TMA4135 Matematikk 4D
Exercise 9

Odd André Owren

November 2019

1a. See figure 1. One can see that when h is halved, e(h) is reduced by approximately $\frac{1}{16}$, or $\frac{1}{2^4}$. This means that we have an order of 4, which is what we wanted to find.

```
In [10]:
def approximate_for_different_h(f, a, b, exact):
    n = 1
    h = (b - a ) / n
    steps = []
    errors = []
    Nmax = 10
    for k in range(Nmax):
        simp = simpson(f, a, b, n)
        eh = abs(exact - simp)
        print(f'h = {h:8.2e},   T(h) = {simp:10.8f},   e(h) = {eh:8.2e}')
        steps.append(h)
        errors.append(eh)
        n = 2 * n
        h = (b - a) / n

    print('\nOrder P and constant C')
    for k in range(1, Nmax-1):
        p = log(errors[k+1]/errors[k])/log(steps[k+1]/steps[k])
        C = errors[k+1]/steps[k+1]**p
        print(f'h = {steps[k]:8.2e},  p = {p:4.2f},  C = {C:6.4f}')

    clf()
    loglog(steps, errors, 'o-')
    xlabel('h')
    ylabel('e(h)')
    grid(True)
```

executed in 12ms, finished 13:48:06 2019-10-29

```
In [29]:
newparams = {'figure.figsize': (8.0, 4.0), 'axes.grid': True,
             'lines.markersize': 8, 'lines.linewidth': 2,
             'font.size': 14}
rcParams.update(newparams)

def f1(x):
    return x * exp(x)

a, b = -1, 1
exact = 2 / exp(1)
approximate_for_different_h(f1, a, b, exact)
```

executed in 1.53s, finished 14:28:31 2019-10-29

```
h = 2.00e+00,   T(h) = 0.78346746,   e(h) = 4.77e-02
h = 1.00e+00,   T(h) = 0.73913060,   e(h) = 3.37e-03
h = 5.00e-01,   T(h) = 0.73597650,   e(h) = 2.18e-04
h = 2.50e-01,   T(h) = 0.73577259,   e(h) = 1.37e-05
h = 1.25e-01,   T(h) = 0.73575974,   e(h) = 8.59e-07
h = 6.25e-02,   T(h) = 0.73575894,   e(h) = 5.37e-08
h = 3.12e-02,   T(h) = 0.73575889,   e(h) = 3.36e-09
h = 1.56e-02,   T(h) = 0.73575888,   e(h) = 2.10e-10
h = 7.81e-03,   T(h) = 0.73575888,   e(h) = 1.31e-11
h = 3.91e-03,   T(h) = 0.73575888,   e(h) = 8.19e-13

Order P and constant C
h = 1.00e+00,  p = 3.95,  C = 0.0034
h = 5.00e-01,  p = 3.99,  C = 0.0035
h = 2.50e-01,  p = 4.00,  C = 0.0035
h = 1.25e-01,  p = 4.00,  C = 0.0035
h = 6.25e-02,  p = 4.00,  C = 0.0035
h = 3.12e-02,  p = 4.00,  C = 0.0035
h = 1.56e-02,  p = 4.00,  C = 0.0035
h = 7.81e-03,  p = 4.00,  C = 0.0035
```

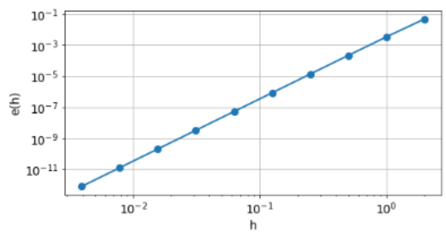

Figure 1: Numerical confirmation of composite Simpson's formula

1b. See figure 2. In this case we only have an order of 1.5, which is considerably lower than in a, and thus we have a lower precision. Plot of the integrals are given in figure 3.

In [13]:

```python
def f2(x):
    return sqrt(1 - x ** 2) * exp(x)

a, b = -1, 1
exact = 1.7754996892121809469
approximate_for_different_h(f2, a, b, exact)
```

executed in 1.86s, finished 13:49:25 2019-10-29

```
h = 2.00e+00,   T(h) = 1.33333333,   e(h) = 4.42e-01
h = 1.00e+00,   T(h) = 1.63540364,   e(h) = 1.40e-01
h = 5.00e-01,   T(h) = 1.72886020,   e(h) = 4.66e-02
h = 2.50e-01,   T(h) = 1.75945841,   e(h) = 1.60e-02
h = 1.25e-01,   T(h) = 1.76989907,   e(h) = 5.60e-03
h = 6.25e-02,   T(h) = 1.77353123,   e(h) = 1.97e-03
h = 3.12e-02,   T(h) = 1.77480571,   e(h) = 6.94e-04
h = 1.56e-02,   T(h) = 1.77525467,   e(h) = 2.45e-04
h = 7.81e-03,   T(h) = 1.77541312,   e(h) = 8.66e-05
h = 3.91e-03,   T(h) = 1.77546909,   e(h) = 3.06e-05

Order P and constant C
h = 1.00e+00,  p = 1.59,  C = 0.1401
h = 5.00e-01,  p = 1.54,  C = 0.1356
h = 2.50e-01,  p = 1.52,  C = 0.1316
h = 1.25e-01,  p = 1.51,  C = 0.1290
h = 6.25e-02,  p = 1.50,  C = 0.1274
h = 3.12e-02,  p = 1.50,  C = 0.1265
h = 1.56e-02,  p = 1.50,  C = 0.1260
h = 7.81e-03,  p = 1.50,  C = 0.1257
```

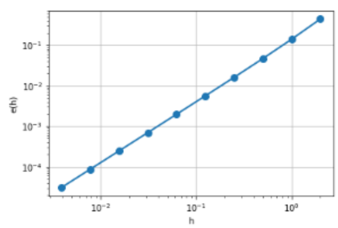

Figure 2: Repeated process from a) with new integral

```
In [16]: newparams = {'figure.figsize': (16.0, 8.0), 'axes.grid': True,
                      'lines.markersize': 8, 'lines.linewidth': 2,
                      'font.size': 14}
         rcParams.update(newparams)

         x = linspace(a, b, 101)

         subplot(1,2,1)
         plot(x, f1(x))
         xlabel('x')
         ylabel('f1(x)')

         subplot(1,2,2)
         plot(x, f2(x))
         xlabel('x')
         ylabel('f2(x)');
executed in 1.52s, finished 13:51:34 2019-10-29
```

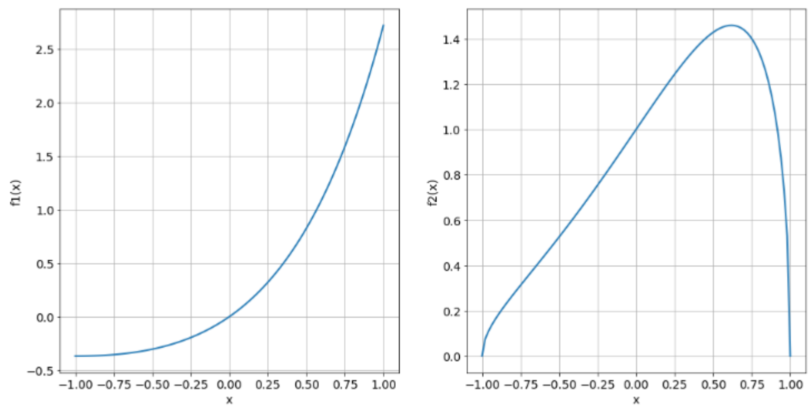

Figure 3: Plot of integrals

3a. We have the function $f(x) = e^x + x^2 - x - 4$. $f(1) = -1.2817$ and $f(2) = 5.3891$, thus f(x) has at least 1 solution for $f(x) = 0$.
Since $f'(x) = e^x + 2x - 1 > 0$ for $x \in [1, 2]$, f(x) is strictly growing, thus we have only one solution r for $f(x) = 0$.

In [17]:
```
def f(x):
    return exp(x) + x**2 - x - 4

x = linspace(1, 2, 101)
plot(x, f(x))
xlabel('x')
ylabel('f(x)');
```
executed in 865ms, finished 14:05:00 2019-10-29

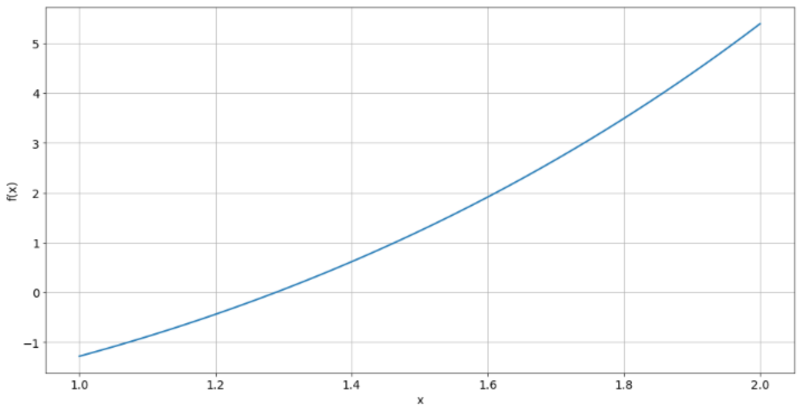

Figure 4: Plot of f(x)

From the plot in figure 4 we can see that x=1.3 is a sufficient starting point for Newton's method. Approximation with Newton's method gives us:

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)} \quad x_0 = 1.3$$

$$x_1 = 1.3 - \frac{f(1.3)}{f'(1.3)} \approx 1.28875$$

$$x_2 = 1.28875 - \frac{f(1.28875)}{f'(1.28875)} \approx 1.28868$$

$$x_3 = 1.28868 - \frac{f(1.28868)}{f'(1.28868)} \approx 1.28868$$

Thus the approximated solution with Newton's method is <u>x=1.28868</u>, which is the same as in the numerical solution with Jupyter, see figure 5.

```
In [18]: def newton(f, df, x0, tol=1.e-8, max_iter=30):
             x = x0
             print('k ={:3d}, x = {:18.15f}, f(x) = {:10.3e}'.format(0, x, f(x)))
             for k in range(max_iter):
                 fx = f(x)
                 if abs(fx) < tol:
                     break
                 x = x - fx/df(x)
                 print('k ={:3d}, x = {:18.15f}, f(x) = {:10.3e}'.format(k+1, x, f(x)))
             return x, k+1
executed in 11ms, finished 14:15:20 2019-10-29
```

```
In [19]: def f(x):
             return exp(x) + x ** 2 - x - 4

         def df(x):
             return exp(x) + 2 * x - 1

         x_0 = 1.3

         x, k = newton(f, df, x_0)

         print(f'\nAfter {k} iterations, the approximated solution found is {x:.5f}.')
executed in 9ms, finished 14:15:52 2019-10-29

k =  0, x =  1.300000000000000, f(x) =  5.930e-02
k =  1, x =  1.288746758560088, f(x) =  3.581e-04
k =  2, x =  1.288677969382023, f(x) =  1.332e-08
k =  3, x =  1.288677966823868, f(x) = -4.441e-16

After 4 iterations, the approximated solution found is 1.28868.
```

Figure 5: Solution with Newton's method

3b. See figure 6. It is clear to see that function 1 converges, while function 2 and 3 does not. Function 2 keeps "jumping" between 2 distinct values and never seems to converge, while function 3 diverges towards infinity and causes a crash in the Jupyter notebook.

```
In [22]: def fixpoint(g, x0, tol=1.e-6, max_iter=30):
             x = x0
             print('k ={:3d}, x = {:14.10f}'.format(0, x))
             for k in range(max_iter):
                 x_old = x                       # Store old values for error estimation
                 x = g(x)                        # The iteration
                 err = abs(x - x_old)              # Error estimate
                 print('k ={:3d}, x = {:14.10f}'.format(k + 1, x))
                 if err < tol:          # The solution is accepted
                     break
             return x, k+1
executed in 12ms, finished 14:19:29 2019-10-29
```

```
In [26]: def g1(x):
             return log(4 + x - x ** 2)

         def g2(x):
             return sqrt(-exp(x) + x + 4)

         def g3(x):
             return exp(x) + x ** 2 - 4


         gs = [g1, g2, g3]
         x_0 = 1.5


         for i, g in enumerate(gs, start=1):

             print(f'\nFunction {i}:')
             x, k = fixpoint(g, x_0)
executed in 22ms, finished 14:23:45 2019-10-29
```

```
Function 1:
k =  0, x =   1.5000000000
k =  1, x =   1.1786549963
k =  2, x =   1.3322149248
k =  3, x =   1.2690350905
k =  4, x =   1.2970764687
k =  5, x =   1.2850003170
k =  6, x =   1.2902719159
k =  7, x =   1.2879840175
k =  8, x =   1.2889795000
k =  9, x =   1.2885468343
k = 10, x =   1.2887349735
k = 11, x =   1.2886531806
k = 12, x =   1.2886887430
k = 13, x =   1.2886732816
k = 14, x =   1.2886800038
k = 15, x =   1.2886770812
k = 16, x =   1.2886783519
k = 17, x =   1.2886777994

Function 2:
k =  0, x =   1.5000000000
k =  1, x =   1.0091139329
k =  2, x =   1.5053054929
k =  3, x =   0.9998878264
k =  4, x =   1.5105995169
k =  5, x =   0.9905322092
k =  6, x =   1.5158710552
k =  7, x =   0.9810634053
k =  8, x =   1.5211088914
k =  9, x =   0.9714992449
k = 10, x =   1.5263017049
k = 11, x =   0.9618589916
k = 12, x =   1.5314381740
k = 13, x =   0.9521632379
k = 14, x =   1.5365070839
k = 15, x =   0.9424337713
k = 16, x =   1.5414974395
k = 17, x =   0.9326934087
k = 18, x =   1.5463985775
k = 19, x =   0.9229658020
k = 20, x =   1.5512002794
k = 21, x =   0.9132752165
k = 22, x =   1.5558928786
k = 23, x =   0.9036462847
k = 24, x =   1.5604673626
k = 25, x =   0.8941037426
k = 26, x =   1.5649154652
k = 27, x =   0.8846721509
k = 28, x =   1.5692297481
k = 29, x =   0.8753756100
k = 30, x =   1.5734036689


Function 3:
k =  0, x =   1.5000000000
k =  1, x =   2.7316890703
k =  2, x =  18.8209324059
k =  3, x = 149220368.2729534507
k =  4, x =            inf
k =  5, x =            inf
k =  6, x =            inf
k =  7, x =            inf
k =  8, x =            inf
k =  9, x =            inf
k = 10, x =            inf
k = 11, x =            inf
k = 12, x =            inf
k = 13, x =            inf
k = 14, x =            inf
k = 15, x =            inf
k = 16, x =            inf
k = 17, x =            inf
k = 18, x =            inf
k = 19, x =            inf
k = 20, x =            inf
k = 21, x =            inf
k = 22, x =            inf
k = 23, x =            inf
k = 24, x =            inf
k = 25, x =            inf
k = 26, x =            inf
k = 27, x =            inf
k = 28, x =            inf
k = 29, x =            inf
k = 30, x =            inf
/opt/conda/lib/python3.6/site-packages/ipykernel_launcher.py:8: RuntimeWarning: overflow encountered in exp

/opt/conda/lib/python3.6/site-packages/ipykernel_launcher.py:7: RuntimeWarning: invalid value encountered in double_scalars
  import sys
```

Figure 6: Fix-point iteration of the 3 functions

3c. The fixed-point theorem states that if $g \in C[a, b]$ and $a < g(x) < b$ for

all $x \in [a,b]$ and there exists a constant L such that $|g'(x)| \leq L < 1$ for all $x \in [a,b]$, then g has one fixed point $r \in (a,b)$.
Using this on the 3 functions given in 3b gives the following:

$g_1(x) = ln(4 + x - x^2)$ $g_1'(x) = \frac{1-2x}{4+x-x^2}$

$g_1'(x) = 0 \Rightarrow x = \frac{1}{2}$

This means that for $x \in [\frac{1}{2}, \rightarrow)$, $g_1(x)$ is strictly decreasing. Since $g_1(\frac{1}{2}) = 1.447$, we know that somewhere in the interval [0.5,1.447] there exists an x such that $g_1(x) = x$. This verifies the convergence found in 3b.

For $g_2(x) = \sqrt{-e^x + x + 4}$ we know from function $g_1(x)$ in 3b that a solution is somewhere between 1.28 and 1.29.

$g_2'(x) = \frac{1-e^x}{2\sqrt{-e^x+x+4}}$

$|g_2'(1.28)| = 1$ and $|g_2'(1.29)| = 1.023$

This means that $g_2(x)$ does not converge since $|g_2'(r)| > 1$. This confirms the finding in 3b

$g_3(x) = e^x + x^2 - 4$ $g_3'(x) = e^x + 2x$

It is clear to see that $|g_3'(x)| > 1$ for any $x \in [1,2]$, and it is impossible for it to converge. This also confirms what was found in 3b.

4a. If r is a fixed point for $g(x) = x$ then $g(r) = r$.
We have by the definition of inverse functions that $g(g^{-1}(x)) = x = g^{-1}(g(x))$. Applying this to a fixed point for g(x) gives:

$g^{-1}(g(x)) = x \Rightarrow g^{-1}(g(r)) = r \Rightarrow g^{-1}(r) = r$

Thus r is a fixed point for $g^{-1}(x)$ as well.

4b. We begin by defining a fixed point $r \in [a,b]$ such that $r = g(r) = g^{-1}(r)$. We have from the definition of inverse functions that $g(g^{-1}(x)) = x$. By differentiating on x, we get the following:

$x' = (g(g^{-1}(x)))' \Rightarrow 1 = g'(g^{-1}(x)) \cdot (g^{-1})'(x) \Rightarrow (g^{-1})'(x) = \frac{1}{g'(g^{-1}(x))}$

Replacing x with the fixed point r gives us:

$(g^{-1})'(r) = \frac{1}{g'(r)}$

From here it is clear that if $|g'(x)| > 1$ then $|(g^{-1})'(x)| < 1$

4c. See figure 7. Since arccos(x) does not converge, we use cos(x) instead.

```
In [23]:
def g(x):
    #Inverse of arccos(x)
    return cos(x)

#We are somewhat close to g(x)=x at pi/4
x_0 = pi/4

x, k = fixpoint(g, x_0,max_iter=50)
executed in 15ms, finished 20:49:25 2019-10-29
k =  0, x =   0.7853981634
k =  1, x =   0.7071067812
k =  2, x =   0.7602445971
k =  3, x =   0.7246674809
k =  4, x =   0.7487198858
k =  5, x =   0.7325608446
k =  6, x =   0.7434642113
k =  7, x =   0.7361282565
k =  8, x =   0.7410736871
k =  9, x =   0.7377441590
k = 10, x =   0.7399877648
k = 11, x =   0.7384768087
k = 12, x =   0.7394947711
k = 13, x =   0.7388091342
k = 14, x =   0.7392710213
k = 15, x =   0.7389599040
k = 16, x =   0.7391694833
k = 17, x =   0.7390283113
k = 18, x =   0.7391234079
k = 19, x =   0.7390593504
k = 20, x =   0.7391025006
k = 21, x =   0.7390734342
k = 22, x =   0.7390930137
k = 23, x =   0.7390798248
k = 24, x =   0.7390887090
k = 25, x =   0.7390827245
k = 26, x =   0.7390867558
k = 27, x =   0.7390840403
k = 28, x =   0.7390858694
k = 29, x =   0.7390846373
k = 30, x =   0.7390854673
```

Figure 7: Fixpoint of f(x)=arccos(x) using inverse

5a. We start by defining $f_1(x,y) = x^2 + y^2 - 4$ and $f_2(x,y) = xy - 1$, such that $f(x,y) = \begin{bmatrix} f_1(x,y) \\ f_2(x,y) \end{bmatrix}$.
Then we have to determine the jacobian for f(x,y).

$$J = \begin{bmatrix} \frac{d(f_1(x,y))}{dx} & \frac{d(f_1(x,y))}{dy} \\ \frac{d(f_2(x,y))}{dx} & \frac{d(f_2(x,y))}{dy} \end{bmatrix} = \begin{bmatrix} 2x & 2y \\ y & x \end{bmatrix}$$

Now, each step in Newton's method from here is defined as $x_{k+1} = x_k + \Delta_k$, where $J(x_k)\Delta_k = -f(x_k)$. Using this to approximate a solution for the system gives us:

$$J(x_0)\Delta_0 = -f(x_0) \Rightarrow \left[\begin{array}{cc|c} 2\cdot 2 & 2\cdot 0 & -(2^2+0^2-4) \\ 0 & 2 & -(2\cdot 0 - 1) \end{array}\right] = \left[\begin{array}{cc|c} 4 & 0 & 0 \\ 0 & 2 & 1 \end{array}\right] \Rightarrow \Delta_0 = \begin{bmatrix} 0 \\ \frac{1}{2} \end{bmatrix}$$

$$x_1 = x_0 + \Delta_0 = \begin{bmatrix} 2 \\ 0 \end{bmatrix} + \begin{bmatrix} 0 \\ \frac{1}{2} \end{bmatrix} = \begin{bmatrix} 2 \\ \frac{1}{2} \end{bmatrix}$$

$$J(x_1)\Delta_1 = -f(x_1) \Rightarrow \left[\begin{array}{cc|c} 4 & 1 & -\frac{1}{4} \\ \frac{1}{2} & 2 & 0 \end{array}\right] = \left[\begin{array}{cc|c} 4 & 1 & -\frac{1}{4} \\ 0 & \frac{15}{4} & \frac{1}{16} \end{array}\right] = \left[\begin{array}{cc|c} 4 & 0 & -\frac{4}{15} \\ 0 & \frac{15}{4} & \frac{1}{16} \end{array}\right] \Rightarrow \Delta_1 = \begin{bmatrix} -\frac{1}{15} \\ \frac{1}{60} \end{bmatrix}$$

$$x_2 = x_1 + \Delta_1 = \underline{\underline{\begin{bmatrix} \frac{29}{15} \\ \frac{31}{60} \end{bmatrix} \approx \begin{bmatrix} 1.93333 \\ 0.51667 \end{bmatrix}}}$$

5b. See figure 8. One can see that the approximation found for x_2 was correctly calculated by hand.

```
In [13]: def newton_system(f, jac, x0, tol = 1.e-10, max_iter=20):
             x = x0
             print('k ={:3d}, x = '.format(0), x)
             for k in range(max_iter):
                 fx = f(x)
                 if norm(fx, inf) < tol:
                     break
                 Jx = jac(x)
                 delta = solve(Jx, -fx)
                 x = x + delta
                 print('k ={:3d}, x = '.format(k+1), x)
             return x, k + 1
executed in 7ms, finished 20:09:07 2019-10-29
```

```
In [35]: def f(x):
             return array([x[0]**2 + x[1]**2 - 4,
                           x[0] * x[1] - 1])

         def jac(x):
             J = array(
                 [[2 * x[0], 2*x[1]],
                  [x[1], x[0]]])
             return J

         x0 = array([2.0, 0.0])
         max_iter = 10
         x, nit = newton_system(f, jac, x0, tol=1.e-12, max_iter=max_iter)

         print(f'\nApply approximated x: f(x)={f(x)}')
         if nit == max_iter:
             print('Not convergent')
executed in 21ms, finished 20:57:26 2019-10-29
k =  0, x =  [ 2.  0.]
k =  1, x =  [ 2.   0.5]
k =  2, x =  [ 1.93333333  0.51666667]
k =  3, x =  [ 1.93185274  0.51763705]
k =  4, x =  [ 1.93185165  0.51763809]
k =  5, x =  [ 1.93185165  0.51763809]

Apply approximated x: f(x)=[ -4.44089210e-16   0.00000000e+00]
```

Figure 8: Newton's method applied to system of equations from 5a

5c. See figure 9. The method uses some extra steps to approximate a solution, except from that there is no change in behaviour. It is worth noticing that starting with $x < 0$ will give a solution $x, y < 0$. This is from the fact that the system of equations has 2 possible solution, one positive and one negative.

In [34]:
```python
def f(x):
    return array([x[0]**2 + x[1]**2 - 2,
                  x[0] * x[1] - 1])

def jac(x):
    J = array(
        [[2 * x[0], 2 * x[1]],
         [x[1], x[0]]])
    return J

x0 = array([2.0, 0.0])
x, nit = newton_system(f, jac, x0, tol=1.e-12, max_iter=40)

print(f'\nApply approximated x: f(x)={f(x)}')
if nit == max_iter:
    print('Not convergent')
```

executed in 91ms, finished 20:56:20 2019-10-29

```
k =  0, x =  [ 2.  0.]
k =  1, x =  [ 1.5  0.5]
k =  2, x =  [ 1.25  0.75]
k =  3, x =  [ 1.125  0.875]
k =  4, x =  [ 1.0625  0.9375]
k =  5, x =  [ 1.03125  0.96875]
k =  6, x =  [ 1.015625  0.984375]
k =  7, x =  [ 1.0078125  0.9921875]
k =  8, x =  [ 1.00390625  0.99609375]
k =  9, x =  [ 1.00195312  0.99804688]
k = 10, x =  [ 1.00097656  0.99902344]
k = 11, x =  [ 1.00048828  0.99951172]
k = 12, x =  [ 1.00024414  0.99975586]
k = 13, x =  [ 1.00012207  0.99987793]
k = 14, x =  [ 1.00006104  0.99993896]
k = 15, x =  [ 1.00003052  0.99996948]
k = 16, x =  [ 1.00001526  0.99998474]
k = 17, x =  [ 1.00000763  0.99999237]
k = 18, x =  [ 1.00000381  0.99999619]
k = 19, x =  [ 1.00000191  0.99999809]
k = 20, x =  [ 1.00000095  0.99999905]
k = 21, x =  [ 1.00000048  0.99999952]

Apply approximated x: f(x)=[  4.54747351e-13  -2.27373675e-13]
```

Figure 9: Newton's method applied to modified system of equations from 5a