COSC 343: homework 2

Micah Sherry

March 20, 2024

1 accelerated Newton's Method

If the root $f(x) = 0$ is a double root (has multiplicity 2), then newtons method can be accelerated by using:

$$x_{n+1} = x_n - 2\frac{f(x)}{f'(x_n)}$$

Numerically compare the convergence of this scheme with newtons method on a function with a known double root.

For this question I will use the function: $f(x) = (x-3)^2(1+e^x)$ to compare the rate of convergence of Newton's Method and the Accelerated Newton's method.

code:

```
import numpy as np
import matplotlib.pyplot as plt


def f(x):
    return (((x - 3) ** 2) * (1 + np.exp(x)))

def fp(x):
    return (x-3) * ((x - 1) * np.exp(x) + 2)


def multiplicy2_NewtonsMethod(x0,trueroot=None, f=f,fp=fp, tol=1e-7,N=100):
    size = 1.0
    errorVec = []
    i = 0
    while (size > tol and i < N) and f(x0) != 0:

        x1   = x0 - 2 * (f(x0) / fp(x0))

        if trueroot is not None:
            errorVec.append(np.abs(x1 - trueroot))

        size = np.abs(x1 - x0)
        x0   = x1
        i   += 1

    if trueroot is not None:
        return x1, errorVec
    else:
        return x1
```

```python
def NewtonsMethod(x0,trueroot=None, f=f,fp=fp, tol=1e-7,N=100):
    size = 1.0
    errorVec = []
    i = 0
    while (size > tol and i < N) and f(x0) != 0:

        x1   = x0 - (f(x0) / fp(x0))

        if trueroot is not None:
            errorVec.append(np.abs(x1 - trueroot))

        size = np.abs(x1 - x0)
        x0   = x1
        i   += 1

    if trueroot is not None:
        return x1, errorVec
    else:
        return x1

def findAlpha(vec):
    alphaVec = []
    for i in range(len(vec) - 3):
        a = np.log(vec[i+2] / vec[i+1]) / np.log(vec[i+1] / vec[i])
        alphaVec.append(a)
    return alphaVec


if __name__=="__main__":
    x1, errorVec1 = NewtonsMethod(4, 3 ,f=f ,fp=fp)
    x2, errorVec2 = multiplicy2_NewtonsMethod(4, 3 ,f=f ,fp=fp)

    alphaVec1 = findAlpha(errorVec1)
    alphaVec2 = findAlpha(errorVec2)
    print("newtons method approximation:",x1)
    print("accelerated newtons method approximation:",x2)

    plt.plot(alphaVec1)
    plt.show()
    plt.plot(alphaVec2)
    plt.show()
```

output and plots:

Newton's method approximated the root of f(x) to be 3.000000071980329
Accelerated Newton's method approximated the root of f(x) to be 3.0 (not exactly 3.0)
From visual inspection of the graphs Newton's method converges at a rate of $O(h^1)$ and the accelerated Newton's method converges at a rate of $O(h^2)$. My hypothesis for why the accelerated Newton's method performs better than the standard Newton's method is that the two cancels out the two in the denominator that comes from the differentiating a squared term in the function.

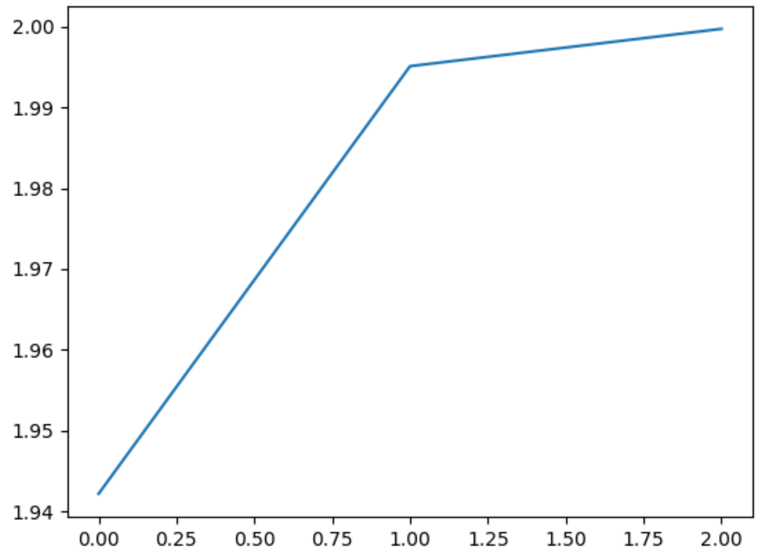

Figure 1: accelerated newtons method

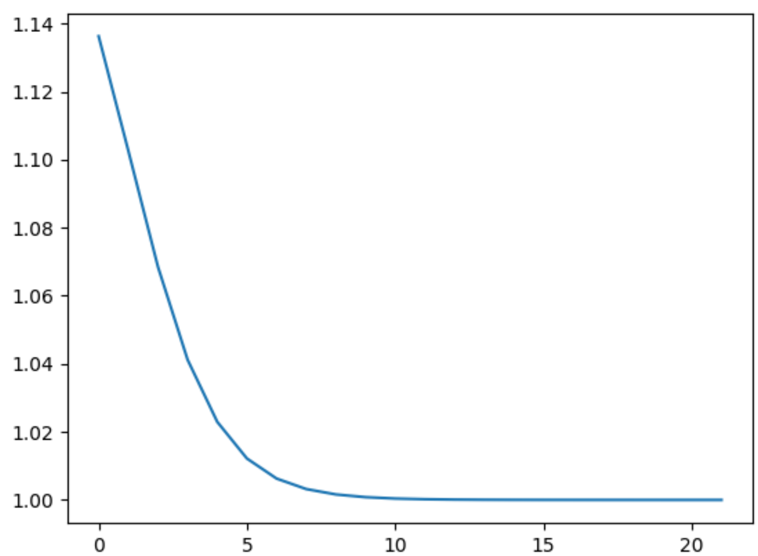

Figure 2: newtons method convergence

2 Newton's method with secant

Write and test a recursive procedure for the secant method, Numerically estimate the methods order of convergence. Note in the secant method the derivative in Newton's method is replaced with an approximation

$$f'(x_i) \approx \frac{f(x_i) - f(x_{i-1})}{x_i - x_{i-1}}$$

for this question I will use the function $f(x) = (x-4)(e^x+1)$ to verify it works.

code:

```
import numpy as np
import matplotlib.pyplot as plt

def f(x):
    return (x-4)* (np.exp(x)+1)

def secant(x0, x1, f=f):
    return (f(x1) - f(x0)) / (x1-x0)


def secantMethod(x0, x1, f=f,trueroot=None, errorVec = [], tol=1e-7, N=100):
    size = np.abs(x1 - x0)

    if size <= tol:
        return x1 if trueroot is None else (x1, errorVec)
    if N <= 0:
        print(trueroot)
        return x1 if trueroot is None else (x1, errorVec)

    x2 = x0 - f(x0)/secant(x0, x1, f=f)

    if trueroot is not None:
        errorVec.append(np.abs(trueroot-x0))
        return secantMethod(x1, x2, f,trueroot, errorVec, tol, N=N-1)
    else:
        return secantMethod(x1, x2, f, None, errorVec, tol, N=N-1)

def findAlpha(vec):
    alphaVec = []
    for i in range(len(vec) - 3):
        a = np.log(vec[i+2] / vec[i+1])/ np.log(vec[i+1] / vec[i])
        alphaVec.append(a)
    return alphaVec


if __name__=="__main__":
    x, errorVec = secantMethod(10,9,trueroot=4)
    alphaVec = findAlpha(errorVec)
    print(x)
    plt.plot(alphaVec)
    plt.show()
```

output and plots:

The secant method approximated the root of f(x) to be 4.000000000076314
From visual inspection of the graph (and discussion In class) we see that the secant method converges at a rate of approximately $O(h^{1.6})$ or more accurately $O(h^{\phi})$ (where ϕ is the golden ratio)

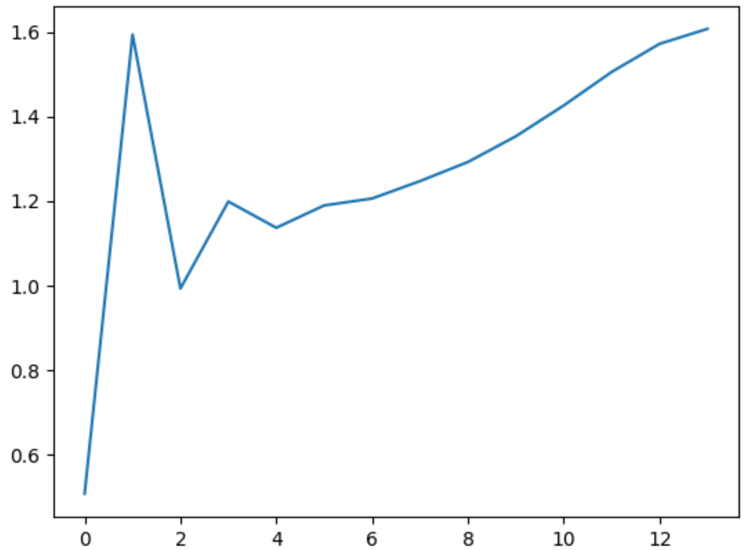

Figure 3: secant method convergence

3 Olver's Method

test numerically Olver's method given by

$$x_{n+1} = x_n - 2\frac{f(x_n)}{f'(x_n)} - \frac{1}{2}\frac{f''(x_n)}{f'(x_n)}\left[\frac{f(x_n)}{f'(x_n)}\right]^2$$

establish an estimate for its rate of convergence.

For Olver's I will use the function $f(x) = (x-3)*e^x$ to verify it works and find the rate of convergence

code:

```
import numpy as np
import matplotlib.pyplot as plt


def f(x):
    return (x-3) * np.exp(x)
def fp(x):
    return (x-2) * np.exp(x)
def fdp(x):
    return (x-1) * np.exp(x)



def olversMethod(x0,trueroot=None, f=f,fp=fp, fdp=fdp, tol=1e-7,N=100):
    size = 1.0
    errorVec = []
    i = 0
    while (size > tol and i < N):
        #print(size)
```

```
        x1    = x0 - (f(x0)/fp(x0)) - ((1/2)*(fdp(x0)/fp(x0)) * ((f(x0)/fp(x0))**2))
        print(i, x1)
        if trueroot is not None:
            errorVec.append(np.abs(x1 - trueroot))

        size = np.abs(x1 - x0)
        x0    = x1
        i    += 1

    if trueroot is not None:
        return x1, errorVec
    else:
        return x1

def findAlpha(vec):
    alphaVec = []
    for i in range(len(vec) - 3):
        a = np.log(vec[i+2] / vec[i+1]) / np.log(vec[i+1] / vec[i])
        alphaVec.append(a)
    return alphaVec


if __name__=="__main__":
    x, errorVec = olversMethod(14.3232, 3 ,f=f ,fp=fp,fdp=fdp)
    alphaVec = findAlpha(errorVec)
    print(alphaVec)
    plt.plot(alphaVec)
    plt.show()
    print(x)
```

output and code

Olver's method approximated the root of f(x) to be 3.000000000173341.
From inspection of the graph the rate of convergence appears to be $O(h^3)$. this makes since because Olver's method is derived from the three term expansion of Taylor's series which itself converges at a rate of $O(h^3)$.

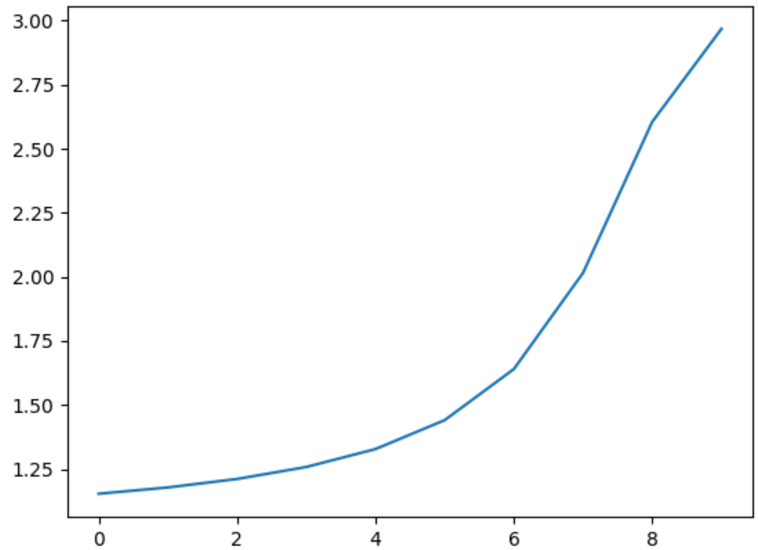

Figure 4: Olver's method convergence

4 solving a Nonlinear system of equations

write a program that can be used to solve the following non linear system

$$w_1 + w_2 = 2$$

$$w_1 x_1 + w_2 x_2 = 0$$

$$w_1 x_1^2 + w_2 x_2^2 = \frac{2}{3}$$

$$w_1 x_1^3 + w_2 x_2^3 = 0$$

how many different solutions can you find?
The first step I took to solve this system was to solve the system of equations was to solve each system for zero and assign each of them a function.

$$f_0(x) = w_1 + w_2 - 2$$

$$f_1(x) = w_1 x_1 + w_2 x_2$$

$$f_2(x) = w_1 x_1^2 + w_2 x_2^2 - \frac{2}{3}$$

$$f_3(x) = w_1 x_1^3 + w_2 x_2^3$$

The next step was to find the Jacobian system for that I got

$$\begin{pmatrix} 1 & 1 & 0 & 0 \\ x_1 & x_2 & w_1 & w_2 \\ x_1^2 & x_2^2 & 2w_1x_1 & 2w_2x_2 \\ x_1^3 & x_2^3 & 3w_1x_1^2 & 3w_2x_2^2 \end{pmatrix}$$

Then used Newton's method for vectors to find the root(s) of the system of equations

code:

```python
import matplotlib.pyplot as plt
import numpy as np
from numpy.random import rand



def f(X):
    w1 = X[0]
    w2 = X[1]
    x1 = X[2]
    x2 = X[3]

    F = np.zeros(4).reshape(4,1)

    F[0] = w1           + w2           - 2
    F[1] = w1*x1        + w2*x2        - 0
    F[2] = w1*x1 ** 2 + w2*x2 ** 2 - 2/3
    F[3] = w1*x1 ** 3 + w2*x2 ** 3 - 0

    return F

def Jacobian(X):

    w1 = X[0]
    w2 = X[1]
    x1 = X[2]
    x2 = X[3]

    J = np.zeros(4*4).reshape(4,4)

    #the compoenents come from the partial derivatives
    # (row col),(i,j) compenetnt is d f_i(x)/ d x_j

    #  partial derivitives of f_0
    J[0,0] = 1
    J[0,1] = 1
    J[0,2] = 0
    J[0,3] = 0

    #  partial derivitives of f_1
    J[1,0] = x1
    J[1,1] = x2
    J[1,2] = w1
    J[1,3] = w2

    #  partial derivitives of f_2
    J[2,0] = x1**2
    J[2,1] = x2**2
    J[2,2] = 2*w1*x1
    J[2,3] = 2*w2*x2

    #  partial derivitives of f_3
    J[3,0] = x1**3
    J[3,1] = x2**3
    J[3,2] = 3*w1*x1**2
```

```
    J[3,3] = 3*w2*x2**2

    return J


def vectorNewtonsMethod(X, Jacobian=Jacobian, f=f, tol = 1e-7, maxIter=1000):
    s = np.ones(4).reshape(4,1)
    i = 0
    while(np.linalg.norm(s)>tol) and  i < maxIter:
        js = Jacobian(X)
        b  = -f(X)
        #print(js)
        #print(b)
        s = np.linalg.solve(js,b)
        X += s
        i +=1
    return(X)


if __name__=="__main__":
    X = -10 + 5*rand(4,1)

    X = vectorNewtonsMethod(X)
    print(X)
    print(f(X))
```

solution

the solutions are of the form

$$\begin{bmatrix} w1 \\ w2 \\ x1 \\ x2 \end{bmatrix}$$

I was only able to find one solution to the system of equations.

$$\begin{bmatrix} 1.0000 \\ 1.0000 \\ 0.5774 \\ -0.5774 \end{bmatrix}$$