# Midterm Numerical Analysis

Lena Siemer

October 2023

# 1 Problem 1

Use Taylor theorem to derive the error term for the approximate formula

$$f'(x) \approx \frac{1}{2h}(-3f(x) + 4f(x+h) - f(x+2h)).$$

**Solution:** Taylor's Theorem
*If $f \in C^n(a,b)$ and $f^{(n+1)}$ exists on $(a,b)$, then for any point c,x in $[a,b]$*

$$f(x) = \sum_{k=0}^{N} \frac{1}{k!} f^{(k)}(c)(x-c)^k + E_n(x),$$

*where*

$$T_n(x) = \sum_{k=0}^{N} \frac{1}{k!} f^{(k)}(c)(x-c)^k$$

*is the Taylor polynomial of $f(x)$ of degree n and*

$$E_n(x) = \frac{1}{(n+1)!} f^{(n+1)}(\xi)(x-c)^{n+1}$$

*where $\xi$ is a point between c and x.*

lecture notes page 6

So if we take the Taylor polynom of order 1 we get

$$f(h) = f(x) + f'(x)(h-x) + E_1(h)$$

where $E_1(h)$ is the error term.
So we get

$$f(x+h) = f(x) + hf'(x) + E_1(h),$$
$$f(x+2h) = f(x) + 2hf'(x) + E_1(2h).$$

If we use Taylor's theorem on the right side of the approximation, we get

$$\begin{aligned}
\frac{1}{2h}(-3f(x) + 4f(x+h) - f(x+2h)) &= \frac{1}{2h}(-3f(x) + 4(f(x) + hf'(x) + E_1(h)) \\
&\quad - (f(x) + 2hf'(x) + E_1(2h)) \\
&= \frac{1}{2h}(4hf'(x) - 2hf'(x) + E_2(h) - E_2(2h)) \\
&= f'(x) + \frac{1}{2h}(4\frac{1}{2}f^{(2)}(\xi_1)(h)^2 - \frac{1}{2}f^{(2)}(\xi_2)(2h)^2) \\
&= f'(x) + (hf^{(2)}(\xi_1) - hf^{(2)}(\xi_2)) \\
&= f'(x) + h(f^{(2)}(\xi_1) - f^{(2)}(\xi_2))
\end{aligned}$$

So the error term is

$$h(f^{(2)}(\xi_1) - f^{(2)}(\xi_2)).$$

## 2 Problem 2

Consider the following variation of the Newton's method:

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_0)}.$$

Find constants C and s such that

$$e_{n+1} = \frac{Ce^s}{n}.$$

**Solution:** We know

$$e_{n+1} = x_{n+1} - r = x_n - r - \frac{f(x_n)}{f'(x_0)} = e_n - \frac{f(x_n)}{f'(x_0)} = \frac{e_n f'(x_0) - f(x_n)}{f'(x_0)}$$

where $f(r) = 0$ is the point we are looking for. Using Taylor's theorem, we get

$$0 = f(r) = f(x_n - e_n) = f(x_n) - e_n f'(x_n) + E_n(x_n)$$

with $E_n(x) = \frac{1}{2}e_n^2 f"(\xi_n)$. Therefore we get

$$0 = f(r) = f(x_0 - e_0) = f(x_0) - e_0 f'(x_0) + \frac{1}{2}e_0^2 f"(\xi_0)$$

$$\rightarrow f'(x_0) = \frac{f(x_0) + \frac{1}{2}e_0^2 f"(\xi_0)}{e_0}$$

$$\rightarrow e_{n+1} = \frac{e_0^{-1} e_n (f(x_o) + \frac{1}{2}e_0^2 f"(\xi_0)) - f(x_n)}{f'(x_0)} = \frac{e_n f(x_0)}{e_0 f'(x_0)} + \frac{e_n e_0 f"(\xi_0)}{2f'(x_0)} - \frac{f(x_n)}{f'(x_0)}$$

So, if $x_0, x_n$ are close to $r$

$$e_{n+1} = \frac{e_0 f"(r)}{2f'(r)} e_n + \frac{f(r)}{f'(r)}(\frac{e_n}{e_0} - 1) \approx \frac{e_0 f"(r)}{2f'(r)} e_n$$

Therefore $C = \frac{e_0 f"(r)}{2f'(r)}$ and $s = 1$.

# 3 Problem 3

Consider the linear system Ax = b, where

$$A = \begin{pmatrix} 6.25 & -1 & 0.5 \\ -1 & 5 & 2.12 \\ 0.5 & 2.12 & 3.6 \end{pmatrix},$$

and

$$b = \begin{pmatrix} 7.5 \\ -8.68 \\ -0.24 \end{pmatrix}.$$

Write a computer program for LU-factorization with a unit lower triangular L (meaning that the diagonal entries should be equal to one). Then write a program for the Cholesky factorization. WARNING: avoid using shortcuts. The programming should be done "from scratch".

**Solution:** First I'd like to introduce three methods which we will need to implement the factorization.

---

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

#Method to print a matrix A:
def Ausgabe(A):
    n = len(A)
    for j in range(0,n):
        for i in range(0,n):
            print(A[j][i])
        print('\\n')

#Creates a matrix:
def Matrix(n):
    Matrix=[]
    #we go through the lines
    for x in range (0,n):
        line=[]
        # we go through the columns
        for y in range (0,n):
            if x==y:
                line.append(1)
            else:
                line.append(0)
        Matrix.append(Zeile)
    return Matrix
```

```

#creates a vector
def Vector(n):
    vec = []
    for i in range (0,n):
        vec.append(0)
    return vec

```

---

We begin with the LU factorization.

---

```
def LU_factorisation(A):
    n=len(A)
    B=Matrix(n)
    for x in range (1,n):
        for y in range (0,x):
            if A[y][y]==0:
                #we have to switch rows
                for q in range (y,n):
                    if A[q][y]!=0:
                        for r in range(0,n):
                            t= A[y][r]
                            A[y][r]=A[x][r]
                            A[x][r]=t
                        break
            q = A[x][y]/A[y][y]
            B[x][y]=q
            for z in range(0,n):
                A[x][z]= A[x][z]-A[y][z]*q
                #B[x][z]=-q


    return B

A = Matrix(3)
A[0][0] = 6.25
A[0][1] = -1
A[0][2] = 0.5
A[1][0] = -1
A[1][1] = 5
A[1][2] = 2.12
A[2][0] = 0.5
A[2][1] = 2.12
A[2][2] = 3.6

print (len(A))
L= LU_factorisation(A)
```

```
print("Print matrix R:")
Ausgabe(A)
print("Print matrix L:")
Ausgabe (L)

```

---

We get

$$L = \begin{pmatrix} 1 & 0 & 0 \\ -0.16 & 1 & 0 \\ 0.08 & 0.45 & 1 \end{pmatrix},$$

and

$$R = \begin{pmatrix} 6.25 & -1 & 0.5 \\ 0 & 4.84 & 2.2 \\ 0 & 0 & 2.56 \end{pmatrix}.$$

Now we want to solve the linear system $Ax = b$. So we need to implement a new function that uses the LR factorisation

---

```
def solve(A,b):
    n=len(A)
    L = LU_factorisation(A)

    #first we solve Lx=b
    x = Vector(len(A))
    for i in range (0, len(A)):
        sum=0
        for j in range(0,i):
            sum+=L[i][j]*x[j]
        x[i] = b[i] - sum

    # then we solve Uy=x
    y=Vector(len(A))
    for i in range(len(A)-1,-1,-1):
        sum=0
        for j in range(len(A)-1,i,-1):
            sum+=A[i][j]*y[j]
        y[i] = (x[i] -sum)/A[i][i]
    return y

#Solve the linear system:
b = [7.5,-8.68,-0.24]
x = solve(A,b)
print(x)
```

---

We obtain the solution of the linear system which is

$$x = \begin{pmatrix} 0.8 \\ -2 \\ 1 \end{pmatrix}.$$

Next we want to implement the Cholesky factorization.Again, we are using the three methods from the begining.

```
def cholesky(A):
    n = len(A)

    L=Matrix(n)
    for k in range (0,n):
        # first we calculate the element on the diagonal
        sum = 0
        for s in range(0,k):
            sum=sum+(L[k][s]*L[k][s])
        L[k][k]=math.sqrt(A[k][k]-sum)
        # then we calculate the lower triangular matrix
        for i in range(k,n):
            sum = 0
            for s in range(0,k):
                sum = sum + L[i][s]*L[k][s]
            L[i][k]=(A[i][k]-sum)/L[k][k]
    return L

A = Matrix(3)
A[0][0] = 6.25
A[0][1] = -1
A[0][2] = 0.5
A[1][0] = -1
A[1][1] = 5
A[1][2] = 2.12
A[2][0] = 0.5
A[2][1] = 2.12
A[2][2] = 3.6

L=cholesky(A)
print("Print matrix L:")
Ausgabe(L)

```

We get

$$L = \begin{pmatrix} 2.5 & 0 & 0 \\ -0.4 & 2.2 & 0 \\ 0.2 & 1 & 1.6 \end{pmatrix}.$$

Lastly we solve the linear system through the Cholesky factorisation.

```
def solve(A,b):
    n=len(A)
    L = cholesky(A)
    L2 = transpose(L)

    # we want to print the matrices:
    #matrix L
    print("L")
    Ausgabe(L)
    #matrix R
    print("A")
    Ausgabe(L2)


    #first we solve Lx=b
    x = Vector(len(A))
    for i in range (0, len(A)):
        sum=0
        for j in range(0,i):
            sum+=L[i][j]*x[j]
        x[i] = (b[i] - sum)/L[i][i]
    # then we solve Uy=x
    print(x)
    y=Vector(len(A))
    for i in range(len(A)-1,-1,-1):
        sum=0
        for j in range(len(A)-1,i,-1):
            sum+=L2[i][j]*y[j]
        y[i] = (x[i] -sum)/L2[i][i]
    return y

#solve the linear system
b = [7.5,-8.68,-0.24]
print(solve(A,b))
```

As expected, we optain

$$R = \begin{pmatrix} 0.8 \\ -2 \\ 1 \end{pmatrix}.$$

# 4 Problem 4

Problem 4. Consider the equation

$$f(x) = 0,$$

where

$$f(x) = x - \frac{1}{2}\left(\cos(\frac{x}{2}) - \left|x - \frac{1}{2}\right|\right).$$

Do the following.

1. Rewrite the equation as a fixed point problem for a function $T(x)$ related to $f(x)$. Prove that $T(x)$ (i) maps the interval $[0.45, 0.55]$ into itself, and (ii) $T(x)$ is contractive on this interval. Based on the theorems in the books and notes, make conclusions about existence and uniqueness of a solution to $f(x) = 0$ .

2. Write the computer program implementing the fixed point iteration $x_{n+1} = T(x_n)$. Discuss the choice of an initial guess $x_0$. Calculate the solution to within 10 decimal places. Note the number of iterations needed.

3. Use a priori and a posteriori error estimates from the notes to estimate the number of iterations needed to obtain the above accuracy. Which estimate is more accurate?

4. Implement the Newton's method on a computer for the same equation starting with $x_0$ within the interval $[0.45, 0.55]$. Discuss the theoretical difficulty related to non-existence of $f'(x)$ for certain point(s) within the interval. How would you set up the Newton's method to resolve the problem?

5. Note the number of iterations needed to calculate the solution to within 10 decimal places using the Newton's method. Discuss the speed of convergence of the Newton's method and compare it with the convergence speed of the fixed point iterations.

6. Implement the alternative error control strategy proposed by you in HW 3, Problem 2. Compare the results with the ones obtained using a priori and a posteriori error estimates.

**Solution:**

1. First we rewrite the equation as a fixed point problem for a function $T(x)$ related to $f(x)$. We have

$$f(x) = 0$$

$$\rightarrow 0 = x - \frac{1}{2}\left(\cos(\frac{x}{2}) - \left|x - \frac{1}{2}\right|\right)$$

$$\rightarrow x = \frac{1}{2}\left(\cos(\frac{x}{2}) - \left|x - \frac{1}{2}\right|\right).$$

It follows that

$$T(x) = \frac{1}{2}\left(\cos(\frac{x}{2}) - \left|x - \frac{1}{2}\right|\right)$$

(i) To find maximum and minimum we need the derivative

$$T'(x) = -\frac{1}{2}(-\frac{1}{2}\sin(\frac{x}{2}) - 1).$$

Note that there exists no derivative for $x = 0.5$. Therefore we look at the intervals $(0.45, 0.5)$ and $(0.5, 0.55)$ separately.

$$0 = -0.5(0.5\sin(\frac{x}{2}) - 1)$$

doesn't give us a solution in $(0.45, 0.5)$ or $(0.5, 0.55)$, so the maximum and minimum of each interval has to be at the border of the intervals.

$$T(0.45) = 0.462$$

$$T(0.5) = 0.484$$

$$T(0.55) = 0.456$$

So the maximum of $T(x)$ is 0.484 and the minimum is 0.456. Therefore

$$0.45 < 0.456 \leq T(x) \leq 0.484 < 0.55$$

(ii) A mapping(or function) F is said to be contractive of there exists a number $\lambda < 1$ such that

$$|F(x) - F(y)| \leq \lambda\,|x - y|$$

for all points x and y in the domain of F .

$$\begin{aligned}
\rightarrow |F(x) - F(y)| &= \frac{1}{2}\left|\cos(\frac{x}{2}) - \cos(\frac{y}{2}) - (|x - 0.5| - |y - 0.5|)\right| \\
&\leq 0.5\left|\cos(\frac{x}{2}) - \cos(\frac{y}{2}) - |x - 0.5 - y + 0.5|\right| \\
&= 0.5\left|\cos(\frac{x}{2}) - \cos(\frac{y}{2}) - |x - y|\right| \\
&\leq \frac{1}{2}\,|x - y|
\end{aligned}$$

Theorem 1 (Contractive Mapping Theorem)
*Let C be a closed subset of the real line. If F is a contractive mapping of C into C, then F has a unique fixed point. Moreover, this fixed point is*

*the limit of every sequence obtained from* $x_{n=1} = F(x_n)$ *with a starting point* $x_0 \in C$

Numerical Analysis by D. Kincaid and W. Cheney page 102 Theorem 1

From the theorem it follows directly that $T(x)$ has a unique fixed point in $[0.45, 0.55]$. Therefore there exists a unique $x$ such that $f(x) = 0$.

2. from part 1) we already know that [0.45,0.55] is contractive, so my initial guess would be to start in that interval: Theorem 1 states that if T is a contactive mapping from C into C, than F has a unique fixed point. So we know about the existence of the fixed point. Moreover, we know that if we start in [0.45,0.55] that the limes of the fixed point iteration is that fixed point. So if we start in [0.45,0.55] than our sequence has a finite limes and that limes is the searched point.
Then we can implement the fixed point iteration:

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

plt.axis((0.4,0.6, 15, 30))

def f(x):
    #that's the iterative function
    return 0.5*(math.cos(x/2)-abs(x-0.5))


def FixedPointIteration(x,n):
    #i denotes the number of iterations
    i=0
    while abs(f(x)-x)>n:
        x=f(x)

        i+=1
    return [x,i]

print(FixedPointIteration(0.45,0.5*10**(-10))[0])
```

We get $x = 0.4722515913$.
The number of iterations depends on our starting point. For all $x_0 \in [0.45, 0.55]$ we will get the same fixed point x, but might need more iterations. We can calculate the number of iterations with our methods, see Figure 1.

```
for i in range(1,100):
    L = FixedPointIteration(0.45+0.001*i,0.5*10**(-10))
```

```
    plt.scatter(0.45+0.001*i,L[1])

plt.show()
```

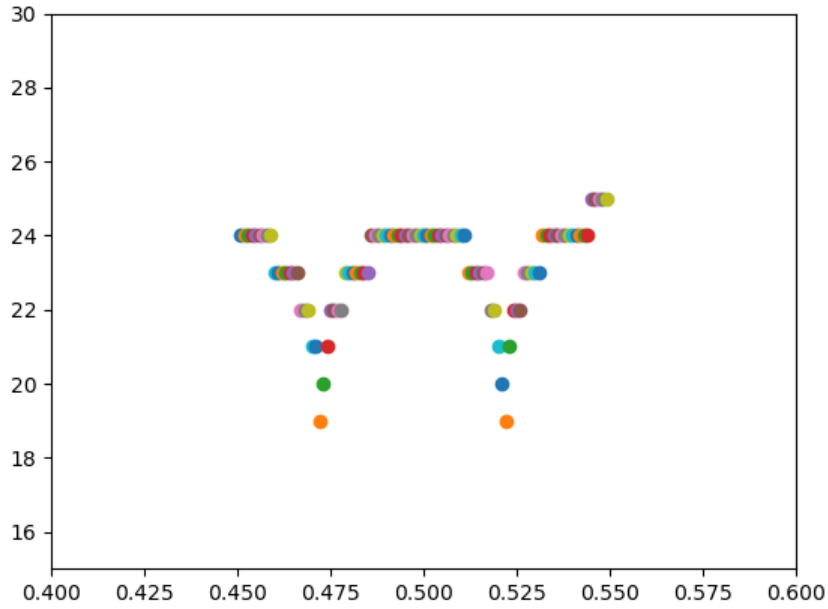


Figure 1: Number of iterations for the Fixed Point iteration

3. A priori error estimate $\rightarrow |x_n - x_{n+m}| \leq \frac{\lambda^n}{1-\lambda} |x_1 - x_0| < \epsilon$
   A posteriori error estimate $\rightarrow |x^* - x_{n+1}| \leq \frac{\lambda}{1-\lambda} |x_{n+1} - x_n| < \epsilon$
   In both cases the minimum number of iterations we need would be the smallest $n$ such that the inequalities are true.
   From the a priori estimate we get

$$n > \frac{\ln\left(\frac{\epsilon(1-\lambda)}{|x_1 - x_0|}\right)}{\ln(\lambda)}.$$

   From the lecture we know that $\lambda = \max_{[0.45,0.55]} |T'(x)|$. So we first need to calculate $\lambda$. We know

$$T'(x) = -\frac{1}{2}(-\frac{1}{2}\sin(\frac{x}{2}) - 1).$$

To calculate $\lambda$ we need to find the max of $T'(x)$, so we need to find a zero in $[0.45, 0.55]$ of

$$T"(x) = \frac{1}{8}\cos(\frac{x}{2}).$$

With a calculator we get that $F"(x)$ doesn't have a zero in that interval. So $|T'(x)|$ takes it maximum at the border.

$$T'(0.45) = 0.556$$

$$T'(0.55) = 0.568$$

So $\lambda = max_{[0.45,0.55]}|T'(x)| = 0.568$

**A priori:** Now we can implement the a priori error estimate:

```
1   import math
2   import matplotlib.pyplot as plt
3   import numpy as np
4
5   plt.axis((0.4,0.6, 25, 45))
6
7   def f(x):
8       #that's the iterative function
9       return 0.5*(math.cos(x/2)-abs(x-0.5))
10
11
12  def FixedPointIteration(x,n):
13          #i denotes the number of iterations
14      i=0
15      x0=x
16      while abs(f(x)-x)>((0.568**(i+1))/(1-0.568))* abs(x0-f(x0))
17                  or ((0.568**(i))/(1-0.568))* abs(x0-f(x0))>n:
18          x=f(x)
19          i+=1
20      return [x,i]
21
22
23
24  print(FixedPointIteration(0.45,0.5*10**(-10))[0])
25
26  for i in range(1,100):
27      L = FixedPointIteration(0.45+0.001*i,0.5*10**(-10))
28      plt.scatter(0.45+0.001*i,L[1])
29
30
31  plt.show()
```

We get $x = 0.4722515914$ and the numbers of iterations for starting points in $[0.45, 0.55]$, see Figure 2.

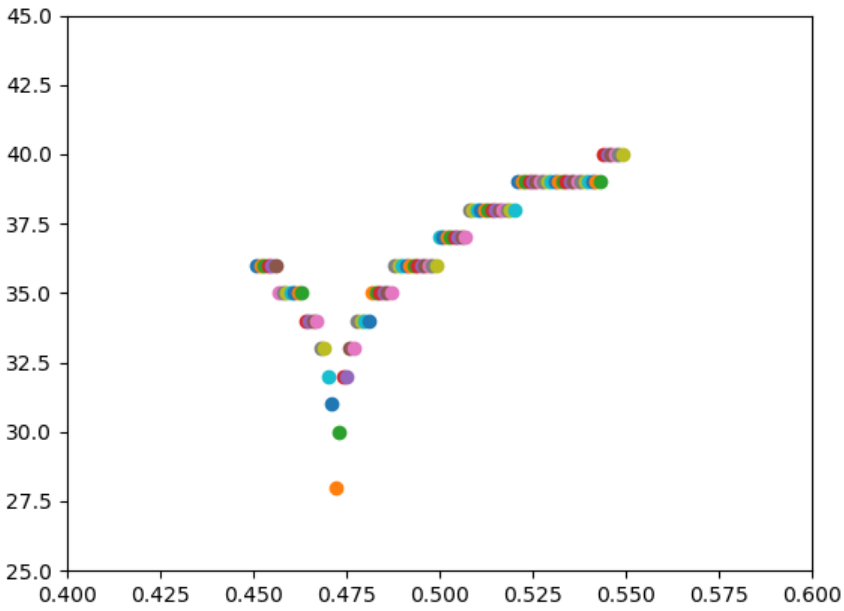


Figure 2: Number of iterations with the A Priori estimate

**A posteriori:**

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

def f(x):
    #that's the iterative function
    return 0.5*(math.cos(x/2)-abs(x-0.5))


def FixedPointIteration(x,n):
    #i denotes the number of iterations
    i=0
    while abs(0.4722515914-f(x))>(0.568/(1-0.568))*abs(f(x)-x)
                or (0.568/(1-0.568))*abs(f(x)-x)>n:
        x=f(x)

        i+=1
    return [x,i]

```

```
20
21      print(FixedPointIteration(0.45,0.5*10**(-10))[0])
```

We get $x = 0.0.4722515913$ for starting point $x = 0.45$.
The problem, I had with the a posteriori error estimate, was that $|f(x) - x| = 0$ at some point due to rounding errors. But then

$$|0.4722515914 - f(x)| > \frac{0.568}{1 - 0.568}|f(x) - x|$$

for all future values. So our program never terminates. In addition $x_*$ might not be exact enough.

The a priori error estimation calculates the error before obtaining the solution while the a posteriori uses the solution to calculate an more accurate error. Both estimates calculate the distance from the current state $x_{n+1}$ or $x_{n+m}$ to a past point $x_n$. The A priori estimate compares it with the starting points. The A posteriori estimate compares it with the distance from the current point to the solution. Therefore it is more accurate to use the a posteriori error estimation.

4. The problem is that the derivative at $x = \frac{1}{2}$ does not exist. In case we have $x = 0.5$ we can't calculate the derivative of f at $x = \frac{1}{2}$. What we can do instead of calculating $dev\_f(x_n)$, we can take $x_{n-1}$ and use the derivative of that value instead. This might increase the number of iterations we need, but we will receive the correct result.

```
1   import math
2   import matplotlib.pyplot as plt
3   import numpy as np
4
5   # the function f
6   def f(x):
7       #that's the iterative function
8       return x-0.5*(math.cos(x/2)- abs(x-0.5))
9
10  # the derivative of f
11  def dev_f(x):
12      return 1-0.5*(0.5*math.sin(x/2)-1)
13
14
15  def NewtonMethod(x,n):
16      i=0
17      if x == 0.5:
18          return
19      x0=x - f(x)/dev_f(x)
20      while abs(x-x0)>n:
```

```
        if x == 0.5:
            x = x - f(x)/dev_f(x0)
        else:
            x0=x
            x = x - f(x)/dev_f(x)
            i+=1
    return [x,i]

print(NewtonMethod(0.45,0.5*10**(-10))[0])
```

As expected we get $x = 0.4722515913$

5. The number of iterations depends on our starting point. I am using the following code with the methods from before to calculate the number of iterations, see Figure 3.

```
plt.axis((0.4,0.45, 30, 60))

print(NewtonMethod(0.45,0.5*10**(-10))[0])
for i in range(1,100):
    if 0.001*i!= 0.05:
        L=NewtonMethod(0.45+0.001*i,0.5*10**(-10))
        plt.scatter(0.45+0.001*i,L[1])

plt.show()
```

We have shown in past homeworks that Newton's method has quadratic convergence.
With Theorem 1 we now that $|x_* - x_{n+1}| \leq q|x_* - x_1|$ for $0 \leq q \leq 1$. It follows that the Fixed Point Iteration has linear convergence.
Therefore we can conclude that Newton's method has a faster convergence speed.

6. I implemented the alternative error control strategy proposed in HW 3, Problem 2 in part 2. The results for $x$ is the same for all error control strategies accept the $10^{th}$ decimal place.
My error control strategy calculates $|x_{n+1} - x_n| < \epsilon$, so the distance between the current point and the past point and tests if they are close enough. The a priori estimate compares it to the distance of the starting points. So if they are significant closer than the starting point then the algorithm terminates. The a posteriori goes one step further and tests if the current point is closer to the actual solution than the last point. Therefore it is the most accurate estimate.

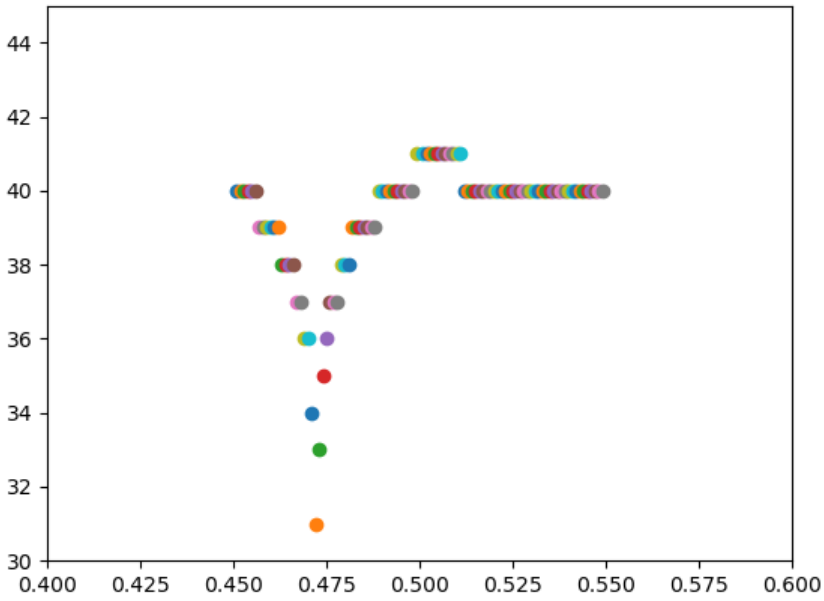


Figure 3: Number of iterations for Newton's method

# 5 Problem 5

Implement and analyze the Ridders' method for solving the equation $f(x) = 0$. For guidance, you might wish to read the Wikipedia page on the Ridders' method. Also feel free to consult other sources.
**Brief Description of the method.** Given the initial bracketing interval $[x0, x2]$, containing one solution, and such that $f(x_0)$ and $f(x_2)$ have opposite signs, compute the midpoint $x_1 = \frac{x_0+x_2}{2}$. Then define a new function $h(x) = f(x)e^{ax}$. Find the parameter a that ensures $h(x_1) = \frac{1}{2}(h(x_0) + h(x_2))$. Then define $x_3$ as the x-intercept of the line passing through $(x_0, h(x_0))$ and $(x_2, h(x_2))$. Use $x_3$ as one of the endpoints of the next interval bracketing the solution. The other endpoint is $x_1$ if $f(x_1)f(x_3) < 0$. Otherwise, choose either $x_0$ or $x_2$ based on the requirement that the sign of $f(x)$ at the chosen point must be opposite to the sign of $f(x_3)$. Continue until the desired accuracy is reached.
Questions and items to implement

1. Show that a is uniquely defined and give the equation for finding it.
2. Work out a formula for x3 in detail.
3. Implement Ridders' method on a computer together with the standard bisection method for finding all solutions of
$$e^x - x^2 = 0.$$
4. Numerically, compare the convergence rate of both algorithms. Discuss the results. What is the approximate order of convergence for the Ridders' algorithm?

**Solution:**

1. We have
$$h(x_1) = \frac{1}{2}(h(x_0) + h(x_2))$$
$$\rightarrow f(x_1)e^{ax} = \frac{1}{2}(f(x_o)e^{ax_0} + f(x_2)e^{ax_2})$$
$$\rightarrow f(x_1)e^{\frac{a}{2}(x_0+x_2)} = \frac{1}{2}(f(x_o)e^{ax_0} + f(x_2)e^{ax_2})$$
$$\rightarrow f(x_1)e^{\frac{a}{2}(x_0+x_2)} = \frac{1}{2}(f(x_o)e^{ax_0} + f(x_2)e^{ax_2})$$
$$\rightarrow f(x_1) = \frac{1}{2}(f(x_o)e^{ax_0-\frac{a}{2}(x_0+x_2)} + f(x_2)e^{ax_2-\frac{a}{2}(x_0+x_2)})$$
$$\rightarrow f(x_1) = \frac{1}{2}(f(x_o)e^{\frac{a}{2}(x_0-x_2)} + f(x_2)e^{\frac{a}{2}(x_2-x_0)})$$
$$\rightarrow 2f(x_1) = f(x_o)e^{-\frac{a}{2}(x_2-x_0)} + f(x_2)e^{\frac{a}{2}(x_2-x_0)}$$

$$\rightarrow 2f(x_1)e^{\frac{a}{2}(x_2-x_0)} = f(x_o) + f(x_2)(e^{\frac{a}{2}(x_2-x_0)})^2$$
$$\rightarrow 0 = f(x_o) - 2f(x_1)e^{\frac{a}{2}(x_2-x_0)} + f(x_2)(e^{\frac{a}{2}(x_2-x_0)})^2$$

This is a quadratic function. Set $y := e^{\frac{a}{2}(x_2-x_0)}$. Then we can solve

$$0 = f(x_0) - 2f(x_1)y + f(x_2)y^2$$

With the abc-formular we get

$$y_{1/2} = \frac{2f(x_1) \pm \sqrt{4f^2(x_1) - 4f(x_0)f(x_2)}}{2f(x_2)}$$

$$\rightarrow y_{1/2} = \frac{f(x_1) \pm \sqrt{f^2(x_1) - f(x_0)f(x_2)}}{f(x_2)}$$

Note: $f(x_0)f(x_2) < 0 \Rightarrow f^2(x_1) - f(x_0)f(x_2) \geq 0$

$$\rightarrow e^{a_{1/2}\frac{x_2-x_0}{2}} = \frac{f(x_1) \pm \sqrt{f^2(x_1) - f(x_0)f(x_2)}}{f(x_2)}$$

Therefore a is uniquely determined by

$$e^{a_{1/2}\frac{x_2-x_0}{2}} = \frac{f(x_1) - sign[f(x_0)]\sqrt{f^2(x_1) - f(x_0)f(x_2)}}{f(x_2)}$$

2. We are looking for a secant between $(x_0, h(x_0))$ and $(x_2, h(x_2))$. Since

$$h(x_1) = \frac{1}{2}(h(x_0) + h(x_2))$$

and

$$x_1 = \frac{1}{2}(x_0 + x_2)$$

we can take the secant between $(x_1, h(x_1))$ and $(x_0, h(x_0))$ instead. Let's find the secant first.
We start with

$$g(x) = mx + c.$$

Take

$$f(x_0) = mx_0 + c$$
$$f(x_1) = mx_1 + c.$$

Solving this linear system gives us

$$m = \frac{h(x_1) - h(x_0)}{x_1 - x_0}$$

and

$$c = h(x_0) - x_0\frac{h(x_1) - h(x_0)}{x_1 - x_0}.$$

To find the x-intercept of g(x), we look for $x_3$ such that

$$0 = g(x_3) = x_3 \frac{h(x_1) - h(x_0)}{x_1 - x_0} + h(x_0) - x_0 \frac{h(x_1) - h(x_0)}{x_1 - x_0}$$

$$\Rightarrow x_3 = x_0 - h(x_0) \frac{x_1 - x_0}{h(x_1) - h(x_0)}$$
$$= \frac{x_0 h(x_1) - x_1 h(x_0)}{h(x_1) - h(x_0)}$$

From part (1), we know that $h(x) = f(x)e^{ax}$. Define
$k = \sqrt{f^2(x_1) - f(x_0)f(x_2)}$

$$\begin{aligned}
x_3 &= \frac{f(x_0)e^{ax_0}x_1 - f(x_1)e^{ax_1}x_0}{f(x_0)e^{ax_0} - f(x_1)e^{ax_1}} \\
&= \frac{f(x_0)x_1 - f(x_1)e^{a(x_1-x_0)}x_0}{f(x_0) - f(x_1)e^{a(x_1-x_0)}} \\
&= \frac{f(x_0)x_1 - f(x_1)(\frac{f(x_1) - sign[f(x_0)]\sqrt{f^2(x_1)-f(x_0)f(x_2)}}{f(x_2)})x_0}{f(x_0) - f(x_1)(\frac{f(x_1) - sign[f(x_0)]\sqrt{f^2(x_1)-f(x_0)f(x_2)}}{f(x_2)})} \\
&= \frac{f(x_0)f(x_2)x_1 - f(x_1)(f(x_1) - sign[f(x_0)]k)x_0}{f(x_0)f(x_2) - f(x_1)(f(x_1) - sign[f(x_0)]k)} \\
&= \frac{f(x_0)f(x_2)x_1 - f^2(x_1)x_0 + sign[f(x_0)]f(x_1)kx_0}{f(x_0)f(x_2) - f^2(x_1) + sign[f(x_0)]k} \\
&= \left(\frac{x_1 f(x_0)f(x_2) - x_0 f^2(x_1) + sign(f(x_0))x_0 f(x_1)k}{f(x_0)f(x_2) - f^2(x_1) + sign(f(x_0))f(x_1)k}\right)1 \\
&= \left(\frac{x_1 f(x_0)f(x_2) - x_0 f^2(x_1) + sign(f(x_0))x_0 f(x_1)k}{f(x_0)f(x_2) - f^2(x_1) + sign(f(x_0))f(x_1)k}\right) \\
&\left(\frac{f(x_0)f(x_2) - f^2(x_1) - sign(f(x_0))f(x_1)k}{f(x_0)f(x_2) - f^2(x_1) - sign(f(x_0))f(x_1)k}\right) \\
&= \frac{x_1 f^2(x_0)f^2(x_2) - x_0 f^2(x_1)f(x_0)f(x_2) + sign(f(x_0))x_0 f(x_1)f(x_0)f(x_2)k+}{f^2(x_0)f^2(x_2) - f^2(x_1)f(x_0)f(x_2) + sign(f(x_0))f(x_1)f(x_0)f(x_2)k+} \\
&\frac{-x_1 f^2(x_1)f(x_0)f(x_2) + x_0 f^4(x_1) - sign(f(x_0))x_0 f^3(x_1)k - sign(f(x_0))x_1 f(x_0)f(x_1)f(x_2)k+}{-f^2(x_1)f(x_0)f(x_2) + f^4(x_1) - sign(f(x_0))f^3(x_1)k - sign(f(x_0))f(x_0)f(x_1)f(x_2)k+} \\
&\frac{+sign(f(x_0))x_0 f^3(x_1)k - x_0 f^2(x_1)(f^2(x_1) - f(x_0)f(x_2))}{+sign(f(x_0))f^3(x_1)k - f^2(x_1)(f^2(x_1) - f(x_0)f(x_2))} \\
&= \frac{x_1 f^2(x_0)f^2(x_2) + sign(f(x_0))(x_0 - x_1)f(x_0)f(x_1)f(x_2)k - x_1 f(x_0)f^2(x_1)f(x_2)}{f^2(x_0)f^2(x_2) - f(x_0)f^2(x_1)f(x_2)} \\
&= \frac{f(x_0)f(x_2)[x_1 f(x_0)f(x_2) + sign(f(x_0))(x_0 - x_1)f(x_1)k - x_1 f^2(x_1)]}{f(x_0)f(x_2)[f(x_0)f(x_2) - f^2(x_1)]}
\end{aligned}$$

$$
\begin{aligned}
&= x_1 \frac{f(x_0)f(x_2) - f^2(x_1)}{f(x_0)f(x_2) - f^2(x_1)} + \frac{sign(f(x_0))(x_0 - x_1)f(x_1)k}{f(x_0)f(x_2) - f^2(x_1)} \\
&= x_1 + \frac{sign(f(x_0))(x_1 - x_0)f(x_1)}{\frac{f^2(x_1) - f(x_0)f(x_2)}{k}} \\
&= x_1 + \frac{sign(f(x_0))(x_1 - x_0)f(x_1)}{\sqrt{f^2(x_1) - f(x_0)f(x_2)}}
\end{aligned}
$$

3. Ridders method:

```
import math


def f(x):
    #function
    return math.exp(x)-x**2


def ridders_method(x0, x2):
    for i in range(0,1000):
        x1 = (x0+x2)/2
        x = -1
        if f(x0):
            x =1
        #h = ((f(x1)-x*math.sqrt(f(x1)**2-f(x0)*f(x2)))/f(x2))**(2/(x2-x0))
        x3= x1 + (x1-x0)*(x*f(x1))/math.sqrt(f(x1)**2-f(x0)*f(x2))
        if f(x1)*f(x3)<0:
            x2 = x3
            x0 = x1
        elif f(x0)*f(x3)<0:
            x2 = x3
        elif f(x2)*f(x3)<0:
            x0 = x3
    return x3


print(ridders_method(-1,0))

```

We get $x = -0.7034674224983917$.

Bisection method:

```
    import math


```

```
    def f(x):
        return math.exp(x)-x**2


    def bisection_method(min, max, error):
        if max-min<error:
            return (min+max)/2
        elif f(min)==0:
            return min
        elif f(max)==0:
            return max
        elif f(min)*f(max)<0:
            d = (max+min)/2
            #print("d", d)
            if f(min)*f(d)<=0:
                return bisection_method(min,d,error)
            elif f(d)*f(max)<=0:
                return bisection_method(d,max,error)
        else:
            print("there is no root in this interval")
            return 0



    x = bisection_method(-1,0, 10**(-15))
    print(x)
```

We get $x = -0.7034674224983917$.

4. **Order of convergence bisection method**
   Claim: The bisection method converges linearly
   Proof: On Homework 2 we have shown that $\lim(\frac{|x_* - x_{n+1}|}{|x_* - x_n|}) = \frac{1}{2}$.
   Therefore the order of convergence is 1.

   **Order of convergence Ridders method** I found online that Ridder's method should converge quadraticly:
   *https://math.stackexchange.com/questions/1596654/why-does-ridders-method-work-as-well-as-it-does*
   To this point, I wasn't able to proof it.

   Therefore it should be faster than the bisection method.