Minimization

Arunima Sarkar, EE21S062

1 Finding roots in 1-D

The following function is given whose roots are to be determined.

$$\tan x = \sqrt{\pi\alpha - x} \tag{1}$$

Thus the function $f(x)$ can be written as:

$$f(x) = \tan x - \sqrt{\pi\alpha - x} \tag{2}$$

1.1 Left and Right sides

The points where both the function overlaps are the roots of the function.

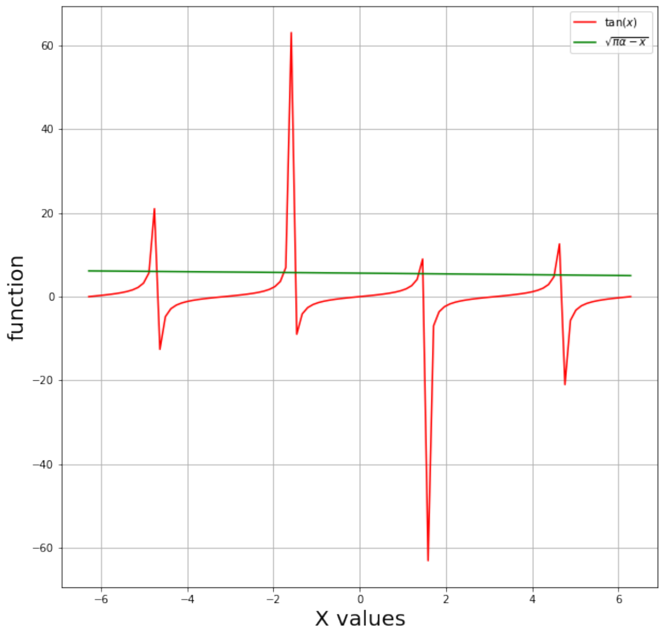

Figure 1: Plot for left and right side of function

The code for it is,

```
def func_left(x):
    return np.tan(x)

def func_right(x):
    alpha =10
    return np.sqrt(np.pi*alpha-x)

xval= np.linspace(-2*np.pi,2*np.pi,100)
yval_left=func_left(xval)
yval_right=func_right(xval)
```

1.2 Bracketing the roots

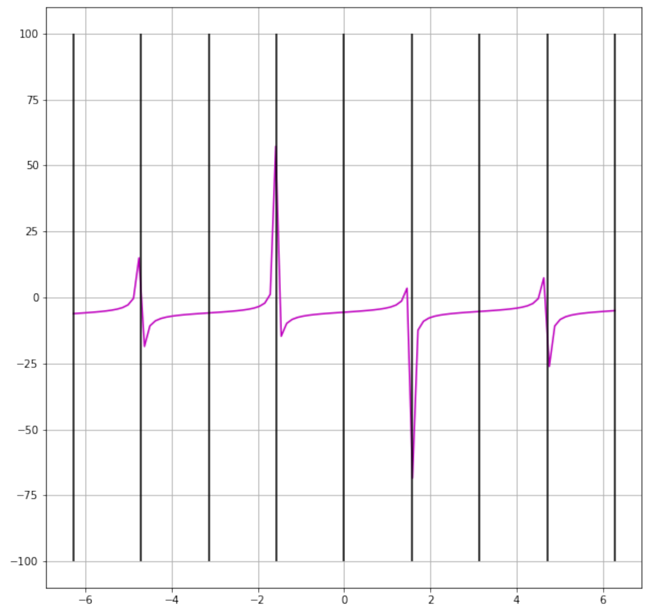

Figure 2: Bracketing the regions

The bracketing regions are: $\left(-\pi, \frac{-\pi}{2}\right)$, $\left(\frac{-\pi}{2}, 0\right)$, $\left(0, \frac{\pi}{2}\right)$, $\left(\pi, \frac{3\pi}{2}\right)$ and so on.

1.3 Bisection, Brent, Newton Raphson

The code for the above mentioned methods are,

```
def bisection(func,a,b,tol,n):
    assert func(a)*func(b)<0
    c=(a+b)/2
    n=n+1
    if(func(c)==0 or (b-a)/2<tol):
        return n
    elif (np.sign(func(a))==np.sign(func(c))):
        return bisection(func,c,b,tol,n)
    else:
        return  bisection(func,a,c,tol,n)

def brents(f, x0, x1, max_iter=50, tolerance=1e-6):

    fx0 = f(x0)
    fx1 = f(x1)

    assert (fx0 * fx1) <= 0, "Root not bracketed"

    if abs(fx0) < abs(fx1):
        x0, x1 = x1, x0
        fx0, fx1 = fx1, fx0

    x2, fx2 = x0, fx0

    mflag = True
    steps_taken = 0

    while steps_taken < max_iter and abs(x1-x0) > tolerance:
        fx0 = f(x0)
        fx1 = f(x1)
        fx2 = f(x2)

        if fx0 != fx2 and fx1 != fx2:# Inverse Quadratic Interpolation
            L0 = (x0 * fx1 * fx2) / ((fx0 - fx1) * (fx0 - fx2))
            L1 = (x1 * fx0 * fx2) / ((fx1 - fx0) * (fx1 - fx2))
            L2 = (x2 * fx1 * fx0) / ((fx2 - fx0) * (fx2 - fx1))
            new = L0 + L1 + L2

        else:
            new = x1 - ( (fx1 * (x1 - x0)) / (fx1 - fx0) )

        if ((new < ((3 * x0 + x1) / 4) or new > x1) or
            (mflag == True and (abs(new - x1)) >= (abs(x1 - x2) / 2)) or
            (mflag == False and (abs(new - x1)) >= (abs(x2 - d) / 2)) or
            (mflag == True and (abs(x1 - x2)) < tolerance) or
            (mflag == False and (abs(x2 - d)) < tolerance)):
            new = (x0 + x1) / 2
            mflag = True

        else:
            mflag = False

        fnew = f(new)
        d, x2 = x2, x1

        if (fx0 * fnew) < 0:
            x1 = new
        else:
            x0 = new

        if abs(fx0) < abs(fx1):
            x0, x1 = x1, x0

        steps_taken += 1

    return x1, steps_taken

def discrete_method_approx(f, x, h=.00000001):
    return (f(x+h) - f(x)) / h

def newton_raphson(f, x, tolerance):
    steps_taken = 0

    while abs(f(x)) > tolerance:
        df = discrete_method_approx(f,x)
        x = x - f(x)/df
        steps_taken += 1
    return x, steps_taken
```

The number of functions calls for Bisection, Newton and Brent's respectively are (22, 14, 11) for $\alpha = 10$. However varying alpha wont change the answer. Upon inspecting Bisection, its linear convergence property makes the number of iterations higher for a certain accuracy. Newton-Raphson gives moderate amount of iterations and Brent perform best.
As stated in Numerical Recipes in C, Brent's method combines root bracketing, bisection, and inverse quadratic interpolation to converge from the neighborhood of a zero crossing, thus it just takes 11 iterations to converge to the given tolerance.

2 Minimization in 1-D

The function to be minimized,

$$f(x) = \sin x + \frac{1}{1+x^2} \tag{3}$$

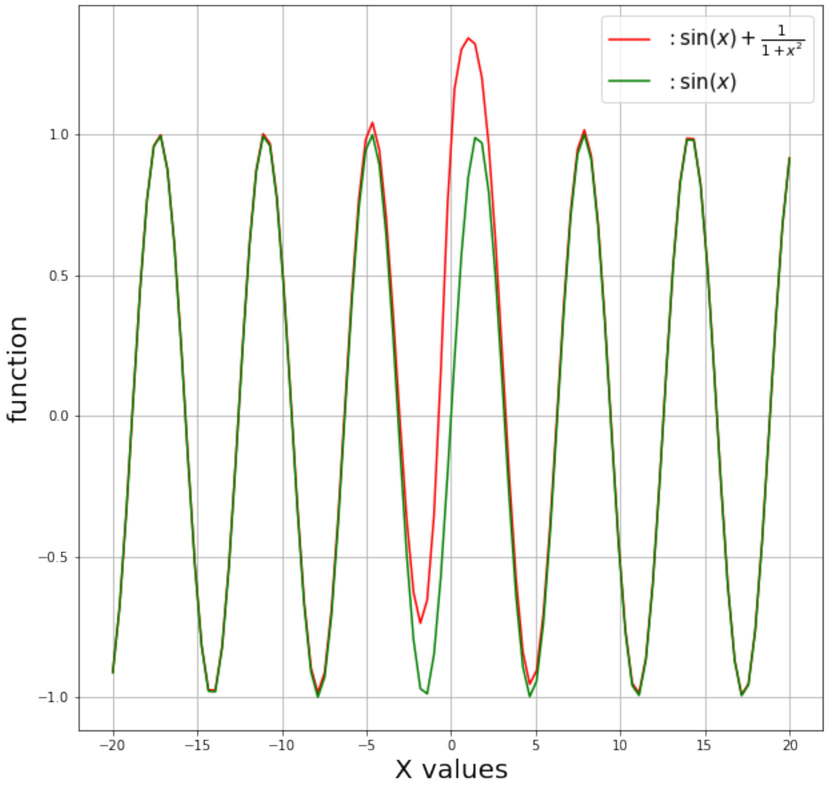

Figure 3

2.1 Bracketing the roots

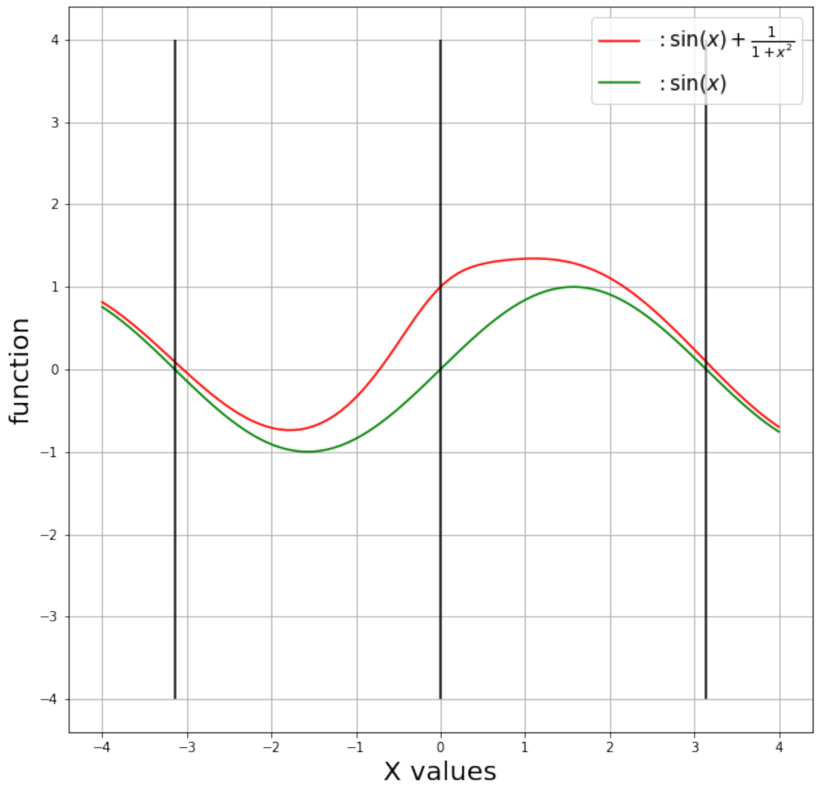

Figure 4

We can see that by bracketing the roots among the zeros of $\sin x$ we end up bracketing one minima of the f(x) among $[-\pi, 0]$.The assumption is correct.

2.2 Golden section search and Brent's Methods

```
def golden(f,xlower, xupper, tol, max_iter):
    count = 0
    while (xupper-xlower)>tol and count<max_iter:
        x_r1 = xupper - 0.61803399*(xupper - xlower)
        x_r2 = xlower + 0.61803399*(xupper - xlower)
        #print(count, xlower, x_r1, x_r2, xupper)
        if f(x_r1) <= f(x_r2):
            xupper = x_r2
            x_r2 = x_r1
            x_r1 = xupper - 0.61803399*(xupper - xlower)
        else:
            xlower = x_r1
            x_r1 = x_r2
            x_r2  = xlower + 0.61803399*(xupper - xlower)
        count = count + 1
    return (x_r1 + x_r2)/2.0, count

_,n_golden_section=golden(func_1D,-np.pi,0,10**(-8),500)

_,_,n_brent,_=brent(func_1D,brack=(-np.pi,0),full_output=1)
```

Upon applying Golden-Section search and Brents method we observe that n_golden_section is 41 and n_brent is 9. Rest analysis is yet to write.

2.3 Finding root using derivative information

The corresponding code,

```
def derivative_func_1D(x):
    return np.cos(x)-2*x/((1+x**2)**2)

n=0
ans_bisection=bisection(derivative_func_1D,-np.pi,0,10**(-8),n)

#Applying Brents on the derivative
_,ans_brents=brents(derivative_func_1D,-np.pi,0)

#Applying Newton-Raphson on the derivative
_,ans_newton=newton_raphson(derivative_func_1D,-1,10**(-8))
```

For Bisection it takes 29 iterations, for brent it takes 24 iterations and Newton Raphson takes 3 iterations