midterm Rahul Deshmukh

October 17, 2017

0.1 Midterm Exam ME581 by Rahul Deshmukh (PUID:0003004932)

```
In [1]: import numpy as np
        import matplotlib.pyplot as plt
```

Problem 1
given that the function $f(x) = 2x^4 - 12x^2 + 16x - 6$ has a root at $x = 1$
Function for Newtons method:

```
In [2]: def newton(ini,N,f,df):
            p = np.zeros(N+1)
            p[0]=ini
            for i in range(0,N):
                p[i+1]= p[i] - f(p[i])/df(p[i])
            print('i \t pi \n')
            print('0 \t '+str(ini)+'\n')
            for i in range(0,N):
                print(str(i+1)+'\t '+str(p[i+1])+'\n');
            return(p)
```

(a)

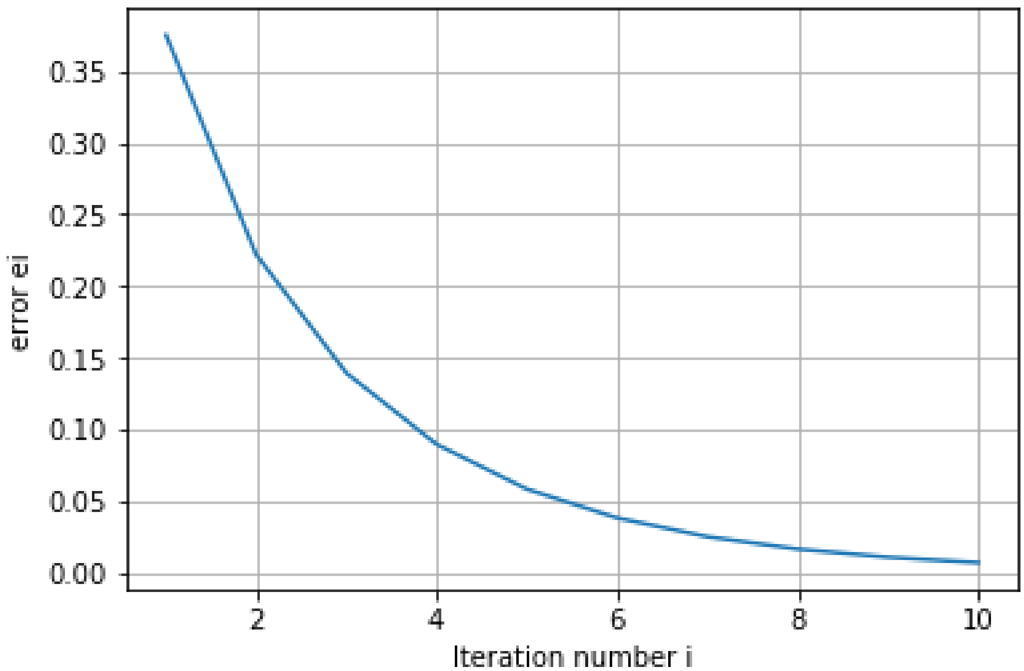
```
In [3]: def f(x):
            return(2*x**4-12*x**2+16*x-6)
        def df(x):
            return(8*x**3-24*x+16)
        ini = 0
        N=10
        r = newton(ini,N,f,df)
```

```
i          pi

0          0

1        0.375

2        0.597039473684

3        0.736569978917
```

```
4          0.826493196294

5          0.88521637001

6          0.923858119974

7          0.949403984543

8          0.966341652806

9          0.977592927951

10          0.985076003498
```

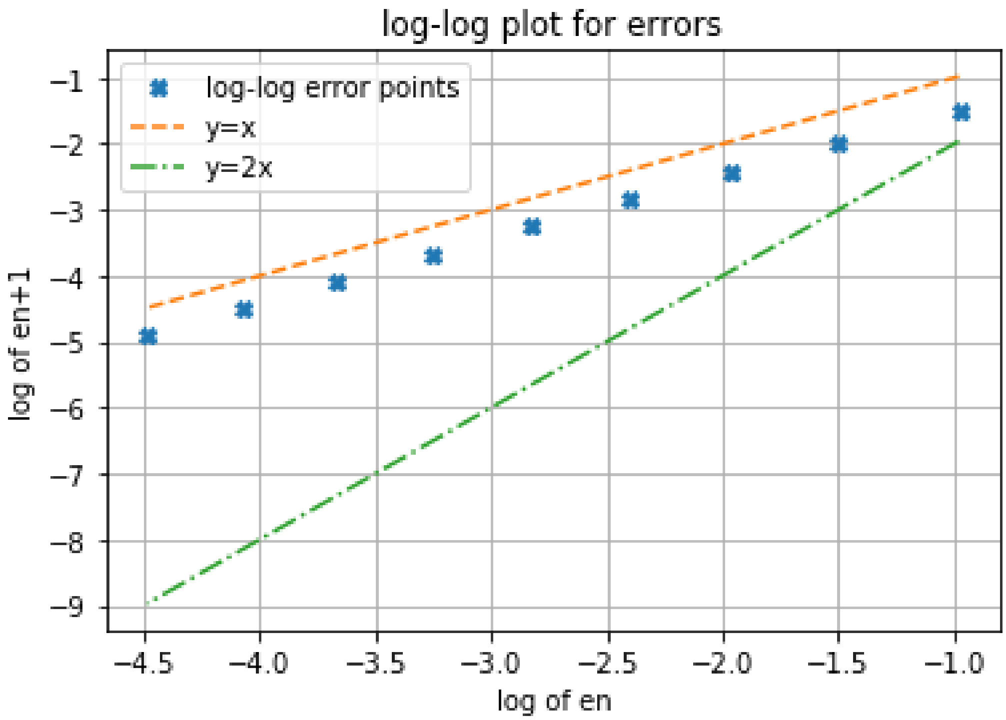
(b)

```
In [4]: er = np.zeros(N)
        for j in range(0,N):
            er[j]=np.abs(r[j+1]-r[j])
        #print(er)
        erp =np.zeros(N-1)
        for j in range(0,N-1):
            erp[j] = er[j+1]
        plt.xlabel('Iteration number i')
        plt.ylabel('error ei')
        plt.grid(1)
        plt.plot(np.ones(N)+np.arange(N),er)
        plt.show()
```



(c)

```
In [5]: plt.xlabel('log of en')
        plt.ylabel('log of en+1')
        plt.title('log-log plot for errors')
        plt.grid(1)
        a,=plt.plot(np.log(er[:-1]),np.log(erp),'X',label='log-log error points')
        b,=plt.plot(np.log(er[:-1]),np.log(er[:-1]),'--',label='y=x')
        c,=plt.plot(np.log(er[:-1]),2*np.log(er[:-1]),'-.',label='y=2x')
        plt.legend(handles=[a,b,c])
        plt.show()
```



We can observe from the above plot that the log-log plot of errors is along the line y=x therefore, the order of convergence for $x_0 = 0$ is **linear**

This is expected as when we factorise the function $f(x) = 2x^4 - 12x^2 + 16x - 6$ we get $f(x) = 2(x-1)^3 * (x+3)$ therefore the multiplicity of root at $x = 1$ is $m = 3$

```
In [6]: root =1
        p=r[1:]
        ratio = np.zeros(N-1)
        for i in range(0,N-1):
            ratio[i]=np.abs(p[i+1]-root)/np.abs(p[i]-root)
        print('|en+1|/|en| is')
        print(ratio)#asymptotic error rate
```

```
|en+1|/|en| is
[ 0.64473684  0.65373654  0.65864476  0.66155118  0.66335139  0.66449654
  0.66523711  0.6657211   0.66603956]
```

Also as can be seen above the ratio of $\frac{|e_{n+1}|}{|e_n|^1}$ converges to the value 0.666 which is same as $\lambda = 1 - \frac{1}{m} = 1 - \frac{1}{3} = \frac{2}{3}$

(d)

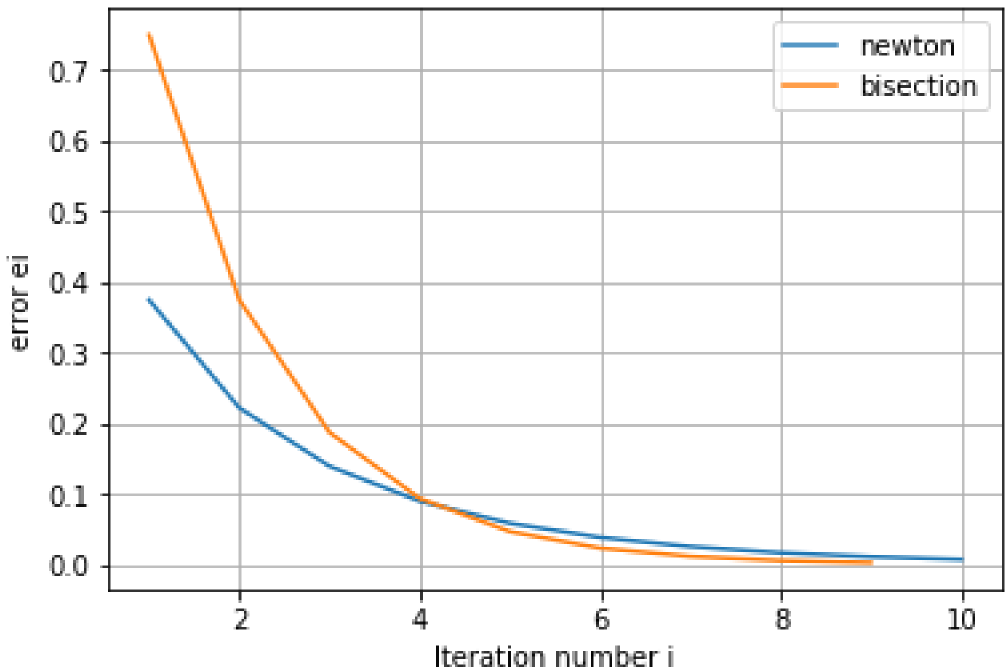
Function for bisection method

```
In [7]: def bisection(f,N,a0,b0):
            p=np.zeros(N)
            a=np.copy(a0)
            b=np.copy(b0)
            for i in range (0,N):
                p[i] = (a + b)/2
                if (f(p[i])*f(b))<0:
                    a = p[i]
                else:
                    b = p[i]
            print('the approximate root after '+str(N)+' iterations is '+str(p[-1]))
            return(p)
```

```
In [8]: a0=0;b0=3;N=10
        bis=bisection(f,N,a0,b0)
        #print(bis)
```

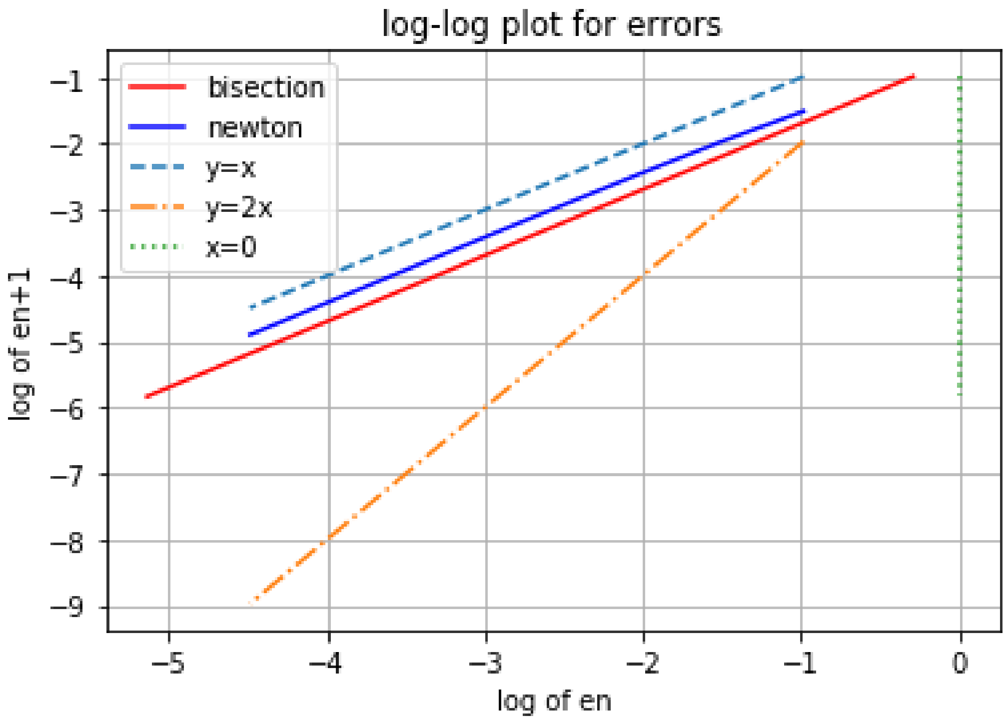
```
the approximate root after 10 iterations is 0.9990234375
```

```
In [9]: erb = np.zeros(N-1)
        for j in range(0,N-1):
            erb[j]=np.abs(bis[j+1]-bis[j])
        #print(erb)
        erbp =np.zeros(N-2)
        for j in range(0,N-2):
            erbp[j] = erb[j+1]
        plt.grid(1)
        plt.xlabel('Iteration number i')
        plt.ylabel('error ei')
        a,=plt.plot(np.ones(N)+np.arange(N),er,label='newton')
        b,=plt.plot(np.ones(N-1)+np.arange(N-1),erb,label='bisection')
        plt.legend(handles=[a,b])
        plt.show()
```



from the above plot we can see that the error converges faster for bisection method compared to newtons method

```
In [10]: plt.xlabel('log of en')
         plt.ylabel('log of en+1')
         plt.title('log-log plot for errors')
         plt.grid(1)
         a,=plt.plot(np.log(erb[:-1]),np.log(erbp),'red',label='bisection')
         b,=plt.plot(np.log(er[:-1]),np.log(erp),'blue',label='newton')
         c,=plt.plot(np.log(er[:-1]),np.log(er[:-1]),'--',label='y=x')
         d,=plt.plot(np.log(er[:-1]),2*np.log(er[:-1]),'-.',label='y=2x')
         e,=plt.plot(0*np.log(erbp),np.log(erbp),':',label='x=0')#line x=0
         plt.legend(handles=[a,b,c,d,e])
         plt.show()
```



from log log plot of error it is evident that the bisection method has a lower value of y-intercept when compared to newtons method. Therefore, the asymptotic error rate for bisection method ($\lambda = 1/2$) is less than newtons method($\lambda = 2/3$) and thus bisection method converges faster.Also as both the plots are along y=x the order of convergence for bisectioon and newton method is **linear**

(e)

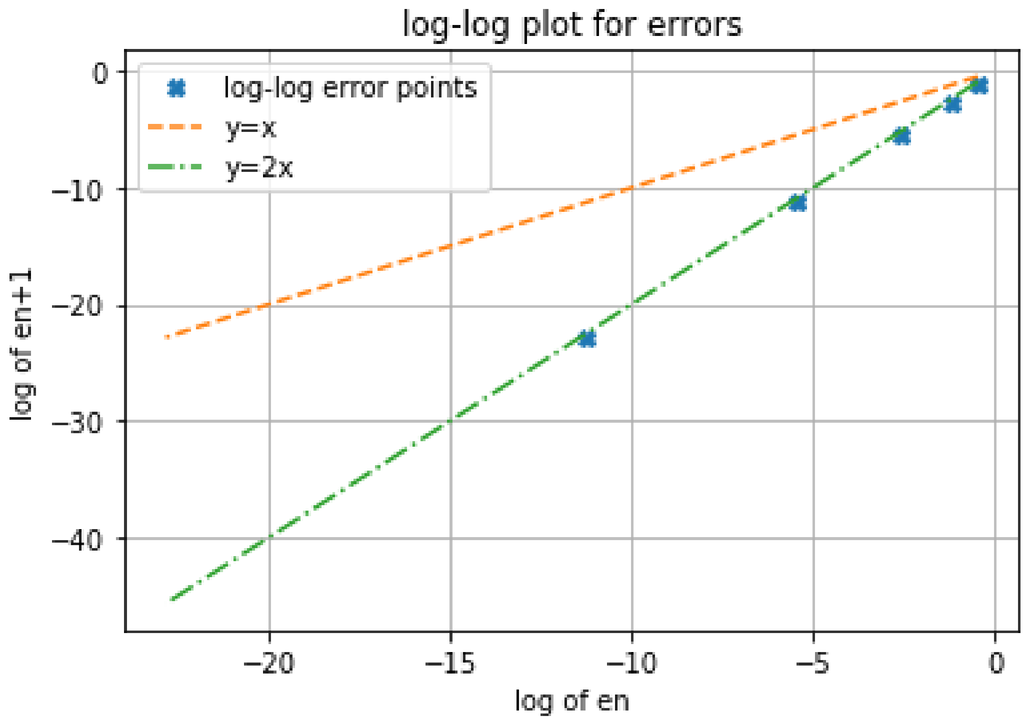
```
In [11]: ini = -4
         N=10
         r2 = newton(ini,N,f,df)
```

```
i           pi

0           -4

1          -3.375

2          -3.07670454545

3          -3.00409832994

4          -3.00001254581

5          -3.00000000012
```

```
6         -3.0

7         -3.0

8         -3.0

9         -3.0

10         -3.0
```

```
In [12]: er2 = np.zeros(N)
         for j in range(0,N):
             er2[j]=np.abs(r2[j+1]-r2[j])
         erp2 =np.zeros(N-1)
         for j in range(0,N-1):
             erp2[j] = er2[j+1]
         plt.xlabel('log of en')
         plt.ylabel('log of en+1')
         plt.title('log-log plot for errors')
         plt.grid(1)
         a,=plt.plot(np.log(er2[:-1]),np.log(erp2),'X',label='log-log error points')
         b,=plt.plot(np.log(er2[:-1]),np.log(er2[:-1]),'--',label='y=x')
         c,=plt.plot(np.log(er2[:-1]),2*np.log(er2[:-1]),'-.',label='y=2x')
         plt.legend(handles=[a,b,c])
         plt.show()
```

```
/apps/share64/debian7/anaconda/anaconda3-4.4/lib/python3.6/site-packages/ipykernel_launcher.py:1
  # This is added back by InteractiveShellApp.init_path()
/apps/share64/debian7/anaconda/anaconda3-4.4/lib/python3.6/site-packages/ipykernel_launcher.py:1
  if sys.path[0] == '':
/apps/share64/debian7/anaconda/anaconda3-4.4/lib/python3.6/site-packages/ipykernel_launcher.py:1
  del sys.path[0]
```



We can observe from the above plot that the log-log plot of errors is along the line y=2*x therefore, the order of convergence for $x_0 = -4$ is **quadratic**

```
In [13]: def ddf(x):
             return(24*x**2-24)
         g= ddf(r2[-1])/df(r2[-1])
         print(g)
```

```
-1.5
```

We know that the order of convergence for newtons method is atleast quadratic and for $x = -3.0$ we have $g''(x) = \frac{f''(x)}{f'(x)} = -1.5 \neq 0$ therefore the order of convergence will be **quadratic.**

Problem 2
Function for second norm and power method

```
In [19]: def secondnorm(a):#for a vector
             n = (np.dot(a,a))**0.5
             return(n)

         def infnorm(a):
             if np.size(a)!=len(a):
```

```
            b=np.zeros(len(a))
            for i in range(0,len(a)):
                s=0
                for k in range(0,len(a)):
                    s=s+np.abs(a[i,k])
                b[i]=s
        else:
            b = np.abs(a)
        norm = np.max(b)
        return(norm)

    def power(A,x,N):
        X = np.copy(x)
        temp = np.zeros(len(x))
        muc = np.zeros(N)
        vc = np.zeros(N)
        mu = np.zeros(N)
        v = np.zeros((len(x),N))
        for i in range(0,N):
            temp = X
            X = np.dot(A,X)
            mu[i] = X[0]/temp[0];
            v[:,i] = X/(secondnorm(X))#normalizing using second norm
            if i>0:
                muc[i] = np.abs(mu[i]-mu[i-1])
                vc[i] = infnorm(v[:,i]-v[:,i-1])
            else:
                muc[i] = np.abs(mu[i])
                vc[i] = infnorm(v[:,i])
        print('The largest eigen value of R is '+str(mu[-1])
              +'\nand the principal component is \n'+str(v[:,-1]))
        return(mu,v,muc,vc)
```

In [20]:
```
R=np.array([[1.00,0.91,0.82, 0.70, 0.69, 0.60, 0.70, 0.50],
            [0.91, 1.00, 0.85, 0.80, 0.77, 0.60, 0.69, 0.60],
            [0.82, 0.85, 1.00, 0.90, 0.83, 0.77, 0.78, 0.67],
            [0.70, 0.80, 0.90, 1.00, 0.97, 0.85, 0.87, 0.79],
            [0.69, 0.77, 0.83, 0.97, 1.00, 0.92, 0.95, 0.80],
            [0.60, 0.60, 0.77, 0.85, 0.92, 1.00, 0.97, 0.92],
            [0.70, 0.69, 0.78, 0.87, 0.95, 0.97, 1.00, 0.94],
            [0.50, 0.60, 0.67, 0.79, 0.80, 0.92, 0.94, 1.00]])
N=10;
x0=np.array([1.,1.,1.,1.,1.,1.,1.,1.])
lam=power(R,x0,N)
```

```
The largest eigen value of R is 6.56032995939
and the principal component is
[ 0.31665391  0.3333308   0.35665523  0.37223543  0.37533722  0.35939018
```

```
 0.37327524  0.33687531]
```

(b) Percentage of variation accounted for by the principal component is:

```
In [16]: mu=lam[0]
         v  = mu[-1]/8
         print(v)
```

```
0.820041244924
```
