M2272 Project

Adrian Azar

April 2025

Contents

1 Roots of f(x)

In this chapter I am going to be solving the equation $f(x) = 0$ in order to find the **roots** with the help of root finding numerical methods.

1.1 Bisection Method

This is the most basic numerical method that is based on the **Intermediate Value Theorem**.

Theorem 1.1 (Intermediate Value Theorem)**.** *Let $f : [a, b] \to \mathbb{R}$ be a continuous function. If $f(a) \neq f(b)$, and k is a real number such that*

$$f(a) < k < f(b) \quad \textit{or} \quad f(b) < k < f(a),$$

then $\exists c \in (a, b)$ such that

$$f(c) = k.$$

In particular, if $f(a) \cdot f(b) < 0$, then $\exists c \in (a, b)$ such that

$$f(c) = 0.$$

The **Bisection Method** is one of the, if not, the best numerical method the relies on the bisecting the interval $[a, b]$ until it is small enough where a and b coincide with a root of $f(x)$, with some error because a and b may only be approximately equal to the root.

1.1.1 Mathematical Approach

Theorem 1.2. *Let f be a function. We say that f is strictly monotonic on an interval $]a, b[$ if for all $x_1, x_2 \in]a, b[$, $x_1 < x_2$ implies that $f(x_1) < f(x_2)$ (strictly increasing) or $f(x_1) > f(x_2)$ (strictly decreasing).*

Until the end of this method, f will always be monotone on $[a, b]$.

1. Let $c = \frac{a+b}{2}$
2. If $f(a) \cdot f(b) < 0 \Rightarrow b = c$, else $a = c$
3. If $|b - a| \leq \epsilon$, then we $\implies$ stop. Note that $\epsilon = 10^{-6}$., where $|b - a|$ is the error and ϵ is the **maximum error**.

You will of course have to loop this process.

1.1.2 Code

Here is the **Python** code for the Bisection Method:

```python
import math as m

def f(x):
    f =  #enter a mathematical fucntion
    return f

# Bisection method
a = float(input('First initial guess a='))
b = float(input('Second initial guess b='))
eps = float(input('What is the error? '))
error = float(eps) + 1
nmax = int(input('The max number of iterations is= '))
n = 1

while (n < nmax) and (error > eps):
    c = (a + b) / 2
    if f(a) * f(c) <= 0:
        b = c
    else:
        a = c
    error = m.fabs(b - a)
    n = n + 1

print(f'The solution is {a} at an error of {error}')
print(f'The number of iterations is {n}')
```

Here is the **MATLAB** code for the Bisection Method: (assuming fucntion already defined)

```matlab
clear
clc
a=input('a= ');
b=input('b= ');
eps=input('eps= ');
N=input('N= ');
n=1;
error=eps+1;
while(n<N)&&(error>eps)
    c=(a+b)/2;
    if(f(a)*f(c)<=0)
        b=c;
    else
        a=c;
    end
    error=abs(a-b);
    n=n+1;
end
```

```
fprintf('The solution is %f near %f \n',a,error);
fprintf('The number of iterations is %g',n)
```

1.2 Newton-Raphson Method

1.2.1 Mathematical Approach

Theorem 1.3.

$$f(x) = 0$$

The idea behind the Newton-Raphson method is to approximate the function near a guess x_0 using a linear approximation. To do this, we use the first-order Taylor expansion of the function $f(x)$ around x_0:

$$f(x) \approx f(x_0) + f'(x_0)(x - x_0)$$

We want to find the value of x where $f(x) = 0$. So, we set the above approximation equal to zero:

$$0 = f(x_0) + f'(x_0)(x - x_0)$$

Solving for x, we get:

$$x = x_0 - \frac{f(x_0)}{f'(x_0)}$$

Thus, the Newton-Raphson iteration formula is:

$$\boxed{x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}}$$

Unlike the Bisection method, the Newton-Raphson method has one major **problem**, the **convergence** of the **initial choice** x_0.

1.2.2 Conditions

The choice of the **initial condition** depends on the following:

1. The relation of the concavity $f''(x_0)$ with $f(x_0)$, $f''(x_0).f(x_0)$ <0, for the method to actually converge to the solution and not diverge.
2. The **Absolute Relative Error**, ϵ, must be less than or equal to 10^{-6}:

$$\epsilon = |\frac{x_{i+1} - x_i}{x_{i+1}}| \leq 10^{-6} \quad (1)$$

3. When the first derivative $f'(x_n) = 0$ in the denominator of the Newton-Raphson formula occurs, we obviously can continue finding a solution.
4. Root jumping when a function like trigonometric functions that have many roots.

1.2.3 Code

Here is the **Python** code for the Newton-Raphson method:

```python
import math as m
def f(x):

    f= #choose a suitable function
    return f
def df(x):
    f=3*m.pow(x,2)-2
    return f

def d2f(x):
    f=6*x
    return f

a=float(input('a= '))
eps=float(input('eps= '))
error=eps+1
nmax=int(input('Max iterations= '))
if f(a)*d2f(a)<0:
    print(f'{a} is not a good initialization')
elif f(a)*d2f(a)==0:
    print('Unknown')
elif f(a)*d2f(a)>0:
    n=1
    while (error>eps) and (nmax>n):
        x=a-f(a)/df(a)
        error=m.fabs((x-a)/x)
        a=x
        n=n+1
    print(f'{a} is the root with {n} iterations and an error
        of {error}')
```

Here is the **MATLAB** codes for the Newton-Raphson Method:(assuming function and its derivative is defined)

```matlab
clear
clc
x=input('x= ');
eps=input('eps= ');
N=input('N= ');
n=1;
error=eps+1;
while(n<N)&&(error>eps)
    y=x-f(x)/df(x);
    error=abs((y-x)/y);
    x=y;
    n=n+1;
end
```

```
fprintf('The solution is %f with an error of %f and %g
    iterations',y,error,n)
```

1.3 Secant Method

1.3.1 Mathematical Interpretation

Theorem 1.4. *Newton - Raphson method is given by:*

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$$

However, the Secant Method approximates the derivative so

$$f'(x_n) \approx \frac{f(x_n) - f(x_{n-1})}{x_n - x_{n-1}}$$

Substitute this approximation into Newton's formula:

$$x_{n+1} = x_n - \frac{f(x_n)}{\frac{f(x_n) - f(x_{n-1})}{x_n - x_{n-1}}}$$

Simplify the expression:

$$\boxed{x_{n+1} = x_n - f(x_n) \cdot \frac{x_n - x_{n-1}}{f(x_n) - f(x_{n-1})}}$$

This is the Secant Method equation

1.3.2 Conditions

The conditions for the Secant Method include:

1. The denominator $f(x_n) - f(x_{n-1})$ must **NOT** be zero at 2 points x_n and x_{n-1}.
2. After several interations, the **Absolute Relative Error**, ϵ, must be less than or equal to 10^{-6}:

$$\epsilon = \left|\frac{x_{i+1} - x_i}{x_{i+1}}\right| \leq 10^{-6} \tag{2}$$

3. Root jumping just like in the Newton-Raphson method.

1.3.3 Code

Here is the **Python** code

```
    import math as m
def f(x):
    E=m.e
    f=    #choose a suitable function
    return f

x0=float(input('x0= '))
x1=float(input('x1= '))
eps=float(input('eps= '))
error=eps+1
nmax=int(input('Max iterations= '))
n=1
while (error>eps) and (nmax>n):
        x2=x1-(f(x1)*(x1-x0))/(f(x1)-f(x0))
        error=m.fabs((x2-x1)/x2)
        x0=x1
        x1=x2
        n=n+1

print(f'{x0} is the root with {n} iterations and an error of
     {error}')
```

Here is the **MATLAB** code:(assuming function and the derivative is already defined)

```
clear
clc
x0=input('x0= ');
x1=input('x1= ')
eps=input('eps= ');
N=input('N= ');
n=1;
error=eps+1;
while(n<N)&&(error>eps)
    x2=x1-(f(x1)*(x1-x0))/(f(x1)-f(x0));
    x0=x1;
    x1=x2;
    error=abs((x2-x1)/x2)
    n=n+1;
end
fprintf('The solution is %f with an error of %f and %g
    iterations',x2,error,n)
```

1.4 Fixed Point Method

We want to solve

$$f(x) = 0$$

by iterative approximation.

Starting from an initial guess x_0, we compute successive values:

$$\begin{aligned} x_1 &= f(x_0) \\ x_2 &= f(x_1) \\ x_3 &= f(x_2) \\ &\vdots \\ x_{i+1} &= f(x_i) \\ &\vdots \\ x_{n+1} &= f(x_n) = g(x_n) \end{aligned}$$

General Formula:

$$\boxed{x_{k+1} = g(x_k)} \quad \text{for } k = 0, 1, 2, \ldots, n$$

1.4.1 Conditions

Theorem 1.5. *Convergence Theorem: Let $g : [a, b] \to \mathbb{R}$ where $g(x)$ is convergent if it satisfies the following:*

1. ***Contraction Hypothesis:***
   g contractant and defined in $[a, b]$
   $\forall (x_1, x_2) \in [a, b], \exists K < 1,$

$$\left| \frac{g(x_1) - g(x_2)}{x_1 - x_2} \right| \leq K$$

or

$$\boxed{g'(x) < 1}$$

2. ***Inclusion Hypothesis:***
   $g \subset [a, b]$

$$\Rightarrow \forall x \in [a, b], g(x) \in [a, b]$$

$$\Rightarrow \boxed{g([a, b]) \subset [a, b]}$$

If both of these are proven, then the chosen initial x_0 will converge to a solution.

1.5 Code

The **Python** code is:

```python
import math as m
def g(x):
    g= #define a suitable function
    return g
# fixed point method
x0=float(input('initial guess x0='))
eps=float(input('What is the error? '))
error= float(eps) + 1
nmax=int(input('The max number of iterations is= '))
n=1
while (n<nmax) and (error>eps):
    x1=g(x0)
    error=m.fabs(x1-x0)
    x0=x1
    n=n+1

print(f'The solution is {x0} at an error of {error}')
print(f'The number of iterations is {n}')
```

The **MATLAB** code is:(assuming function is defined)

```matlab
    clear
clc
x0=input('x0= ');
eps=input('eps= ');
error=eps+1;
N=input('N= ');
n=1;
while(error>eps)&&(N>n)
    x1=g(x0);
    error=abs((x1-x0)/x1);
    x0=x1;
    n=n+1;
end
fprintf('The solution is %f near %f with %g iterations',x1,
    error,n)
```

2 Differential Equations

2.1 Analytical Solution

In this chapter, we will **ONLY** solve first order differential equations(first order **ODE**) with initial conditions:

$$\frac{dy}{dx} = f(x, y(x))$$

with $y(0) = y_0$.
In order to solve this type of **ODE** we need to verify the following conditions of the **Cauchy Problem**(or of initial values):

1. The function must be defined and continuos $\forall(x, y)$
2. The function must be **Lipschitizian**

   Lemma 2.1. ***Lipschitizian***
   $\forall(y_1, y_2) \in \mathbb{R}$, $\exists L \geq 0/\ |f(x, y_2) - f(x, y_1)| \leq L|y_2 - y_1|$

2.2 Numerical Solutions

Now, we shall solve ODEs using different numerical methods. We use numerical methods **ESPECIALLY** when the equation we are solving is non-linear and/or unsolvable via analytical methods.Numerically, we use *discrete* functions based on x_i values where:

$$\begin{aligned}
x_0 &= a \\
x_1 &= a + h \\
x_2 &= a + 2h \\
&\vdots \\
x_i &= a = ih \\
&\vdots \\
x_N &= a + Nh = b \Rightarrow h = \frac{b-a}{N}
\end{aligned}$$

2.2.1 Euler's Method

Theorem 2.1. *The Euler Method of* 1 *step(explicit) is derived from the following* 1^{st} *order taylor expansion:*

$$y(x_{n+1}) = y(x_n + h) = y(x_n) + \frac{h}{1!}y'(x_n)$$

$$OR$$

$$\boxed{y(x_{n+1}) = y(x_n) + hf(x_n, y_n)}$$

Note that the 2 step(Implicit) Euler method is:

$$y(x_{n+1}) = y(x_n + h) = y(x_n) + \frac{h}{1!}y'(x_n) + \frac{h^2}{2!}f(x_n, y_n)$$

We shall not concern ourselves with the Implicit method
Here is the **Python** code for Euler method of 1 step:

```python
import numpy as np
import matplotlib.pyplot as plt
def f(x,y):
    return x*y #or any other function
a=1
b=2
N=50
h=(b-a)/N
x=np.zeros(N)
y=np.zeros(N)
y[0]=1 #pick any intial condition
x[0]=a #xo
for i in range(N-1):
    y[i+1]= y[i]+h*f(x[i],y[i])
    x[i+1]= x[0]+i*h

plt.plot(x,y)
plt.show()
```

The **MATLAB** code is:

```matlab
clear
clc
a=input('a= ');
b=input('b= ');
N=input('max= ');
h=(b-a)/N;
x(1)=a;
y(1)=1;
for i=1:N-1
    y(i+1)=y(i)+h*fct(x(i),y(i));
    x(i+1)=a+i*h;
end
plot(x,y)
```

2.2.2 Runge Kutta 2a

The Runge Kutta 2a or RK-2a method is derived from the **Midpoint** integration method:

Theorem 2.2. *Let $k_1 = hf(x_n, y_n)$ and $k_2 = hf(x_n + 0.5h, y_n + 0.5k_1)$. The RK-2a formula is:*

$$\boxed{y_{n+1} = y_n + k_2}$$

The **Python** code of RK-2a is:

```python
import numpy as np
import matplotlib.pyplot as plt
def f(x,y):
    return 10*x*y


a=float(input('a= '))
b=float(input('b= '))
N=int(input('max= '))
h=(b-a)/N
x=np.zeros(N)
y=np.zeros(N)
y[0]=1
x[0]=a
for i in range(N-1):
    k1=h*f(x[i],y[i])
    k2=h*f(x[i]+0.5*h,y[i]+0.5*k1)
    y[i+1]=y[i]+k2
    x[i+1]=a+i*h


plt.plot(x,y)
plt.show()
```

The **MATLAB** code is:(assume function is defined)

```
clear
clc
a=input('a= ');
b=input('b= ');
N=input('max= ');
h=(b-a)/N;
x(1)=a;
y(1)=1;
for i=1:N-1
    k1=h*fct(x(i),y(i));
    k2=h*fct(x(i)+0.5*h,y(i)+0.5*k1);
    y(i+1)=y(i)+k2;
    x(i+1)=a+i*h;
end
plot(x,y)
```

2.2.3 Runge Kutta 2b

The Runge Kutta 2b or RK-2b for short is:

Theorem 2.3. *Let $k_1 = hf(x_n, y_n)$ and $k_2 = hf(x_n + h, y_n + k_1)$. The RK-2b formula is:*

$$\boxed{y_{n+1} = y_n + 0.5(k_2 + k_1)}$$

The **Python** code is:

```python
import numpy as np
import matplotlib.pyplot as plt
def f(x,y):
    return x*y


a=float(input('a= '))
b=float(input('b= '))
N=int(input('max= '))
h=(b-a)/N
x=np.zeros(N+1)
y=np.zeros(N+1)
y[0]=1
x[0]=a
for i in range(N):
    k1=h*f(x[i],y[i])
    k2=h*f(x[i]+h,y[i]+k1)
    y[i+1]=y[i]+0.5*(k2+k1)
    x[i+1]=x[0]+i*h

plt.plot(x,y)
plt.show()
```

The **MATLAB** code is:(assuming function is defined)

```matlab
clear
clc
a=input('a= ');
b=input('b= ');
N=input('max= ');
h=(b-a)/N;
x(1)=a;
y(1)=1;
for i=1:N-1
    k1=h*fct(x(i),y(i));
    k2=h*fct(x(i)+h,y(i)+k1);
    y(i+1)=y(i)+0.5*(k1+k2);
    x(i+1)=a+i*h;
end
plot(x,y)
```

2.2.4 Runge Kutta 4

The Runge Kutta 4 or RK-4 method is:

Theorem 2.4. *Let* $k_1 = hf(x_n, y_n)$,$k_2 = hf(x_n + 0.5h, y_n + 0.5k_1)$, $k_3 = hf(x_n + 0.5h, y_n + 0.5k_2)$, *and* $k_4 = hf(x_n + h, y_n + k_3)$.

$$\boxed{y_{n+1} = y_n + (k_1 + 2(k_2 + k_3) + k_4)/6}$$

The **MATLAB** code is:(assuming you defined a function)

```matlab
clear
clc
a=input('a= ');
b=input('b= ');
N=input('max= ');
h=(b-a)/N;
x(1)=a;
y(1)=1;
for i=1:N-1
    k1=h*fct(x(i),y(i));
    k2=h*fct(x(i)+0.5*h,y(i)+0.5*k1);
    k3=h*fct(x(i)+0.5*h,y(i)+0.5*k2);
    k4=h*fct(x(i)+h,y(i)+k3);
    y(i+1)=y(i)+(k1+2*(k2+k3)+k4)/6;
    x(i+1)=a+i*h;
end
plot(x,y)
```

The **Python** code is:

```python
    import numpy as np
import matplotlib.pyplot as plt
def f(x,y):
    return x*y


a=float(input('a= '))
b=float(input('b= '))
N=int(input('max= '))
h=(b-a)/N
x=np.zeros(N+1)
y=np.zeros(N+1)
y[0]=1
x[0]=a
for i in range(N):
    k1=h*f(x[i],y[i])
    k2=h*f(x[i]+0.5*h,y[i]+0.5*k1)
    k3=h*f(x[i]+0.5*h,y[i]+0.5*k2)
    k4=h*f(x[i]+h,y[i]+k3)
    y[i+1]=y[i]+1/6*(k1+2*(k2+k3)+k4)
    x[i+1]=x[0]+i*h

plt.plot(x,y)
plt.show()
```

3 Problems

Problem 1. *A ball is launched vertically upward with an initial velocity* $v_0 = 50m/s$*.The motion is influenced by gravity and linear air resistance. The governing equation for the velocity after applying Newton's Second Law is:*

$$m\frac{dv}{dt} = -mg - kv$$

where:
$m = 1kg$
$k = 0.1kg/s$ *(coefficienct of air resistance)*
$g = 9.81m/s^2$ *1) Find the plot of* $v(t)$ *by solving the ODE while using the RK4 method*
2) Find the general solution(symbolic) of the ODE
3)After finding the symbolic solution, find the time t *in which* $v(t) = 0$

Solution. *1) Rewrite the the ODE in the form* $\frac{dv}{dt} = f(v, t)$*:*

$$\frac{dv}{dt} = -g - \frac{kv}{m}$$

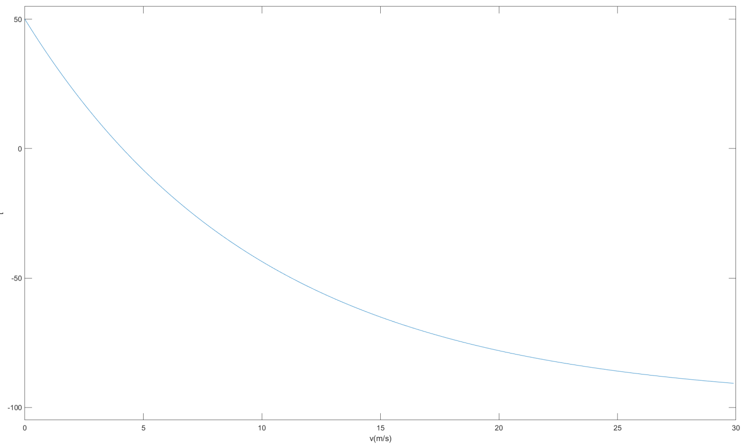
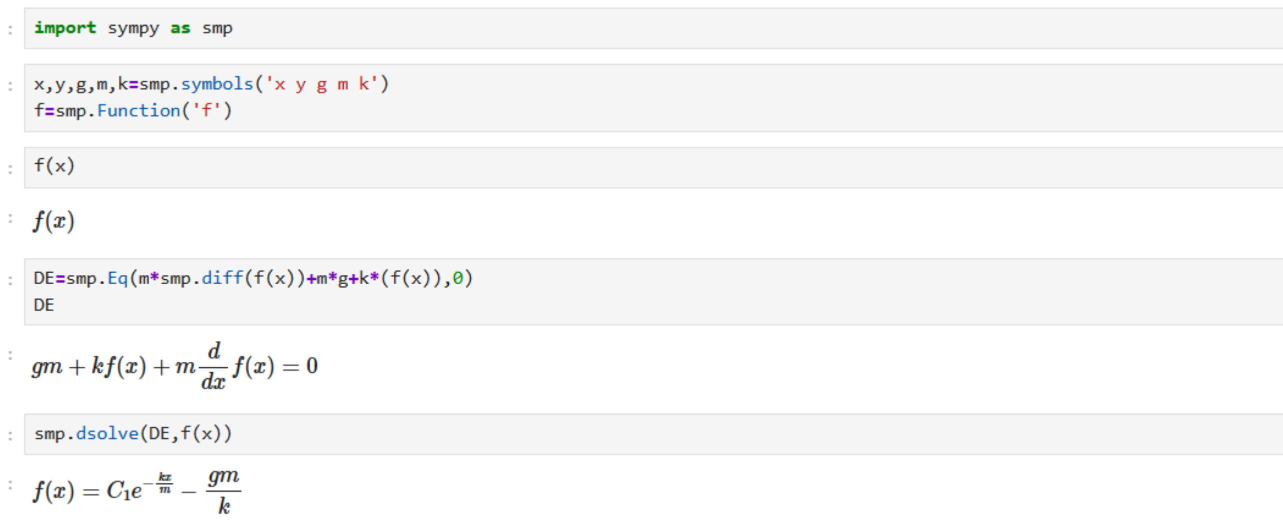


Figure 1: Plot generated from MATLAB.

2)
In terms of v *and* t*:*

$$v(t) = C_1 e^{-\frac{kt}{m}} - \frac{mg}{k}$$



```
import sympy as smp
```

```
x,y,g,m,k=smp.symbols('x y g m k')
f=smp.Function('f')
```

```
f(x)
```

$f(x)$

```
DE=smp.Eq(m*smp.diff(f(x))+m*g+k*(f(x)),0)
DE
```

$gm + kf(x) + m\frac{d}{dx}f(x) = 0$

```
smp.dsolve(DE,f(x))
```

$f(x) = C_1 e^{-\frac{kx}{m}} - \frac{gm}{k}$

Figure 2: Plot generated from Jupyter Notebook.

Knowing at v(0)=50:

$$\boxed{v(t) = (50 + \frac{mg}{k})e^{-\frac{kt}{m}} - \frac{mg}{k}}$$

3)To find the roots of $v(t)$, we can use the bisection method:
This function only has one root which is $t = 4.119003$(check it by plotting)

Problem 2. *Assume the* ***Temperature as a function of time*** *of an extraterrestial object during the day varies according to the following equation:*

$$T(t) = T_{surr} + (T(0) - T_{surr})e^{-kt} + 5sin(0.1t)$$

Where:
$T_{surr} = 20°\text{C}$ *is the average surrounding temperature*
Time in min
$k = 0.01/min$
$T(0) = 13°\text{C}$
1) Find at which time ***t*** *where the temperature is equal to the* ***surrounding*** *temperature between $t \in [40, 60]$.*
By DERIVING both sides of the equation wrt to time, we get the following non-linear ODE:

$$\frac{dT}{dt} = -k(T(0) - T_{surr})e^{-kt} + 0.5cos(0.1t)$$

2) Plot the solution to the following differential equation by using the euler method.

Solution. *1)We have to solve the following equation:*

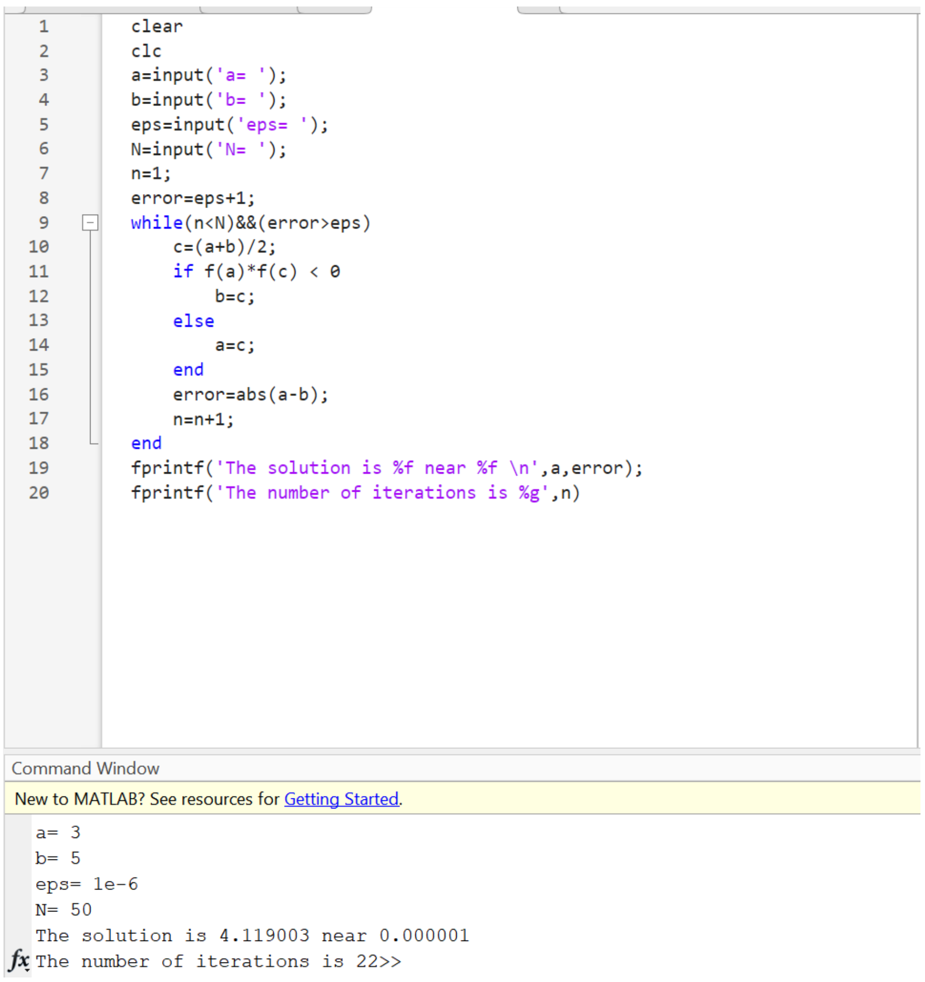
$$T(t) = T_{surr} + (T(0) - T_{surr})e^{-kt} + 5sin(0.1t) = 13$$

Shifting so that we get an equation equals to zero:

$$T(t) = T_{surr} + (T(0) - T_{surr})e^{-kt} + 5sin(0.1t) - 13 = 0$$

By using the Newton-Raphson method(fig 4)

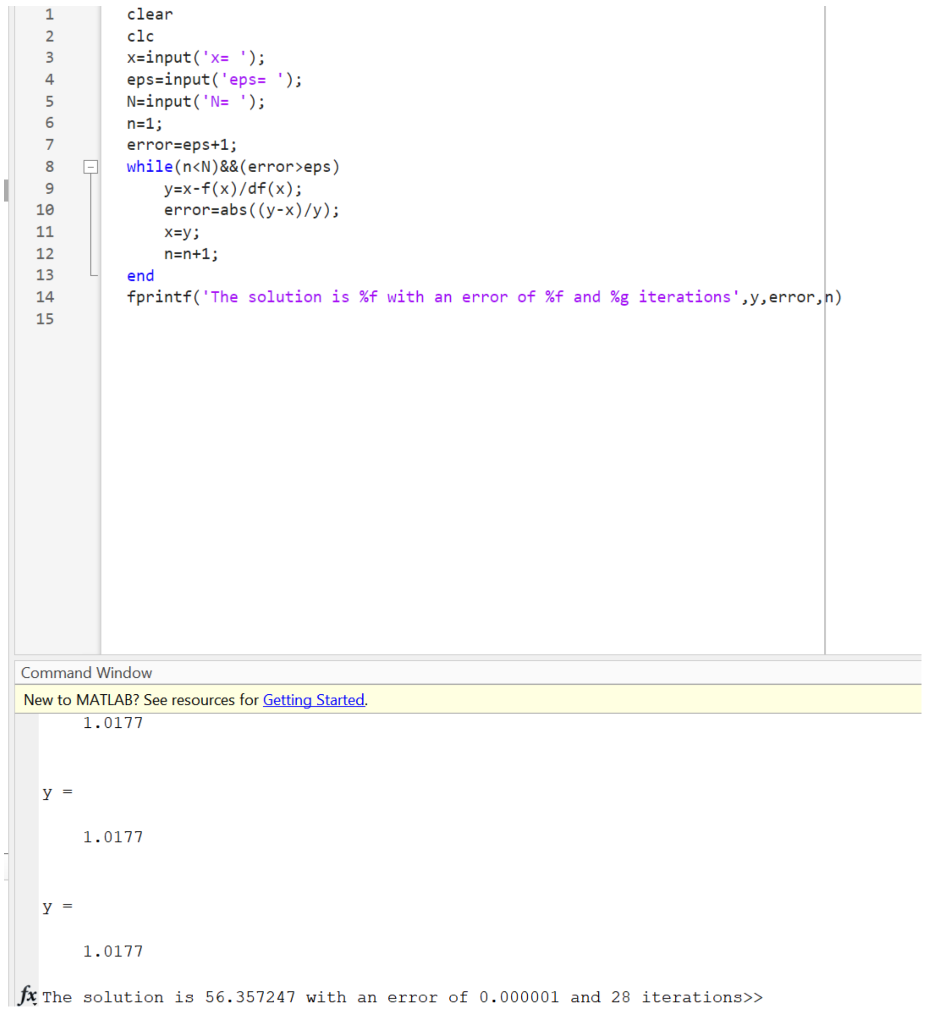
2)We want to solve this very simple ODE numerically(fig 5):



```
clear
clc
a=input('a= ');
b=input('b= ');
eps=input('eps= ');
N=input('N= ');
n=1;
error=eps+1;
while(n<N)&&(error>eps)
    c=(a+b)/2;
    if f(a)*f(c) < 0
        b=c;
    else
        a=c;
    end
    error=abs(a-b);
    n=n+1;
end
fprintf('The solution is %f near %f \n',a,error);
fprintf('The number of iterations is %g',n)
```

```
a= 3
b= 5
eps= 1e-6
N= 50
The solution is 4.119003 near 0.000001
The number of iterations is 22>>
```

Figure 3: Plot generated from MATLAB.



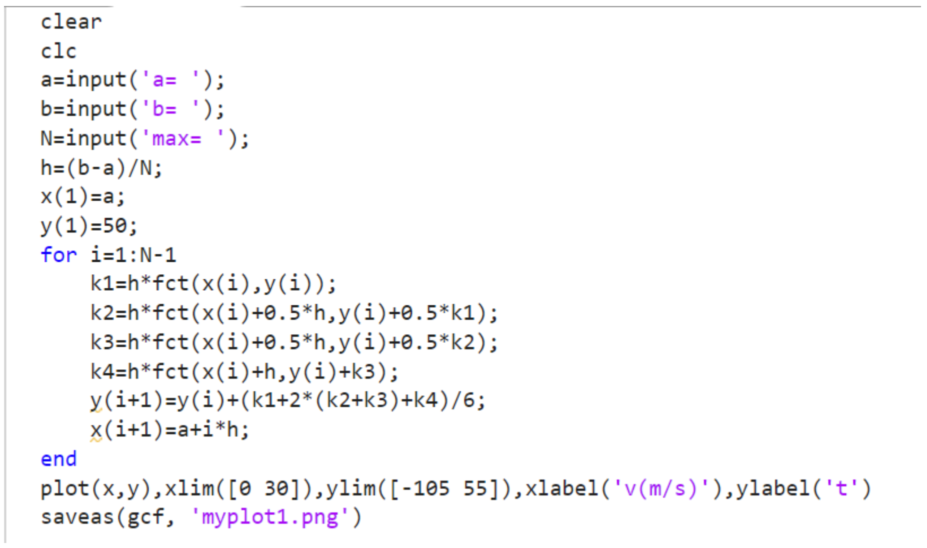
```
clear
clc
x=input('x= ');
eps=input('eps= ');
N=input('N= ');
n=1;
error=eps+1;
while(n<N)&&(error>eps)
    y=x-f(x)/df(x);
    error=abs((y-x)/y);
    x=y;
    n=n+1;
end
fprintf('The solution is %f with an error of %f and %g iterations',y,error,n)
```

Command Window

New to MATLAB? See resources for Getting Started.

```
    1.0177


y =

    1.0177


y =

    1.0177

The solution is 56.357247 with an error of 0.000001 and 28 iterations>>
```

Figure 4: Solving temperature equation



```
clear
clc
a=input('a= ');
b=input('b= ');
N=input('max= ');
h=(b-a)/N;
x(1)=a;
y(1)=50;
for i=1:N-1
    k1=h*fct(x(i),y(i));
    k2=h*fct(x(i)+0.5*h,y(i)+0.5*k1);
    k3=h*fct(x(i)+0.5*h,y(i)+0.5*k2);
    k4=h*fct(x(i)+h,y(i)+k3);
    y(i+1)=y(i)+(k1+2*(k2+k3)+k4)/6;
    x(i+1)=a+i*h;
end
plot(x,y),xlim([0 30]),ylim([-105 55]),xlabel('v(m/s)'),ylabel('t')
saveas(gcf, 'myplot1.png')
```

Figure 5: Euler Method