Project report (2)

Name (300493343)

Question 1

Introduction

The purpose of this project was to explore different methods of solving for the root of an equation. Some of which go on indefinitely. (Some of the print() have been removed in order to keep this document looking neat)

Procedure

Here you need to describe what you did to solve this question, include proofs, algorithms etc...

Here you need to include your code as well...

In [1]:
```
import numpy as np

#question 1a and 1b

def bisection(f, a, b, nmax = 50, tol = 1.e-6):
    iteration = 0
    #a = xn
    #b = xn+1
    if (f(a) * f(b) < 0.0):
        while ((b-a) > tol and iteration < nmax):
            iteration += 1;
            if(iteration == 1 and abs(b-a) <= tol):
                return b;
            x = (a + b)/2
            if (f(a) * f(x) < 0.0):
                b = x
            elif (f(b) * f(x) < 0.0):
                a = x
            else:
                print('failure')
                break
            print(iteration, x)
    return x

def f(x):
    y = np.log(x) + x
    return y

a = 0.1
b = 1

x = bisection(f, a, b)
print('The approximate solution is: ', x)
print('And the error is: ', f(x))
```

```
1 0.55
2 0.775
3 0.6625000000000001
4 0.6062500000000001
5 0.578125
6 0.5640625
7 0.57109375
8 0.567578125
9 0.5658203125000001
10 0.5666992187500001
11 0.567138671875
12 0.5673583984375
13 0.56724853515625
14 0.567193603515625
15 0.5671661376953125
16 0.5671524047851563
17 0.5671455383300781
18 0.567142105102539
19 0.5671438217163085
20 0.5671429634094238
The approximate solution is:  0.5671429634094238
And the error is:  -9.035750279107191e-07
```

Observations

I noticed that the bisection method takes alot of iterations to find a approximate solution

Discussion

This may be because it takes an interval that's between the root to find the mid point which will eventually lead to an estimate of the root. The number of iteration this method will take depends on how precise the 2 point guess is.

Conclusions

In conclusion, bisection method isn't the best way to find the root of a equation as it can take a number of iteration depending on how precise the interval of the user's guess is.

Question 2

Procedure

In [33]:
```
import numpy as np
from numpy import *

def newton(f, df, x0, tol, nmax = 50):
    # f = the function f(x)
    # df = the derivative of f(x)
    # x0 = the initial guess of the solution
    # tol = tolerance for the absolute error of two subsequent approximations
    xk = x0
    for i in range(0, nmax):

        if(abs(f(xk)) <= tol):
            print("solution found after ", i, " iterations")
            return xk

        if(df(xk) == 0): #if df reaches 0
            print("no solutions avaliable")
            return None

        originalX = xk #before creating xn+1
        xk = originalX - f(originalX)/df(originalX) #xn+1 = xn - f(xn)/df(xn)
        print(i, "xn:", originalX, "\n  f(xn):", f(originalX), " \n  |xn+1 - xn|:",  abs(x
        if(i == 0 and (abs(xk  - originalX) <= tol)):#sotpping at first citerion
            return xk;
    print("Failure: algorithm fail to converge using only NMAX iteration")
    return None



def f(x):
    y = np.log(x) + x
    return y

def df(x):
    y = 1.0 / x + 1.0
    return y


tol = 1.e-4
x0 = 1.0 #inital guess
x = newton(f, df, x0, tol)
print('The approximate solution is: ', x)
print('And the error is: ', f(x))
```

```
0 xn: 1.0 
  f(xn): 1.0  
  |xn+1 - xn|: 0.5
1 xn: 0.5 
  f(xn): -0.1931471805599453  
  |xn+1 - xn|: 0.06438239351998176
2 xn: 0.5643823935199818 
  f(xn): -0.007640861009083455  
  |xn+1 - xn|: 0.0027565941950783435
solution found after  3  iterations
The approximate solution is:  0.5671389877150601
And the error is:  -1.1889333088377363e-05
```

Observations

I noticed that the newton method takes a lot less time than the bisection method, in terms of total iteration.

Discussion

I believe the newton's method is faster than bisection method as it uses calculus in order to find the root of the equation. This happens by comparing the gradient of a point to how high or low it is on the output axis. Hence after a couple of iteration gives us an estimate of where the root is.

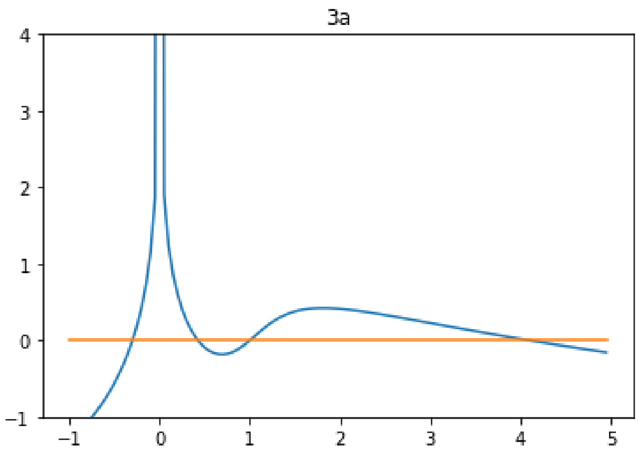
Conclusion

In conclusion, newton's method is faster than bisection method as it doesn't take a step by step method to calculate the roots. But, instead compares the gradient of the estimated point to how high it is on the graph in order to provide a solution.

Question 3

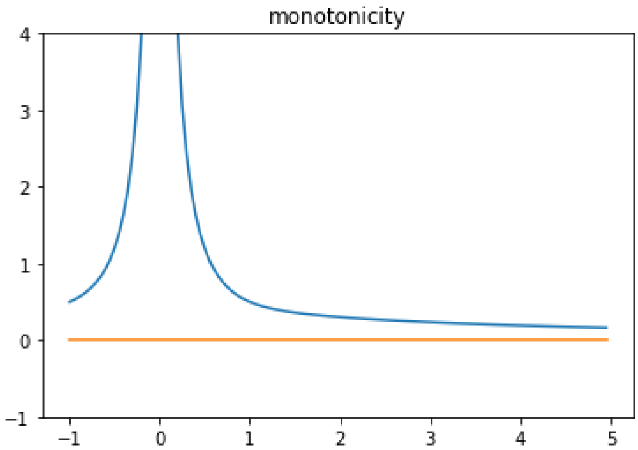
Procedure

In [3]:
```
from matplotlib import pyplot as plt
import scipy.optimize as opt

def f(x):
    y = np.arctan(2.0 * (x-1.0)) - np.log(np.abs(x))
    return y

def df(x):
    y = (-1.0/(x**2.0 + 1.0)) + 1.0/abs(x)
    return y


x0 = -0.5
x = np.arange(-1.0, 5.0, 0.05)

plt.ylim(-1,4)
plt.title("3a")
fig1 = plt.plot(x, f(x), x, np.zeros(x.shape))
```



In [4]:
```
x0 = -0.5
x = np.arange(-1.0, 5.0, 0.05)
fig2 = plt.plot(x, df(x), x, np.zeros(x.shape))
plt.title("monotonicity")
plt.ylim(-1,4)
```

Out[4]: (-1, 4)



the monotonicity of the function can be describe in this graph, as we see before 0, the rate of increase goes up exponentially to infinity. However after 0, the rate of decrease goes down inverse exponentially until it reaches an asymptote to the x axis.

In [38]:
```
#question 3c

import numpy as np
from numpy import *

def newton(f, df, x0, tol, nmax = 50):
    # f = the function f(x)
    # df = the derivative of f(x)
    # x0 = the initial guess of the solution
    # tol = tolerance for the absolute error of two subsequent approximations
    xk = x0
    for i in range(0, nmax):

        if(abs(f(xk)) <= tol):
            print("solution found after ", i, " iterations")
            return xk

        if(df(xk) == 0): #if df reaches 0
            print("no solutions avaliable")
            return None

        originalX = xk #before creating xn+1
        xk = originalX - f(originalX)/df(originalX) #xn+1 = xn - f(xn)/df(xn)
       # print(i, "xn:", originalX, "\n  f(xn):", f(originalX), " \n  |xn+1 - xn|:",  abs(
        if(i == 0 and (abs(xk  - originalX) <= tol)):#sotpping at first citerion
            return xk;
    print("Failure: algorithm fail to converge using only NMAX iteration")
    return None

def bisection(f, a, b, nmax = 50, tol = 1.e-6):
    iteration = 0
    #a = xn
    #b = xn+1
    if (f(a) * f(b) < 0.0):
        while ((b-a) > tol and iteration < nmax):
            iteration += 1;
            if(iteration == 1 and abs(b-a) <= tol):
                return b;
            x = (a + b)/2
            if (f(a) * f(x) < 0.0):
                b = x
            elif (f(b) * f(x) < 0.0):
                a = x
            else:
                print('failure')
                break
            #print(iteration, x)
    return x

def f(x):
    y = np.arctan(2.0 * (x-1.0)) - np.log(np.abs(x))
    return y

def df(x):
    y = (-1.0/(x**2.0 + 1.0)) + 1.0/abs(x)
    return y

x0 = 0.1
x1 = 0.5
x2 = 1.0
x3 = 4.0

print("roots via newton's method: ", newton(f,df,x0, 1e-6), "at x0 = 0.1")
print("roots via newton's method: ", newton(f,df,x1, 1e-6), "at x0 = 0.5")
print("roots via newton's method: ", newton(f,df,x2, 1e-6), "at x0 = 1.0")
print("roots via newton's method: ", newton(f,df,x3, 1e-6), "at x0 = 4.0")

print("roots via bisection method: ", bisection(f,-1.0, 0.0), "at interval between -1.0 an
print("roots via bisection method: ", bisection(f, 0.0, 0.5), "at interval between 0.0 and
```

```
print("roots via bisection method: ", bisection(f, 0.5, 2.0), "at interval between 0.5 and
print("roots via bisection method: ", bisection(f, 3.0, 5.0), "at interval between 3.0 and
```

```
solution found after  20  iterations
roots via newton's method:  -0.3000977097378081 at x0 = 0.1
Failure: algorithm fail to converge using only NMAX iteration
roots via newton's method:  None at x0 = 0.5
solution found after  0  iterations
roots via newton's method:  1.0 at x0 = 1.0
Failure: algorithm fail to converge using only NMAX iteration
roots via newton's method:  None at x0 = 4.0
roots via bisection method:  -0.3000974655151367 at interval between -1.0 and 0.0
roots via bisection method:  0.4254121780395508 at interval between 0.0 and 0.5
roots via bisection method:  1.000000238418579 at interval between 0.5 and 2.0
roots via bisection method:  4.09946346282959 at interval between 3.0 and 5.0
```

```
C:\Users\ian\anaconda3\lib\site-packages\ipykernel_launcher.py:51: RuntimeWarning: divide
by zero encountered in log
```

in this code, i have removed both printing of the iteration. I noticed both newton's and bisection method gave the same root, which makes sense as they're both very close to each other.

In [29]:
```
#question 3d
import time
import scipy.optimize as opt

x0 = 0.4
print("my version")
start = time.time()
print("roots via newton's method: ", newton(f,df,x0, 1e-6))
end = time.time()
print("roots via newton's method elapsed time: ", abs(end-start))
start = time.time()
print("first visible root estimation: ", bisection(f,-0.5, -0.2))
end = time.time()
print("roots via bisection method elapsed time: ", abs(end-start))
print()
print("optimize version")
print("-------\n")
start = time.time()
print("bisection method: ", opt.bisect(f, 1, 4))
end = time.time()
print("optimize CPU bisection elapsed time: ", abs(end-start), "\n-------")

start = time.time()
print("newton_method: ", opt.newton(f,x0, df, tol=1e-9))
end = time.time()
print(" optimize CPU newton method elapsed time: ", abs(end-start), "\n-------")
#newtons method doesn't show all the roots between -0.5 and 4

start = time.time()
print("fsolve: ", opt.fsolve(f, x0))
end = time.time()
print("optimize CPU fsolve elapsed time", abs(end-start), "\n-----")
```

```
my version
solution found after  24  iterations
roots via newton's method:  -0.30009771682362235
roots via newton's method elapsed time:  0.0
first visible root estimation:  -0.3000974655151367
roots via bisection method elapsed time:  0.0

optimize version
-------

bisection method:  1.0
optimize CPU bisection elapsed time:  0.0
-------
newton_method:  -0.3000975657663618
 optimize CPU newton method elapsed time:  0.0009999275207519531
-------
fsolve:  [0.42541157]
optimize CPU fsolve elapsed time 0.00099945068359375
-----
```

In this code the main reason why i remove the formatting is to reduce the load hence how much it can affect the elapsed time of my algorithms. This is because it wouldn't be fair to take into account the clock speed if i was to tell the algorthim to print out each and every iteration to finding the root.

In [32]:
```
#question 3e
def newton(f, df, x0, tol, nmax = 50):
    # f = the function f(x)
    # df = the derivative of f(x)
    # x0 = the initial guess of the solution
    # tol = tolerance for the absolute error of two subsequent approximations
    xk = x0
    for i in range(0, nmax):

        if(abs(f(xk)) <= tol):
            print("solution found after ", i, " iterations")
            return xk

        if(df(xk) == 0): #if df reaches 0
            print("no solutions avaliable")
            return None

        originalX = xk #before creating xn+1
        xk = originalX - f(originalX)/df(originalX) #xn+1 = xn - f(xn)/df(xn)
       # print(i, "xn:", originalX, "\n  f(xn):", f(originalX), " \n  |xn+1 - xn|:",  abs(
        if(i == 0 and (abs(xk  - originalX) <= tol)):#sotpping at first citerion
            return xk;
    print("Failure: algorithm fail to converge using only NMAX iteration")
    return None


def different_guesses(guesses):

    for i in range(len(guesses)):
        print("at x0 = ", guesses[i], "\n\tnewtons method gives us: ", newton(f,df, guesse
        print("-------------------")

def f(x):
    y = np.arctan(2.0 * (x-1.0)) - np.log(np.abs(x))
    return y

def df(x):
    y = (-1.0/(x**2.0 + 1.0)) + 1.0/abs(x)
    return y


list_of_nmax = [-1.0, 0.65, 0.7, 1.7, 1.8, 1.9, 5.0, 10.0]

#print("at x0 = -1.0 \n\tnewtons method gives us: ", newton(f,df, -1.0, 1e-6, 50))
#print("at x0 = 0.65 \n\tnewtons method gives us: ", newton(f,df, 0.65, 1e-6, 50))
#print("at x0 = 0.7 \n\tnewtons method gives us: ", newton(f,df, 0.7, 1e-6, 50))
different_guesses(list_of_nmax)
```

```
solution found after  28  iterations
at x0 =  -1.0
	newtons method gives us:  -0.3000972954197656
-------------------
Failure: algorithm fail to converge using only NMAX iteration
at x0 =  0.65
	newtons method gives us:  -1
-------------------
Failure: algorithm fail to converge using only NMAX iteration
at x0 =  0.7
	newtons method gives us:  -1
-------------------
Failure: algorithm fail to converge using only NMAX iteration
at x0 =  1.7
	newtons method gives us:  -1
-------------------
Failure: algorithm fail to converge using only NMAX iteration
at x0 =  1.8
	newtons method gives us:  -1
-------------------
```

```
Failure: algorithm fail to converge using only NMAX iteration
at x0 =  1.9
        newtons method gives us:  -1
-------------------
Failure: algorithm fail to converge using only NMAX iteration
at x0 =  5.0
        newtons method gives us:  -1
-------------------
Failure: algorithm fail to converge using only NMAX iteration
at x0 =  10.0
        newtons method gives us:  -1
-------------------
```

it seems like all but x0 = -1 would fail to converge using only nmax iterations.

Observations

I have noticed that my algorthim goes just as fast as the optimize's algorthim. I also plot out the direved graph of $\arctan(2(x-1)) - \ln|x|$. which helps me describe the monotonicity of the function.

Discussion

I believe the reason why my alogrthim works just as fast as optimize's algorthim is because it's based off the same idea. For example, newton's method can only be done in a certain way to find it's output, thus it's almost certain that optimize and I would be using the same structure for the algorthim. Using the direvative of the function you can easily visualise the monotonicity of the fucntion.

Conclusion

In conclusions, optimize algorthims are ideal but you can always write your own rendition which can at times work just as well as they follow the same structure.

Question 4

Procedure

In [43]:
```
#question 4a

def newton(f, df, x0, tol, nmax = 50):
    # f = the function f(x)
    # df = the derivative of f(x)
    # x0 = the initial guess of the solution
    # tol = tolerance for the absolute error of two subsequent approximations
    err = 1.0
    iteration = 0

    xk = x0
    while (err > tol):
        if(iteration > nmax):
            print("Failure: algorithm fail to converge using only NMAX iteration")
        iteration = iteration + 1
        err = xk
        originalX = xk #xn
        xk = originalX - f(originalX)/df(originalX)#xn+1 = xn - f(xn)/df(xn)
        if(iteration == 1 and (abs(xk  - originalX) <= tol)):
            return xk;
        if((abs(xk  - originalX) >= 1e10)):
            break
        err = abs(err - xk)
        #print(iteration, "xn", originalX, "\n f(xn)", f(originalX), " \n |xn+1 - xn|",  a
    return xk

def bisection(f, a, b, nmax = 50, tol = 1.e-6):
    iteration = 0
    olda = a
    oldb = b
    #a = xn
    #b = xn+1
    x = 0
    if (f(a) * f(b) < 0.0):
        while ((b-a) > tol):
            if(iteration > nmax):
                print("Failure: algorithm fail to converge using only NMAX iteration")
            iteration += 1;
            if(iteration == 1 and abs(b-a) <= tol):
                return b;
            x = (a + b)/2
            if (f(a) * f(x) < 0.0):
                b = x
            elif (f(b) * f(x) < 0.0):
                a = x
            else:
                print('failure')
                break
#            print(iteration, x)
        print("total of ", iteration, " number of iteration for interval between ", olda,
    return x

def find_zero(a,b,tol, maxit, f, df):

    ierr = 0
    niter = 0
    xstar = 0

    if(f(a)*f(b) <= 0.0):

        xi = (a + b)/2.0
        for niter in range(maxit):
            oldx = xi
            xi = oldx - f(oldx)/df(oldx)

            if(xi <= a or xi >= b): #not inside a or b and including a and b
                xi = (a+b)/2.0
            if(f(a)*f(xi) <= 0.0): #check for convergence
                b=xi
```

```
            else:
                a=xi
            ierr = abs(oldx - xi)
            #print(niter,xi,ierr)
            if(ierr <= tol):
                break

        if(niter >= maxit):
            ierr = 2 #maximum number of iteration has been reached, did not converge
        else:
            ierr = 0 #made convergence

        xstar = xi
        return xstar ,niter ,ierr

    ierr = 1 # df singular
    return xstar, niter, ierr
```

In [44]:
```
#question 4b

def f(x):
    y = np.arctan(2.0 * (x-1.0)) - np.log(np.abs(x))
    return y

def df(x):
    y = (-1.0/(x**2.0 + 1.0)) + np.divide(1.0 , abs(x))
    return y

a = -1.0
b = 0.0
tol = 1.0e-6
nmax = 50
root, niter, ierr = find_zero(a,b,tol,nmax,f,df)

print("root approx: ", root)
print("number of iteration", niter)
if(ierr == 0):
    print("method has converged")
elif(ierr == 1):
    print("df is singular")
elif(ierr == 2):
    print("maximum number of iteration has been reached")

print("------------------------")

a = 0.0
b = 0.5
tol = 1.0e-6
nmax = 50
root, niter, ierr = find_zero(a,b,tol,nmax,f,df)

print("root approx: ", root)
print("number of iteration", niter)
if(ierr == 0):
    print("method has converged")
elif(ierr == 1):
    print("df is singular")
elif(ierr == 2):
    print("maximum number of iteration has been reached")

print("------------------------")
a = 0.5
b = 2.0
tol = 1.0e-6
nmax = 50
root, niter, ierr = find_zero(a,b,tol,nmax,f,df)

print("root approx: ", root)
print("number of iteration", niter)
if(ierr == 0):
    print("method has converged")
elif(ierr == 1):
    print("df is singular")
elif(ierr == 2):
    print("maximum number of iteration has been reached")

print("------------------------")
a = 3.0
b = 5.0
tol = 1.0e-6
nmax = 50
root, niter, ierr = find_zero(a,b,tol,nmax,f,df)

print("root approx: ", root)
print("number of iteration", niter)
if(ierr == 0):
    print("method has converged")
```

```
elif(ierr == 1):
    print("df is singular")
elif(ierr == 2):
    print("maximum number of iteration has been reached")

print("------------------------")
```

```
root approx:  -0.30009726667125747
number of iteration 18
method has converged
------------------------
root approx:  0.4254111285263085
number of iteration 19
method has converged
------------------------
root approx:  0.9745687903631043
number of iteration 49
method has converged
------------------------
root approx:  4.099463403268368
number of iteration 21
method has converged
------------------------

C:\Users\ian\anaconda3\lib\site-packages\ipykernel_launcher.py:4: RuntimeWarning: divide
by zero encountered in log
  after removing the cwd from sys.path.
```

Observation

I have noticed my guesses could not find the root. Hence why it returned 0.

Discussion

I believe the reason why it could not find a root at this point is because a root might not have existed between the points of 0.1 and 4.0 which i have PROVED by just using the bisection method.

Conclusion

In conclusion, I believe that this function can be extremely useful when used correctly. However, it wasn't able to work with my estimation as there wasn't a root between the 2 values.

Question 5

Procedure

In [29]:
```
#question 5a
def secant(f, x1, x2, tol = 1.e-6):
    # f = the function f(x)
    # x1 = first guess of the root
    # x2 = second guess of the root
    # tol = tolerance for the absolute error of two subsequent approximations
    err = 1.0
    iteration = 0
    e1 = abs(x1 - x2)
    e2 = e1
    e3 = e1

    while (err > tol):
        e1 = e2
        e2 = e3

        xk = x1
        xk1 = x2
        iteration = iteration + 1
        err = xk1
        xk1 = xk - f(xk)*(xk-xk1)/(f(xk)-f(xk1))
        err = abs(err - xk1)
        x1 = x2
        x2 = xk1
        print(iteration, xk1)
        e3 = np.abs(xk1-1.0)

        rate = np.log(e2/e3)/np.log(e1/e2)
        #i noticed the final rate of convergence is 1.618 which is phi
        print("rate: ",rate)
    return xk

def f(x):
    y = x**20.0 - 1.0 #x^20=1
    return y

tol = 1.e-10
x1 = 2.0
x2 = 3.0
x = secant(f, x1, x2, tol)
print('The aproximate solution is: ', x)
print('And the error is: ', f(x))
```

```
1 1.9996991811621294
rate:  inf
2 1.9993991760039374
rate:  0.9975948517825786
3 1.8995717405987151
rate:  350.6207212065357
4 1.8436563962557535
rate:  0.6098087382335381
5 1.7752727198108538
rate:  1.3172169773669937
6 1.7147354051866672
rate:  0.9618127855092251
7 1.6542910390366605
rate:  1.0868078707951423
8 1.596714502668227
rate:  1.0424777710574755
9 1.5408696846047667
rate:  1.0667320372555233
10 1.4870843258447324
rate:  1.0659543505678852
11 1.435146222630197
rate:  1.076512784049195
12 1.385051927825446
rate:  1.0846859172392922
13 1.33673155019979
```

```
rate:  1.0963849823393614
14 1.290159069541252
rate:  1.1101075496747683
15 1.2453264532911057
rate:  1.1275237173142396
16 1.2022828842682187
rate:  1.1494472310617383
17 1.1611766610067804
rate:  1.1774983229424387
18 1.122343957689072
rate:  1.2134937202023979
19 1.0864551632045294
rate:  1.2595394035552345
20 1.0547196260583802
rate:  1.3173666624015208
21 1.0290153392485129
rate:  1.3869544417744664
22 1.0114631890742571
rate:  1.4638822267087968
23 1.0027515374756275
rate:  1.5365631451567356
24 1.0002851344488886
rate:  1.588636510341001
25 1.0000073743917859
rate:  1.6122721994425884
26 1.0000000199551384
rate:  1.6176107070682562
27 1.000000000001398
rate:  1.6180223448623643
28 1.0
rate:  inf
The aproximate solution is:  1.0000000199551384
And the error is:  3.99102844328425e-07
```

```
C:\Users\ian\anaconda3\lib\site-packages\ipykernel_launcher.py:28: RuntimeWarning: divide
by zero encountered in double_scalars
```

as you can see on the 26th iteration we get a rate of 1.618 the golden ratio!

Observation

I have noticed that using the rate of convergence of the secant method for the function x^20 = 1 gives us a rate which == phi. not only that it also gives us a root == 1 which is what we want.

Discussion

Although secant method uses 2 points to approximate the root, it is often thought of as the "finite approximation of Newton's method". I also notice that the rate of convergence for x^20 gives us phi, 1.618... which is often known as the golden ratio. This number is known as the most beautiful number.

Conclusion

In conclusion, I believe the secant method is modification of newton's method. However, the drawback to the secant method is that it uses 2 approximation first where as newton's method only requires a guess of the inital root.

In [ ]: