Root FindinG METHODS

Numerical Computing

PREPARED BY BUSHRA (63759)  
C.ID 105126

PAF-KIET

UNDER SUPERVISON: SIR SAAD AKBER

Contents

[**INTRODUCTION** 2](#_Toc57336046)

[**NUMERICAL METHODS VS. ANALYTICAL METHODS** 2](#_Toc57336047)

[**ITERATIVE METHODS FOR ROOT FINDING** 3](#_Toc57336048)

[**Bisection Method** 3](#_Toc57336049)

[**Regula Falsi Method (False Position Method)** 4](#_Toc57336050)

[**Secant Method** 4](#_Toc57336051)

[**Newton Raphson Method** 5](#_Toc57336052)

[**Muller Method** 5](#_Toc57336053)

[**Comparison** 6](#_Toc57336054)

[**All Code** 6](#_Toc57336055)

# **INTRODUCTION**

Let a number such that f(ξ) is 0 i.e. f(ξ)=0 is said to be the root of an equation as f(x) is zero.

As per study, Root finding problem is considered as a classical problem. Though the function can be algebraic, logarithmic, trigonometric and their combination i.e. Transcendental Equations.

## **NUMERICAL METHODS VS. ANALYTICAL METHODS**

It is not possible to find the expression for root (analytical) from where the root is exactly determined, except some specified functions.

The **Fundamental Theorem of Algebra** is stated as:

*A polynomial Pn (x) of degree n(n≥1) has at least one zero.*

At early we used to learn how to solve a quadratic equation using analytical formula.

But, such analytical formulas fail for degree higher to 5 and this was proved by **ABEL & GALOIS**.

So, For Root finding methods i.e. computational methods we must require an iterative approach.

When starts from an initial approximation 0, we generate a sequence to k using an iterative method in order to have a convergence to a root i.e. f(x) = 0  
 **For Iterative Method**

These are the two aspects for iterative method

* Convergence
* Stopping Criteria

**Stopping Criteria**

We can say a root is k at some iteration if it will satisfy the below conditions:

* Functional value less than or equal to tolerance
* Relative change less than or equal to tolerance
* The iterations processed to the given limit

**Intermediate Value Property**

If a function f(x) is real valued continuous function in the closed interval . If f(a) and f(b) have opposite signs that is f(x) = 0 has at least one root such that .

Example

# **ITERATIVE METHODS FOR ROOT FINDING**

y-axis

a

b

x-axis

Here exists one root!

* Bisection Method
* Regula Falsi Method (False Position Method)
* Newton Raphson
* Secant Method
* Muller Method

These are the methods implemented and compared in the project in python (Code and Graph).

# **Bisection Method**

Bisection itself is derived from a word bisect means to divide into 2 equal parts.

**Idea:**

Let f(x)=0 have a real root in and interval of [a,b].

Then , where c is the middle point. If c is the root, then we are done we don’t need t proceed further, but if not, then [a,c] or[c,b] contain the root.

Again bisect into equal intervals and this will be continued until the root is trapped in an interval with desired accuracy.

**Algorithm**

**Inputs**

* f(x) is the function
* a, b (two initial guesses) but these two must satisfy IVP

i.e. f(a)\*f(b) < 0, So Root Exists.

**Output**

start

for i 0, 1, 2, … , n

* compute
* test if c is the desired root then stop
* if not then f(a) \* f(c) > 0 so c🡪a or if f(a) \* f(c) < 0 so c🡪b

end

# **Regula Falsi Method (False Position Method)**

In bisection method, every interval under consideration has a guaranteed desired root that is why it was referred as BRACKET METHOD.

Thus to find a real root of f(x) = 0 using Regula Falsi Method, we replace the part of the curve between the points A [n-1]and B[n-1)] by a chord in that interval and we take the point of intersection.  
 **Algorithm**

**Input:**

f(x) - The given function x0, x1

The two initial approximations f(x0)\*f(x1) < 0

E - The tolerance

N - The maximum number of iterations

**Output:**

An approximation to the root x = ξ.

Set i = 1.

Compute

Continue until🡪 If | (xi+1−xi )/xi | < T or i ≥ N, stop.

End

# **Secant Method**

This method is advanced of Newton Raphson Method. Since in Newton method its disadvantage was we always required the first derivativeat each approximation. So this consideration leads to Secant Method.

**Algorithm**

**Inputs:**

f(x) - The given function x0, x1 (The two initial approximations of the root)

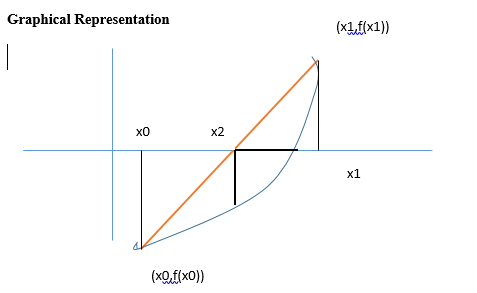
T - The error tolerance

N - The maximum number of iterations

**Output**

Start

For k = 1, 2, · · · , n (until the stopping criteria is met)

Compute f(xk) and f(xk−1).

Compute the next approximation:

xk+1 = xk-1 − f(xk)- f(xk-1)\*xk/ f(xk)- f(xk-1) .

Test for convergence or maximum number of iterations:

If |xk+1 − xk| < T or if k > N,

Stop.

End

In secant Method we do not check whether the root lies between two successive approximations X n-1 and x n.

# **Newton Raphson Method**

In the Newton Raphson Method, the root is not bracketed. Only one initial guess is required to get started with the iterative procedure and find the root of a nonlinear equation.

Hence the method falls in the category of OPEN METHOD. It is also called as Tangents Method.

Tangent

y-axis

x-axis

X2 x x1

Slope = f(x1)/ x

X = f(x1)/Slope x=(f(x1)/f’(x1))

x-x2= (f(x1)/f’(x1)) So, x2= x1-(f(x1)/f’(x1))

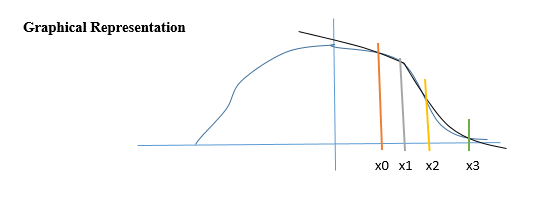
General Formula: xn=xn-1-(f(x n-1)/f’(x n-1))

# **Muller Method**

This method is an extension to the secant method. It is used to approximate a complex root of a real equation; this method takes 3 initial guesses.

Neither Secant nor newton Raphson can produce an approximation of complex root starting from real approximations but Muller can do using real approximations or just a real root.

Thus computing x3 from the starting approximations that we have is (x0, x1, x2),

we perform:

xr=x3=x0 -

where a=

b = , c= y0

if xr<x0

x0🡨x3

x1🡨x0

x2=x2

if xr>x0

x0🡨x3

x1=x1

x2🡨x0

h1=x1-x0, h2=x0-x2 and h =   
But conditions further are 🡪  
  
(Computing x4 and successive approximations.)

# **Comparison**

|  |  |  |  |
| --- | --- | --- | --- |
| Method | Required | Convergence | Remarks |
| Bisection | 2 initial guesses | Always Converge (but slow) | Bracketing method-every iteration requires brackets the root and initial interval is must at the beginning. |
| Regula falsi | 2 initial guesses | Always Converge (faster than bisection) | Bracketing method-every iteration requires brackets the root and derivative is required. |
| Newton | 1 initial guess | Convergence is linear at multiple root | Initial approximation taken carefully else will diverge.  Derivative is difficult to compute |
| Secant | 2 initial guesses | Super linear | Needs 2 initial guesses |
| Muller | 3 initial guesses | Convergence is almost quadratic near root | Can compute complex root without requirement of derivative. |

# **All Code**

from numpy.lib.scimath import sqrt  
from tkinter import \*  
from sympy import var  
from sympy import sympify  
from sympy.utilities.lambdify import lambdify  
import math  
import numpy as np  
import matplotlib.pyplot as plt  
  
window = Tk()  
window.title(**"Numerical Computing - Root Finding Methods"**)  
window.geometry(**'650x550'**)  
window.resizable(width=False, height=False)  
  
from sympy import \*  
global x  
global expr  
def funcinput(y):  
 x = var(**'x'**)  
 expr = sympify(txt0.get())  
 f = lambdify(x, expr)  
 value=f(y)return float(value)  
  
def derivFunc(y):  
 x = var(**'x'**)  
 expr = sympify(txt0.get())  
 fd=expr.diff(x)  
 fdd=lambdify(x,fd)  
 value= fdd(y)  
 return float(value)  
  
def Newton():  
 iter=int(txt1.get())  
 x=float(txt2.get())  
 E = 0.0001  
 print(**"----------Newton Rephson-------------"**)  
 global NR  
 newtonCount = 0  
 mylist = []  
 h = float(funcinput(x) / derivFunc(x))  
 while (abs(h) >= E):  
  
 if (derivFunc(x) == 0):  
 print(**"Not Possible"**)  
 break  
 else:  
 h = float( funcinput(x) / derivFunc(x))  
 mylist.append(h)  
 x = x - h  
 newtonCount = newtonCount + 1  
 if (newtonCount > iter):  
 break  
 if (newtonCount > iter):  
 print(**"Could Not Found Solution even in "**, iter, **"iterations"**)  
 NR = 0  
 else:  
 NR = newtonCount  
 print(**"The value of the root By Newton rephson method is : "**, **"%.4f"** % x)  
 print(**"Counts : "**, newtonCount)  
  
 plt.plot(mylist, color=**'g'**)  
 plt.xlabel(**'No. of iteration'**)  
 plt.ylabel(**'Tolerence'**)  
  
 plt.title(**'Graph - NewtonRephson'**)  
 plt.show()  
 return  
def regula():  
 print(**"----------Regula Falsi-------------"**)  
 a=float(txt2.get())  
 b = float(txt3.get())  
 N=int(txt1.get())  
 TOL=0.0001  
 global RF  
 mylist = []  
 i = 1  
 FA = funcinput(a)  
  
 if (funcinput(a) \* funcinput(b) < 0):  
  
 while True:  
 p = (a \* funcinput(b) - b \* funcinput(a)) / (funcinput(b) - funcinput(a))  
 FP = funcinput(p)  
  
 if (FP == 0 or np.abs(funcinput(p)) < TOL):  
 break  
 mylist.append(np.abs(funcinput(p)))  
  
 i = i + 1  
  
 if (FA \* FP > 0):  
 a = p  
 else:  
 b = p  
  
 if (i > N):  
 break  
  
 if (i > N):  
 print(**"Could Not Found Solution even in "**, N, **"iterations"**)  
 RF = 0  
 else:  
 print(**"The value of root is : "**, **'%.4f'** % a)  
 print(**"Count : "**, i)  
 RF = i  
 else:  
 print(**"Here f(a)>0 and f(b)>0 So, Solution is not possible. Try Again..."**)  
  
 plt.plot(mylist, color=**'g'**)  
 plt.xlabel(**'No. of iteration'**)  
 plt.ylabel(**'Tolerance'**)  
 plt.title(**'Graph - Regula Falsi'**)  
 plt.show()  
 return  
import pandas as pd  
import math  
def muller():  
 print(**"----------Muller Method-------------"**)  
 a = float(txt2.get())  
 b = float(txt3.get())  
 c=float(txt4.get())  
 iter = int(txt1.get())  
 TOL = 0.0001  
 mylist = []  
 res = 0  
 i = 0  
 global MM  
 while (True):  
 f1 = funcinput(a)  
 f2 = funcinput(b)  
 f3 = funcinput(c)  
 d1 = f1 - f3  
 d2 = f2 - f3  
 h1 = a - c  
 h2 = b - c  
 a0 = f3  
 a1 = (((d2 \* pow(h1, 2)) - (d1 \* pow(h2, 2))) / ((h1 \* h2) \* (h1 - h2)))  
 a2 = (((d1 \* h2) - (d2 \* h1)) / ((h1 \* h2) \* (h1 - h2)))  
 x = ((-2 \* a0) / (a1 + abs(math.sqrt(a1 \* a1 - 4 \* a0 \* a2))))  
 y = ((-2 \* a0) / (a1 - abs(math.sqrt(a1 \* a1 - 4 \* a0 \* a2))))  
  
 if (x >= y):  
 res = x + c  
 else:  
 res = y + c  
  
 m = res \* 100  
 n = c \* 100  
 m = math.floor(m)  
 n = math.floor(n)  
if abs(res - c) < 0.0001:  
 break  
  
 mylist.append(abs(res - c))  
 a = b  
 b = c  
 c = res  
 if (i > iter):  
 print(**'Root cannot be found using Muller’s method'**)  
 MM = 0  
 break  
  
 i += 1  
 if (i <= iter):  
 print(**"The value of the root is {0} in {1} iterations"**.format(round(res, 4), i))  
 MM=i  
 plt.plot(mylist, color=**'g'**)  
 plt.xlabel(**'No. of iteration'**)  
 plt.ylabel(**'Tolerance'**)  
 plt.title(**'Graph - Muller'**)  
 plt.show()  
 return  
  
def all\_com():print(**" "**)  
 print(**" --- "**)  
 print(**" "**)  
 bisection()  
 Newton()  
 regula()  
 Secant()  
 muller()  
 draw\_Bar\_Graph()  
 draw\_Graph()  
  
def draw\_Bar\_Graph():  
 s = pd.Series(  
 [BM, RF, NR, SC,MM],  
 index=[**'Bisection Method'**, **'Regula Falsi'**, **'Newton Rephson'**, **'Secant Method'**,**'Muller Method'**]  
 )  
  
 plt.title(**"Comparision of Root Finding Methods"**)  
 plt.ylabel(**'No of Iteration'**)  
 plt.xlabel(**'Method Name'**)  
  
 ax = plt.gca()  
 ax.tick\_params(axis=**'x'**, colors=**'blue'**)  
 ax.tick\_params(axis=**'y'**, colors=**'red'**)  
  
 my\_colors = **'Green'** s.plot(  
 kind=**'bar'**,  
 color=my\_colors,  
  
 )  
 plt.show()  
 return  
def draw\_Graph():  
 MethodName = [**'Bisection Method'**, **'Regula Falsi'**, **'Newton Rephson'**, **'Secant Method'**,**'Muller Method'**]  
  
 Count = [BM, RF, NR, SC,MM]  
 plt.plot(MethodName, Count, color=**'g'**)  
 plt.xlabel(**'Method Name'**)  
 plt.ylabel(**'No. of iteration'**)  
 plt.title(**'Comparision of root finding Methods'**)  
 plt.show()  
 return  
  
def Secant():  
 x1=float(txt2.get())  
 x2 = float(txt3.get())  
 iter = int(txt1.get())  
 E = 0.0001  
 print(**"----------Secant-------------"**)  
 global SC  
 n = 0  
 xm = 0.0  
 x0 =0.0  
 c = 0.0  
 mylist = []  
 if ((funcinput(x1) \* funcinput(x2) )< 0):  
 while True:  
 x0 = ((x1 \* funcinput(x2) - x2 \* funcinput(x1)) /(funcinput(x2) - funcinput(x1)))  
 c = funcinput(x1) \* funcinput(x0)  
 x1 = x2  
 x2 = x0  
 n += 1  
 if (n > iter):  
 break  
 if (c == 0):  
 break  
 xm = ((x1 \* funcinput(x2) - x2 \* funcinput(x1)) /(funcinput(x2) - funcinput(x1)))  
 if (abs(xm - x0) < E):  
 break  
 mylist.append(abs(xm - x0))  
 SC = n  
 if (n > iter):  
 print(**"Could Not Found Solution even in "**, iter, **"iterations"**)  
 SC = 0  
 else:  
 print(**"Root of the given equation ="**,round(x0, 4))  
 print(**"No. of iterations = "**, n)  
 else:  
 print(**"Here f(a)>0 and f(b)>0 So, Solution is not possible. Try Again..."**)  
  
 plt.plot(mylist, color=**'g'**)  
 plt.xlabel(**'No. of iteration'**)  
 plt.ylabel(**'Tolerence'**)  
 plt.title(**'Graph - Secant'**)  
 plt.show()  
 return  
  
def bisection():  
  
 a=float ( txt2.get())  
 b=float ( txt3.get())  
 iter=int ( txt1.get())  
 E=0.0001  
 print(**"----------Bisection-------------"**)  
 global BM  
 bisectionCount = 0  
 mylist = []  
 if (funcinput(a) \* funcinput(b) >= 0):  
 print(**"f(a) = "**,funcinput(a))  
 print(**"f(b) = "**,funcinput(b))  
 print(**"Here f(a)>0 and f(b)>0 So, Solution is not possible. Try Again..."**)  
 return  
 c = a  
 while ((b - a) >= E):  
 mylist.append((b - a))  
 c = (a + b) / 2  
 if (funcinput(c) == 0.0):  
 break  
 if (funcinput(c) \* funcinput(a) < 0):  
 b = c  
 else:  
 a = c  
 bisectionCount = bisectionCount + 1  
 if (bisectionCount > iter):  
 break  
 if (bisectionCount > iter):  
 BM = 0  
 print(**"Could Not Found Solution even in "**, iter, **"iterations"**)  
 else:  
 BM = bisectionCount  
 print(**"The value of root is : "**, **"%.4f"** % c)  
 print(**"Counts : "**, bisectionCount)  
  
 plt.plot(mylist, color=**'g'**)  
 plt.xlabel(**'No. of iteration'**)  
 plt.ylabel(**'Tolerence'**)  
 plt.title(**'Graph - Bisection'**)  
 plt.show()  
 return  
  
lbl = Label(window, text=**"Root Finding Methods"**,fg=**'Green'**,font=(**'Helvetica'**, 22, **'bold'**))  
lbl.place(x=180 ,y=30)  
  
lbl = Label(window, text=**"Instructions:"**,fg=**'Blue'**,font=**"Calibiri"**)  
lbl.place(x=0 ,y=90)  
lbl = Label(window, text=**"For Bisection, Regula Falsi, Secant fill(a and b) only "**,fg=**'Black'**,font=**"Calibiri"**)  
lbl.place(x=0 ,y=112)  
lbl = Label(window, text=**"For Muller Method fill a,b,c (all) and for Newton Raphson Method fill Only a"**,fg=**'Black'**,font=**"Calibiri"**)  
lbl.place(x=0 ,y=132)  
  
lbl= Label(window, text=**"Enter Equation : "**)  
lbl.place(x=15 ,y=180)  
lbl.config(bg=**"white"**)  
  
txt0 =Entry(window, bd=5, width=**"30"**)  
txt0.place(x=115, y=180)  
  
lbl = Label(window, text=**"! Only Linear/Trigonometric "**,fg=**'Red'**)  
lbl.place(x=315, y=180)  
  
lbl = Label(window, text=**"Max Iterations"**)  
lbl.place(x=15, y=220)  
lbl.config(bg=**"white"**)  
  
txt1 = Entry(window, bd=5, width=**"30"**)  
txt1.place(x=115, y=220)  
  
lbl = Label(window, text=**"Enter a"**)  
lbl.place(x=15, y=260)  
lbl.config(bg=**"white"**)  
lbl = Label(window, text=**"! Only 1 Initial Guess for Newton Raphson "**,fg=**'Blue'**)  
lbl.place(x=315, y=260)  
txt2 = Entry(window, bd=5, width=**"30"**)  
txt2.place(x=115, y=260)  
lbl = Label(window, text=**"Enter b"**)  
lbl.place(x=15, y=300)  
lbl.config(bg=**"white"**)  
txt3 = Entry(window, bd=5, width=**"30"**)  
txt3.place(x=115, y=300)  
lbl = Label(window, text=**"Enter c"**)  
lbl.place(x=15, y=340)  
lbl.config(bg=**"white"**)  
lbl = Label(window, text=**"! All 3 Initial Guesses for Muller Method "**,fg=**'Blue'**)  
lbl.place(x=315, y=340)  
txt4 = Entry(window, bd=5, width=**"30"**)  
txt4.place(x=115, y=340)  
btn = Button(window, text=**"Bisection"**, height=**"2"**, width=**"20"**, command=bisection)  
btn.place(x=50, y=390)  
btn = Button(window, text=**"Newton"**, height=**"2"**, width=**"20"**, command=Newton)  
btn.place(x=250, y=390)  
btn = Button(window, text=**"Secant"**, height=**"2"**, width=**"20"**, command=Secant)  
btn.place(x=450, y=390)  
btn = Button(window, text=**"Regula Falsi"**, height=**"2"**, width=**"20"**, command=regula)  
btn.place(x=50, y=460)  
btn = Button(window, text=**"Muller"**, height=**"2"**, width=**"20"**, command=muller)  
btn.place(x=250, y=460)  
btn = Button(window, text=**"All Compare"**, height=**"2"**, width=**"20"**, command=all\_com)  
btn.place(x=450, y=460)  
window.mainloop()