root; otherwise if $f(p_1) \neq 0$ one of the following conditions must hold:

1. $f(a_1) \cdot f(p_1) < 0$ and the root $p$ is between the points $a_1$ and $p_1$, i.e. $p \in [a_1, p_1]$
2. $f(b_1) \cdot f(p_1) < 0$ and the root $p$ is between the points $p_1$ and $b_1$, i.e. $p \in [p_1, b_1]$

If the first condition holds, then we set a new interval $[a_2, b_2]$ so that $a_2 = a_1$ and $b_2 = p_1$. Similarly, if the second condition holds we set a new interval $[a_2, b_2]$ so that $a_2 = p_1$ and $b_2 = b_1$. We then reapply this process to the interval $[a_2, b_2]$, with mean $p_2$, and continue forming intervals $[a_3, b_3], [a_4, b_4], \ldots, [a_n, b_n]$ with means $p_3, p_4, \ldots, p_n$ until the root $p$ is found or an approximation $p_n$ of a desired accuracy is achieved, where $n$ is the total number of iterations until we achieve this result. The MATLAB algorithm below calculates an approximation of $p$ within an absolute accuracy of $10^{-15}$; where we define absolute accuracy as $|p - p_n|$; and the algorithm's stopping condition is given by $\frac{b_n - a_n}{2^n} \leq TOL = 10^{-15}$

**ALGORITHM - BISECTION METHOD**

```
function t = bisection(f,a,b,TOL,MaxIter)
tic
f(a);
f(b);
for n=1:MaxIter
   p=(a+b)/2;
   if f(p)==0
      break
   elseif f(a)*f(p)<0
      b=p;
   elseif f(b)*f(p)<0
      a=p;
   end
   if (b-a)/(2^n) < =TOL
      break
   end
end
n
p
toc
```

Where the function was called with the following commands:

- ***f=inline('(((9*acos((3-x)/3))-((3-x)*sqrt((6*x)-x^2)))*8)-100')***
- ***bisection(f,0,6,10^(-15),1000)***

## Results

When executed, the algorithm for the Bisection method above approximated the depth $h = p$ within an absolute accuracy of $10^{-15}$ to be $p \approx p_n = 2.726760551333427$. The number of bisection iterations required to achieve this result was $n = 27$. The time needed to calculate this result was $0.012728$ seconds. We will store these values to use them as reference values against other root-finding methods when analyzing whether we can reach similar levels of accuracy with a reduced computational cost.

## 2.3 Newton-Raphson Method

We now proceed to approximate the depth $h = p$ with the *Newton-Raphson method* within the interval $[0, 6]$. To begin the method we need to choose an initial approximation $p_0$ to the root $p$ of equation (3). Let's consider this initial guess to be $p_0 = 1$, and let's consider the curve $y = f(h)$ with tangent line at the point $(p_0, f(p_0))$ of the form:

$$y = f(p_0) + f'(p_0)(h - p_0) \tag{4}$$

The intersection of this line with the $x$-axis is taken as the next approximation for $p$, denoted by $p_1$. Then plugging $y = 0$ and $h = p_1$ in equation (4) yields:

$$0 = f(p_0) + f'(p_0)(p_1 - p_0) \Rightarrow 0 = f(p_0) + f'(p_0)p_1 - f'(p_0)p_0$$

Solving for $p_1$ we obtain:

$$p_1 = \frac{f'(p_0)p_0 - f(p_0)}{f'(p_0)} \Rightarrow p_1 = p_0 - \frac{f(p_0)}{f'(p_0)}$$

We then repeat this process to obtain the next approximation $p_2 = p_1 - \frac{f(p_1)}{f'(p_1)}$; and continue obtaining approximations $p_3, p_4, \ldots, p_n$. In general:

$$p_n = p_{n-1} - \frac{f(p_{n-1})}{f'(p_{n-1})}$$

with $n$ being the total number of Newton-Raphson steps until an approximation of a desired accuracy is achieved. And it is easy to notice that if at some step of the Newton-Raphson algorithm we get $f'(p_n) = 0$, then the next approximation $p_{n+1}$ cannot be found and the method fails.

The following MATLAB algorithm was written to find an approximation $p_n \approx p$ within a relative precision of $10^{-14}$, where

$$f'(h) = \frac{df(h)}{dh} = 16 \cdot \sqrt{-h \cdot (h - 6)}$$

And defining relative precision as $\frac{|p_n - p_{n-1}|}{|p_n|}$; then the code is as follows:

**ALGORITHM - NEWTON-RAPHSON METHOD**

```
function r = newtonraphson(f,fprime,p,TOL,MaxNiter)
tic
for n=1:MaxNiter
   f(p);
   fprime(p);
   if fprime(p)==0
      disp('method failed')
      break
   end
   pn = p-(f(p)/fprime(p));
   if abs((pn-p)/pn)<=TOL
      break
   end
   p=pn;
end
p
n
toc
```

Where the function was called with the following commands:

- ***f=inline('(((9*acos((3-x)/3))-((3-x)*sqrt((6*x)-x^2)))*8)-100')***
- ***fprime=inline('16*sqrt(-x*(x-6))')***
- ***newtonraphson(f,fprime,1,10^(-14),1000)***

## Results

The approximation obtained when executing this code was $p \approx p_n = 2.726760594032835$ within a relative precision of $10^{-14}$. The number iterations required to achieve this result was $n = 5$. Time elapsed to calculate this result was $0.003196$ seconds.

## 2.4 Secant Method

Another way to approximate the roots of nonlinear algebraic equations is by using the *Secant method* which requires only one function evaluation at each step and it is almost as fast as the Newton-Raphson method. To begin the Secant method, we choose two initial guess approximations

$p_0$ and $p_1$ to the root $p$ of equation (3). Let's consider $p_0 = 0$, $p_1 = 1$, and the line that passes through the points $(p_0, f(p_0))$ and $(p_1, f(p_1))$, which has an equation of the form:

$$y = f(p_1) + \frac{f(p_1)-f(p_0)}{p_1-p_0}(h - p_1) \quad (5)$$

Then, the intersection of this line with the $x$-axis is taken as the next approximation for $p$, denoted by $p_2$. When plugging $y = 0$ and $h = p_2$ in equation (5) we obtain:

$$0 = f(p_1) + \frac{f(p_1)-f(p_0)}{p_1-p_0}(p_2 - p_1) \Rightarrow 0 = f(p_1) + \frac{f(p_1)-f(p_0)}{p_1-p_0}p_2 - \frac{f(p_1)-f(p_0)}{p_1-p_0}p_1$$

And solving for $p_2$:

$$\frac{f(p_1)-f(p_0)}{p_1-p_0}p_2 = \frac{f(p_1)-f(p_0)}{p_1-p_0}p_1 - f(p_1) \Rightarrow p_2 = p_1 - \frac{(p_1-p_0)f(p_1)}{f(p_1)-f(p_0)}$$

We reapply the process to obtain the next approximation $p_3 = p_2 - \frac{(p_2-p_1)f(p_2)}{f(p_2)-f(p_1)}$; and continue obtaining approximations $p_3, p_4, \ldots, p_n$. More generally:

$$p_{n+1} = p_n - \frac{(p_n-p_{n-1})f(p_n)}{f(p_n)-f(p_{n-1})}$$

with $n + 1$ being the total number of Secant steps until an approximation of a desired accuracy is achieved. The following MATLAB algorithm was written to find an approximation $p_n \approx p$ within a relative precision of $10^{-14}$, with relative precision defined as $\frac{|p_{n+1}-p_n|}{|p_{n+1}|}$, i.e. $\frac{|p_{n+1}-p_n|}{|p_{n+1}|} \leq TOL$.

**ALGORITHM - SECANT METHOD**

```
function s = secant(f,p0,p1,TOL,MaxIter)
tic
f(p0);
for n=1:MaxIter
   p2=p1-(((p1-p0)*f(p1))/(f(p1)-f(p0)));
   if abs((p2-p1)/p2)<=TOL
      break
   else
      p0=p1;
      p1=p2;
   end
end
p1
n
toc
```

Where the function was called with the following commands:

- ***f=inline('(((9*acos((3-x)/3))-((3-x)*sqrt((6*x)-x^2)))*8)-100')***
- ***secant(f,0,6,10^(-14),1000)***

## Results

When executing the code above we obtained the approximation $p \approx p_n = 2.726760594032836$ within a relative precision of $10^{-14}$. The number of iterations needed to achieve this result was $n = 6$. The time to run this algorithm was $0.002615$ seconds.

## 2.5 Computational costs

The Bisection method took $0.012728$ seconds, Newton-Raphson method took $0.003196$ seconds, and Secant method took $0.002615$ seconds. The Secant method was fastest, but required one more iteration than Newton-Raphson. We consider the Secant method computationally cheaper overall with an acceptable approximation. Bisection method is largely about checking the conditions in the loop, while Newton-Raphson requires the most function valuations. The computational costs and time required reflects the theory discussed in the course.

## Analysis of results

From the results above it is clear to notice that the number of bisection iterations required to achieve an approximation $p_n \approx p$ was considerably higher compared to both the Newton-Raphson and the Secant methods. The reason behind this is because the Bisection method converges linearly, while Newton-Raphson converges quadratically (meaning that with each iteration, the digits of accuracy are approximately doubled), and the Secant method falls somewhere in between, but closer to the converging rate of the latter method.

Another important thing to remark is the time needed to perform the algorithms for the three methods. Here, the Bisection method takes the most amount of time for the same reason as above, i.e. it converges linearly. A surprising result comes from analyzing both the Newton-Raphson Method and the Secant method, since even though the former converges faster than the latter, the time needed to perform the Secant method is slightly lower. A reason for this may be that at each

step of the Secant method only one function evaluation is needed, while at each Newton-Raphson step it requires two function evaluations, requiring less computational power. So, in practice, the Secant method is actually faster than Newton-Raphson algorithm.

The only advantage the Bisection method has against the other two, is that the bisection algorithm is the only one guaranteed to converge since both Newton-Rapshon and Secant convergence depend on the initial root guess. This means that, depending on the function, if an initial guess of the root is too far off the root we are searching for, these methods can diverge from the actual root or fail to converge in the root that we want.

Lastly, we can see that all three methods converged to a root approximation $p_n$ up to the first seven decimal places (i.e. $p_n = 2.7267605$) while both the Secant method and the Newton-Raphson methods converged to a root approximation $p_n$ to the first 14 decimal places (i.e. $p_n = 2.72676059403283$) which leads us to believe that the last two methods used are more precise compared to the first one. The table below shows all these results.

| **Method** | **Iterations ($n$)** | **Approximation ($p_n$)** | **Time ($s$)** |
|---|---|---|---|
| *Binomial* | 27 | $2.726760551333427$ | $0.012728$ |
| *Newton-Raphson* | 5 | $2.726760594032835$ | $0.003196$ |
| *Secant* | 6 | $2.726760594032836$ | $0.002615$ |

# 3. Interpolation and Curve Fitting

According to *www.worldometers.info*, world's total population in billions from 1955 to 1990 is given in the following table:

| Year | 1955 | 1960 | 1965 | 1970 | 1975 | 1980 | 1985 | 1990 |
|---|---|---|---|---|---|---|---|---|
| Population | 2.7730 | 3.0349 | 3.3396 | 3.7004 | 4.0795 | 4.4580 | 4.8709 | 5.3272 |

1. Using the given data set, construct the seventh degree Newton interpolating polynomial. Plot the graph of the obtained polynomial from the year 1955 to the year 1990. Data points must be indicated on the curve with markers. Use the seventh degree Newton polynomial to interpolate the world's population in 1976. Compare this result with the actual world's population in 1976, which is approximately 4.1547 billion according to *www.worldometers.info*.
2. Find the least squares line for given data by using the normal equations. Plot the least squares line from the year 1955 to the year 1990. Data points must be shown with markers. Use the least squares line to approximate the world's population in 1976. Compare your result with the actual world's population in 1976. Does the least squares line give a better approximation for the world's population in 1976 than the polynomial interpolation?

## 3.1 Newton Interpolating Polynomial

The $n$-th degree Newton interpolating polynomial that passes through $n + 1$ points has the form:

$$P_n(x) = a_0 + a_1(x - x_0) + a_2(x - x_0)(x - x_1) + \ldots + a_n(x - x_0)\ldots(x - x_{n-1}) \qquad (1)$$

Where the constants $a_0, a_1, \ldots, a_n$ are given by:

$$a_0 = f[x_0]$$
$$a_1 = f[x_0, x_1]$$
$$a_2 = f[x_0, x_1, x_2]$$
$$a_n = f[x_0, x_1, x_2, \ldots, x_n]$$

And the right side of the equations are called the divided differences of function $f$, defined as

$$f[x_k] = f(x_k),\ k = 0, 1, \ldots, n$$

$$f[x_{k-1}, x_k] = \frac{f[x_k] - f[x_{k-1}]}{x_k - x_{k-1}}\ k = 1, 2, \ldots, n$$

$$f[x_{k-2}, x_{k-1}, x_k] = \frac{f[x_{k-1}, x_k] - f[x_{k-2}, x_{k-1}]}{x_k - x_{k-2}}\ k = 2, 3, \ldots, n$$

$$f[x_{k-3}, x_{k-2}, x_{k-1}, x_k] = \frac{f[x_{k-2}, x_{k-1}, x_k] - f[x_{k-3}, x_{k-2}, x_{k-1}]}{x_k - x_{k-3}}\ k = 3, \ldots, n$$

In general:

$$f[x_0, x_1, x_2, \ldots, x_n] = \frac{f[x_1, x_2, x_3, \ldots, x_n] - f[x_0, x_1, x_2, \ldots, x_{n-1}]}{x_n - x_0}$$

To construct the seventh degree Newton polynomial for the data given above, we take the 8 points from the data above; so that $n = 7$. Next, we need to calculate the $1^{st}$, $2^{nd}$, … , $7^{th}$ divided differences for all of these points and finally we can build the $7^{th}$ degree Newton polynomial. Since this process is both tedious and has substantial room for error when carrying out the calculations analytically, the following MATLAB code was written to perform the calculations instead.

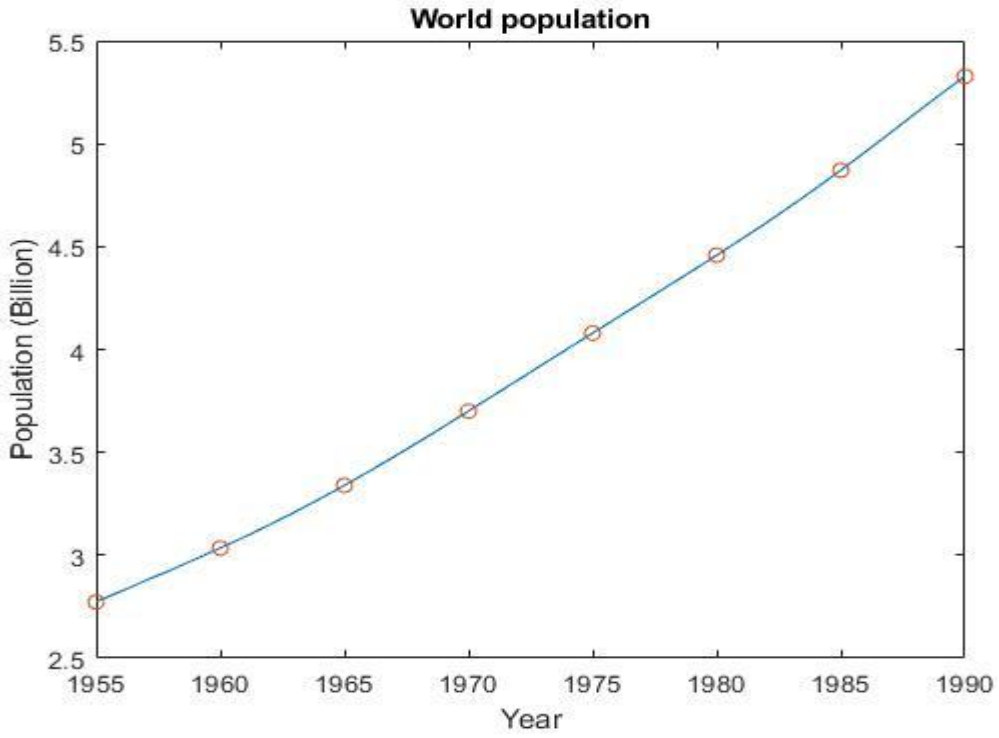
**ALGORITHM - NEWTON INTERPOLATING POLYNOMIAL**

```
function pvalue = newtoninterpolationfunc(year)
x=[1955 1960 1965 1970 1975 1980 1985 1990];
y=[2.7730 3.0349 3.3396 3.7004 4.0795 4.4580 4.8709 5.3272];
n=7; [m s]=size(x); M=[x' y'];
for k=3:s+1
  for i=k-1:s
     M(i,k)=(M(i,k-1)-M(i-1,k-1))/(M(i,1)-M(i-(k-2),1));
  end
end
terms=ones(1,n+n);
for k=1:n
   for i=1:n
   terms(i+k)=terms(i+k)*(year-x(k));
   end
end
for k=1:n+1
   terms(k)=terms(k)*M(k,k+1);
end
pvalue=0;
for k=1:n+1
   pvalue=pvalue+terms(k);
end
```

And the script used to plot the interpolation function is:

```
clear; clc
for k=1955:1990
   polyvalues(k-1954)=newtoninterpolationfunc(k);
end
xvalues=1955:1990;
plot(xvalues,polyvalues)
hold on
plot ([1955 1960 1965 1970 1975 1980 1985 1990],[2.7730 3.0349 3.3396 3.7004 4.0795 4.4580
4.8709 5.3272],'o')
title('World population')
xlabel('Year')
ylabel('Population (Billion)')
```

The output graph is as follows:




We can interpolate the world's population for 1976 by evaluating the function for $x = 1976$:

i.e. *newtoninterpolationfunc*(1976) = 4.1548. And according to *www.worldometers.info* the world's population in 1976 was of $4.1547$, so our interpolation contains an absolute error of $|4.1547 - 4.1548| = 0.0001 = 1 \cdot 10^{-4}$.

## 3.2 Curve Fitting - Least Squares Line

The least squares method refers to minimizing the sum

$$S = \sum_{k=1}^{n} \left(f(x_k) - y_k\right)^2$$

The least squares line given by:

$$y = f(x) = Ax + B$$

Is the line that minimizes the sum

$$S = \sum_{k=1}^{n} (Ax + B - y_k)^2$$

Where the coefficients $A$ and $B$ can be calculated from the *normal equations*:

$$A \sum_{k=1}^{n} x_k^2 + B \sum_{k=1}^{n} x_k = \sum_{k=1}^{n} x_k \cdot y_k \quad (1)$$

$$A \sum_{k=1}^{n} x_k + B \times n = \sum_{k=1}^{n} y_k \quad (2)$$

We calculate the sums of the terms in the following table:

| $k$ | $x_k$ | $y_k$ | $x_k^2$ | $x_k \cdot y_k$ |
|---|---|---|---|---|
| 1 | 1,955 | 2.7730 | 3,822,025 | 5,421.215 |
| 2 | 1,960 | 3.0349 | 3,841,600 | 5,948.404 |
| 3 | 1,965 | 3.3396 | 3,861,225 | 6,562.314 |
| 4 | 1,970 | 3.7004 | 3,880,900 | 7,289.788 |
| 5 | 1,975 | 4.0795 | 3,900,625 | 8,057.013 |
| 6 | 1,980 | 4.4580 | 3,920,400 | 8,826.84 |
| 7 | 1,985 | 4.8709 | 3,940,225 | 9,668.737 |
| 8 | 1,990 | 5.3272 | 3,960,100 | 10,601.13 |
| | $\sum_{k=1}^{8} x_k = 15,780$ | $\sum_{k=1}^{8} y_k = 31.583$ | $\sum_{k=1}^{8} x_k^2 = 31,127,100$ | $\sum_{k=1}^{8} x_k \cdot y_k = 62,375.44$ |

Replacing these values in equations (1) and (2) we obtain:

$$31,127,100 \times A + 15,780 \times B = 62,375.44 \quad (3)$$

$$15,780 \times A + 8 \times B = 31.583 \quad (4)$$

From equation (4) we solve for $B$ to obtain:

$$B = \frac{31.583 - 15,780 \times A}{8} = 3.947875 - 1,972.5 \times A$$

And replacing this in equation (3) yields:

$$31,127,100 \times A + 15,780 \cdot (3.947875 - 1,972.5 \times A) = 62,375.44$$

$$\Rightarrow 31,127,100 \times A + 62,297.4675 - 31,126,050 \times A = 62,375.44$$

$$\Rightarrow 1050 \times A = 77.9725 \Leftrightarrow A \approx 0.0733183333335233$$

And $B \approx -140.672475000375$

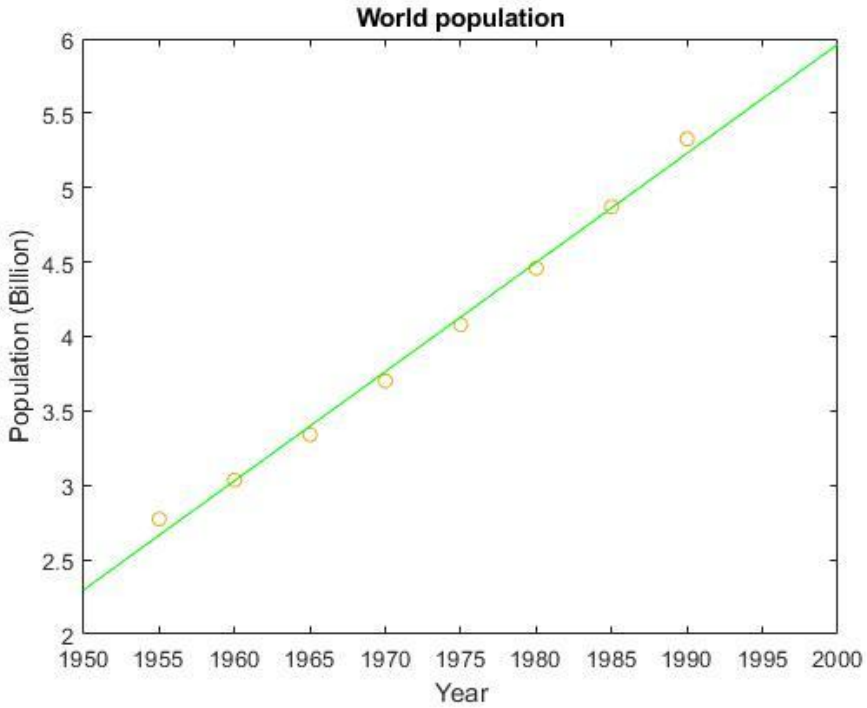
So the least squares line is given by the line of equation:

$$y = f(x) \approx 0.0733183333335233x - 140.672475000375$$

We can approximate the world's population in 1976 by solving

$$f(1976) \approx 0.0733183333335233 \times 1,976 - 140.672475000375 = 4.2046$$

This approximation can also be calculated through a MATLAB algorithm which generated the following graph of the least squares line method:



**ALGORITHM - LEAST SQUARES LINE (CURVE FITTING)**

```
clear; clc
x=[1955 1960 1965 1970 1975 1980 1985 1990];
y=[2.7730 3.0349 3.3396 3.7004 4.0795 4.4580 4.8709 5.3272];
[m n]=size(x);
sumx=sum(x);
sumy=sum(y);
sum2x=sum(x.^2);
sumxy=sum(x.*y);
M=[sum2x sumx; sumx n];
b=[sumxy; sumy];
p=linsolve(M,b);
p=p';
xx=linspace(1950,2000,100);
pvalues=polyval(p,xx);
plot(x,y,'o',xx,pvalues,'g')
title('World population')
xlabel('Year')
ylabel('Population (Billion)')
polyval(p,1976)
```

This script approximates the world population in 1976 at 4.2046; which is, as expected, the same as the analytical result achieved before. Comparing this result with the data given by *www.worldometers.info*, we can calculate the absolute error of our approximation to be $|4.1547 - 4.2046| = 0.0499 = 4.99 \times 10^{-2}$. This is indeed a worse approximation than achieved when using Newton interpolation. However, since curve fitting methods are mostly used when having many more data points that likely have higher input errors, this is quite reasonable.

## Analysis of results

The Least Squares Line is a line constructed to be in the middle of all scattered data points. It minimizes the sum of error-distances squared (they are squared because absolute values are less practical in this application, and derivatives are more difficult to attain that way). The Newton Interpolating Polynomial is constructed to cross through all data points. That is why we don't get a substantial error as we did with the curve fitting method when comparing with the true value of the population in 1976. Based on the errors obtained from both methods, we can conclude that the Newton Interpolating Polynomial approximation is more accurate for this type of application.

# 4. Numerical Integration

Approximate the integral

$$I = \int_0^1 e^{-x^2} dx$$

by using the Trapezoidal and Simpson's rules with the stepsizes

$$h = h_k = 2^{-k},\ k = 1, 2, \dots, 10.$$

Using the value $I = 0.746824132812427$, known from the literature, calculate the absolute errors in these two approximations for each step size $h_k$. Plot in one frame the errors against the stepsizes $\{h_k\}_{k=1}^{10}$ in log-log scale. Illustrate that the numerical results agree with the theoretical order of convergence.

## 4.1 Trapezoidal Rule

If a function $f''(x)$ is continuous on an interval $[a, b]$, then there exists a point $\xi \in [a, b]$ such that:

$$\int_a^b f(x)dx = T_N - \frac{h^2}{12}(b - a)f''(\xi), \quad (1)$$

Where

$$T_N = \frac{h}{2}\left(f(x_0) + 2\sum_{k=1}^{N-1} f(k) + f(x_N)\right) = \frac{h}{2}\left(2f(x_1) + 2f(x_2) + \dots + 2f(x_{N-1}) + f(x_N)\right)$$

is the *Trapezoidal sum*. Assuming that $h$ is small enough, we can approximate the integral by

$$\int_a^b f(x)dx \approx \frac{h}{2}\left(f(x_0) + 2\sum_{k=1}^{N-1} f(k) + f(x_N)\right) \quad (2)$$

Equation (2) is called the *Trapezoidal rule* with step size $h$ for the integral of function $f(x)$ over interval $[a, b]$; and the truncation error in such approximation is of order $O(h^2)$.

To approximate the integral $I = \int_0^1 e^{-x^2} dx$ with step sizes $h_k = 2^{-k}$, $k = 1, 2, \dots, 10$, we have written the following MATLAB algorithm that takes as arguments a function, the boundaries for where we want to approximate the integral and the number of intervals that should be used. The step sizes will be $h = \frac{b-a}{N}$ where $a$, $b$ are the boundaries and $N$ is the number of intervals.

**ALGORITHM - TRAPEZOIDAL RULE**

```
function integral = trapezoidalsum(f,a,b,N)
integral=0;
xk=linspace(a,b,N+1);
h=(b-a)/N;
for k=0:N
   if k==0||k==N
      integral=integral+f(xk(k+1));
   else
      integral=integral+2*f(xk(k+1));
   end
end
integral=(h/2)*integral;
```

And the following commands were used to evaluate this function:

- ***f=inline('*exp(-x^2)')**
- **trapezoidalsum(f,0,1,100)**

The output for the code above was:

$I = 0.746818001467970$

Using the true value know from the literature $I = 0.746824132812427$, we can calculate the absolute error as:

$|0.746824132812427 - 0.746818001467970| = 6.131344457038779 \cdot 10^{-6}$

## 4.2 Simpson's Rule

If a function $f^{(4)}(x)$ is continuous on an interval $[a, b]$, then there exists a point $\xi \in [a, b]$ such that:

$$\int_a^b f(x)dx = S_N - \frac{h^4}{180}(b - a)f^{(4)}(\xi), \qquad (3)$$

Where

$$S_N = \frac{h}{3}\left(f(x_0) + 4\sum_{k=1}^{N/2} 2f(x_{2k-1}) + 2\sum_{k=1}^{(N/2)-1} f(x_{2k}) + f(x_N)\right)$$

$$\Rightarrow S_N = \frac{h}{3}\left(f(x_0) + 4f(x_1) + 2f(x_2) + 4f(x_3) + \ldots + 2f(x_{N-2}) + 4f(x_{N-1}) + f(x_N)\right)$$

is the *Simpson's sum*. And, assuming that $h$ is small enough, we can approximate the integral by

$$\int_a^b f(x)dx \approx \frac{h}{3}\left(f(x_0) + 4\sum_{k=1}^{N/2} 2f(x_{2k-1}) + 2\sum_{k=1}^{(N/2)-1} f(x_{2k}) + f(x_N)\right) \quad (4)$$

Equation (4) is called the *Simpson's rule* with step size $h$ for the integral of function $f(x)$ over interval $[a, b]$; and the truncation error in such approximation is of order $O(h^4)$.

To approximate the integral $I = \int_0^1 e^{-x^2} dx$ with step sizes $h_k = 2^{-k}$, $k = 1, 2, \dots, 10$, the following MATLAB algorithm was written. This algorithm takes as arguments a function, the boundaries for where we want to approximate the integral and the number of intervals that should be used. The step sizes will be $h = \frac{b-a}{N}$ where $a$, $b$ are the boundaries and $N$ is the number of intervals.

**SIMPSONS FUNCTION**

```
function integral = simpsons(f,a,b,N)
integral=0;
xk=linspace(a,b,N+1);
h=(b-a)/N;
t=0;
for k=0:N
   if k==0||k==N
      integral=integral+f(xk(k+1));
      t=t+1;
   elseif t==1
      integral=integral+4*f(xk(k+1));
      t=t-1;
   else
      integral=integral+2*f(xk(k+1));
      t=t+1;
   end
end
integral=(h/3)*integral;
```

And the following commands were used to evaluate this function:

- ***f=inline('*exp(-x^2)')**
- **simpsons(f,0,1,100)**

Which resulted in an approximation of $I = 0.746824132894176$ with an absolute error of $|0.746824132812427 - 0.746824132894176| = 8.174905197222415 \cdot 10^{-11}$

## Analysis of results

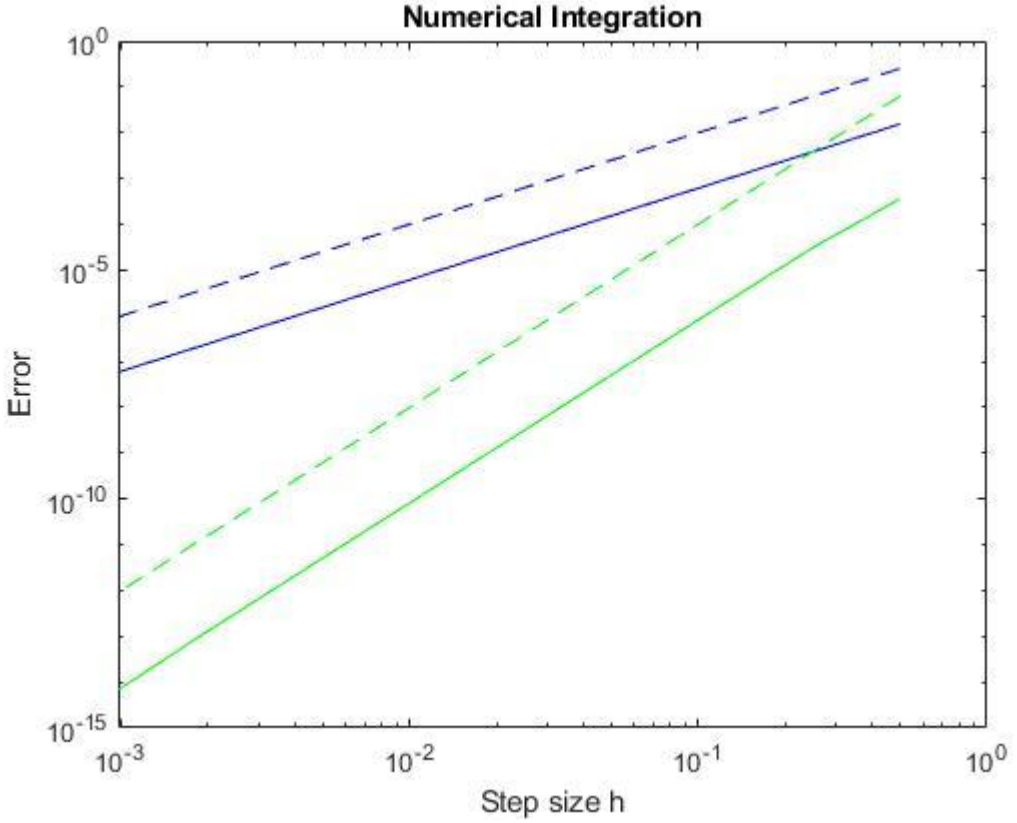
We can compare our solutions with the correct value I to get the error for different step sizes. We have plotted these errors using log-log scale together with the theoretical magnitude of error for each step size using the following script.

**SCRIPT**

```
I=0.746824132812427;
a=0;
b=1;
k=1:10;
h=2.^(-k);
f=@(x) exp(-x.^2);
m=1;
for n=1:10
   errort(m)=abs(I-trapezoidalsum(f,a,b,2^(n)));
   m=m+1;
end
m=1;
for n=1:10
   errors(m)=abs(I-simpsons(f,a,b,2^(n)));
   m=m+1;
end
loglog(h,errort,'b')
hold on
loglog(h,h.^2,'b--')
hold on
loglog(h,errors,'g')
hold on
loglog(h,h.^4,'g--')
```

The output is shown below. Here, the blue line is the error for the trapezoidal sum and the green line is the error for the simpsons sum. Since the theoretical error of the trapezoidal sum is $O(h^2)$, we have plotted $h^2$ with a blue dotted line to compare. The same is done for the simpsons sum where we have plotted $h^4$ with a green dotted line since its theoretical error is $O(h^4)$.

From theory we expect that the errors should be of the same slope as the corresponding dotted lines. That is, the trapezoidal sum should have an error of $O(h^2)$ which corresponds to the blue dotted line and the Simpsons sum should have an error of $O(h^4)$ which corresponds to the green dotted line. Since this is what we observe, we can conclude that our calculations are in line with the theory.


Numerical Integration
Error
Step size h
$10^{0}$
$10^{-5}$
$10^{-10}$
$10^{-15}$
$10^{-3}$
$10^{-2}$
$10^{-1}$
$10^{0}$

## 5. Numerical Solution of Differential Equations

One of the simplest mathematical models known in epidemiology, was proposed in 1927 by W.O.Kermack and A.G.McKendrick. It is defined by the following system of differential equations

$$\begin{cases} S'(t) = -\frac{\beta}{N} I(t)S(t) \\ I'(t) = \frac{\beta}{N} I(t)S(t) - \gamma I(t) \\ R'(t) = \gamma I(t) \end{cases}$$

where *S(t)* denotes the number of susceptible individuals, *I(t)* denotes the number of infected individuals, *R(t)* denotes the number of recovered individuals, $\beta$ is the infection rate, $\gamma$ is the recovery rate and $N = S + I + R$ is the total population being constant in time. (3) is known in the literature as the *SIR* model. Assume that $S(0) = 997$, $I(0) = 3$, $R(0) = 0$, $\beta = 0.4$, $\gamma = 0.04$ and $t \in [0, 150]$.

- Use Matlab ODE solver *ode45* to calculate the numerical solution of the *SIR* model with given parameters and initial conditions. Plot in one frame the numerical solutions of *S*(t), *I*(t) and *R*(t) against time. What do you observe from the plots and how do you interpret the dynamics of the system?
- Write the code implementing the classical 4-th order RK method Runge-Kutta method and use it with step size $\tau = 10^{-3}$ to find the numerical solution of the *SIR* model with given parameters and initial conditions. Plot in one frame the numerical solutions of *S*(t), *I*(t) and *R*(t) against time. Make a qualitative comparison with the results obtained with *ode45* solver.

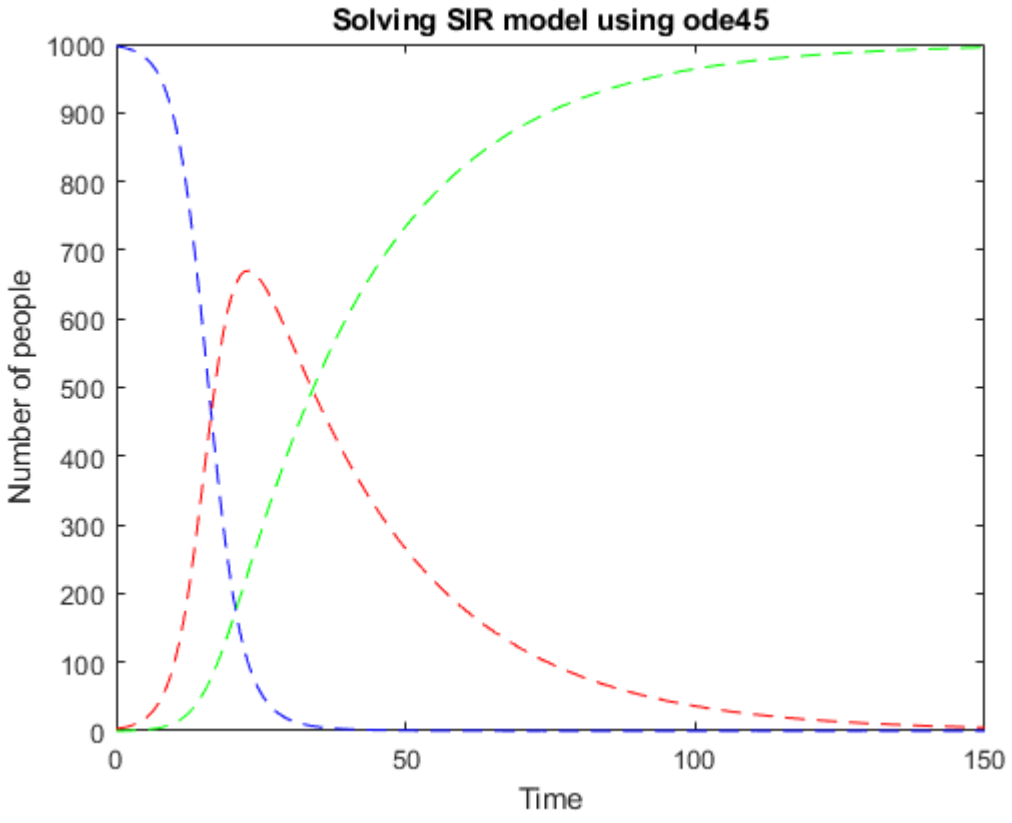
## 5.1 MATLAB ODE SOLVER *ode45*

We want to use ode45 to solve this system of equations. This function takes three arguments, the first one is the equations that we want to solve, the second is what span of time we are looking at and the third is our initial values. Since we have a system of equations we need to create a vector function and use that as the first argument. We do the same for our third argument, we have three initial values and we use a vector with those values as input. So our inputs are

$$F(t,y) = \begin{bmatrix} -\frac{0.4}{1000} I(t)S(t) \\ \frac{0.4}{1000} I(t)S(t) - 0.04I(t), \\ 0.04I(t) \end{bmatrix} \qquad t \in [0, 150]$$

$$\text{and } y_0 = \begin{bmatrix} 997 \\ 3 \\ 0 \end{bmatrix}$$

Our script executed in MATLAB generates the following plot:



The blue curve represents the number of susceptible individuals, the red curve is the number of infected individuals and the green curve the number of recovered individuals. We can see from the plot that in the beginning, about from when $0 \leq t \leq 30$, the number of infected people are increasing and the number of susceptible people are decreasing since the people that are susceptible are becoming people that are infected. After that the number of infected people are decreasing because there are no longer any susceptible people left, all of them have been infected. From then the number of recovered people increases until there are no infected people left.

**SCRIPT**

```
f=@(t,y) [((-0.4/1000)*y(2)*y(1));((0.4/1000)*y(2)*y(1)-0.04*y(2));0.04*y(2)];
y0=[997 3 0];
tspan=[0 150];
[t y]=ode45(f,tspan,y0);
plot(t,y(:,1),'--b',t,y(:,2),'--r',t,y(:,3),'--g')
title('Solving SIR model using ode45')
xlabel('Time')
ylabel('Number of people')
```

## 5.2 Runge-Kutta Method

We will now solve the system of equations again with Matlab using the classical 4-th order Runge-Kutta method and compare the results to what we got using ode45. We will use the method with step sizes $\tau = \frac{b-a}{N} = \frac{150}{N} = 10^{-3} \Leftrightarrow N = 150000$ i.e. with 150000 iterations. Now we define

$$k_1 = f\left(t_{n-1}, y_{n-1}\right)$$

$$k_2 = f\left(t_{n-1} + \frac{\tau}{2}, y_{n-1} + \frac{\tau}{2} k_1\right)$$

$$k_3 = f\left(t_{n-1} + \frac{\tau}{2}, y_{n-1} + \frac{\tau}{2} k_2\right)$$

$$k_4 = f\left(t_{n-1} + \frac{\tau}{2}, y_{n-1} + \frac{\tau}{2} k_3\right)$$

$$y_n = y_{n-1} + \tau\left(\frac{1}{6} k_1 + \frac{1}{3} k_2 + \frac{1}{3} k_3 + \frac{1}{6} k_4\right)$$

where $t_n = \tau n$ is the time on iteration $n$ and $k_1, k_2, k_3, k_4$ are intermediate weighted measurements to get an estimated solution. Our function takes in arguments $f$, $a$, $b$, $y_0$ and $N$ i.e. the vector function with our system of equations, the starting and end point in time, $y_0$, is the initial values and $N$ is the number of iterations we want to use.

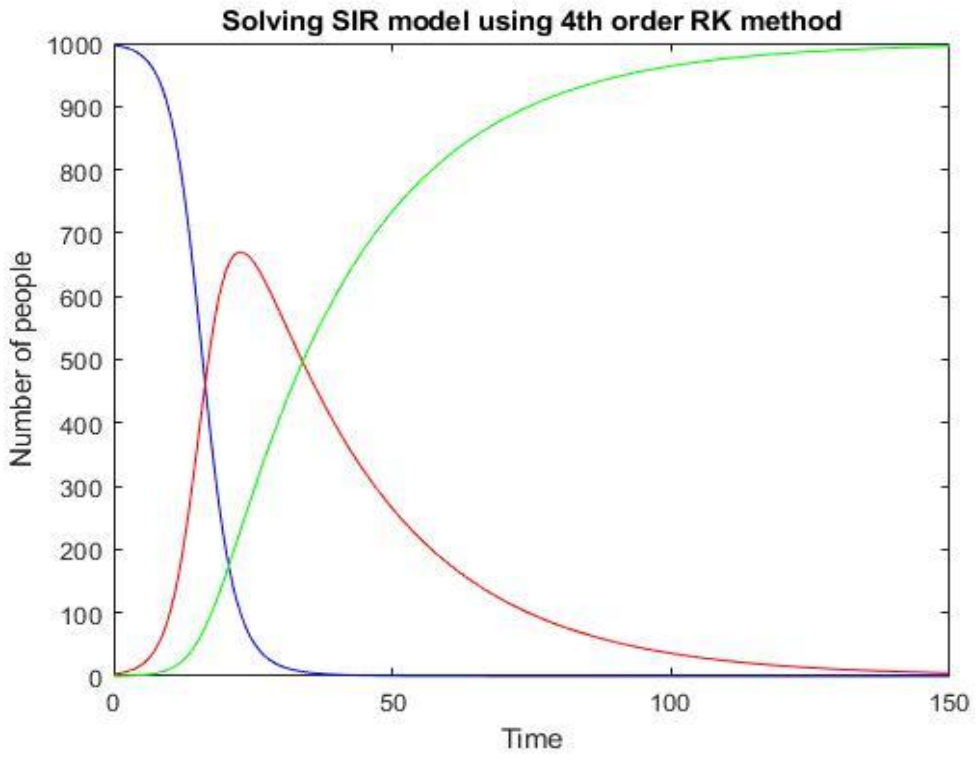
**ALGORITHM - 4TH ORDER RUNGE-KUTTA FUNCTION**

```
function r = rk4(f,a,b,y0,N)
q=(b-a)/N;
t=linspace(a,b,N+1)';
yn=y0;
for j=1:N
   k1=f(t(j),yn(j,:))';
   k2=f(t(j)+(q/2),yn(j,:)+(q/2)*k1)';
   k3=f(t(j)+(q/2),yn(j,:)+(q/2)*k2)';
   k4=f(t(j)+q,yn(j,:)+q*k3)';
   yn(j+1,:)=yn(j,:)+((1/6)*q*(k1+2*k2+2*k3+k4));
end
[t yn];
plot(t,yn(:,1),'b',t,yn(:,2),'r',t,yn(:,3),'g')
```

Where the function was called with the following commands:

- **f=@(t,y) [((-0.4/1000)*y(2)*y(1));((0.4/1000)*y(2)*y(1)-0.04*y(2));0.04*y(2)];**
- **y0=[997 3 0];**
- **rk4(f,0,150,y0,150000)**

And the resulting plot was:



## Analysis of results

The plots achieved in the second part of the question are similar to the first. MATLAB's *ode45*-solver is based on an explicit Runge-Kutta formula according to their documentation[1], this reassures us that the plot from the Runge-Kutta MATLAB code is accurately depicting the first order differential equation. The implications from the plots are, as previously argued, that susceptible, infected and recovered individuals run their natural course then start to converge to their limit values. Everyone in the population sample will have been susceptible, everyone will have gotten infected and everyone will have recovered at time's end.

[1] https://se.mathworks.com/help/matlab/ref/ode45.html#bu0200e-1 (11/3-22)