Project 1

MAD4401 - Numerical Analysis

Devere Anthony Weaver

October 7, 2022

Problem 1

Consider the problem of finding the roots of the following function,

$$f(x) = \frac{1}{100}(4x^4 - 44x^3 + 61x^2 + 270x - 525).$$

Graph the function on the interval $[-3, 9]$. Include the graph and the code used to generate the graph in your write-up and answer the following questions based on the graph.

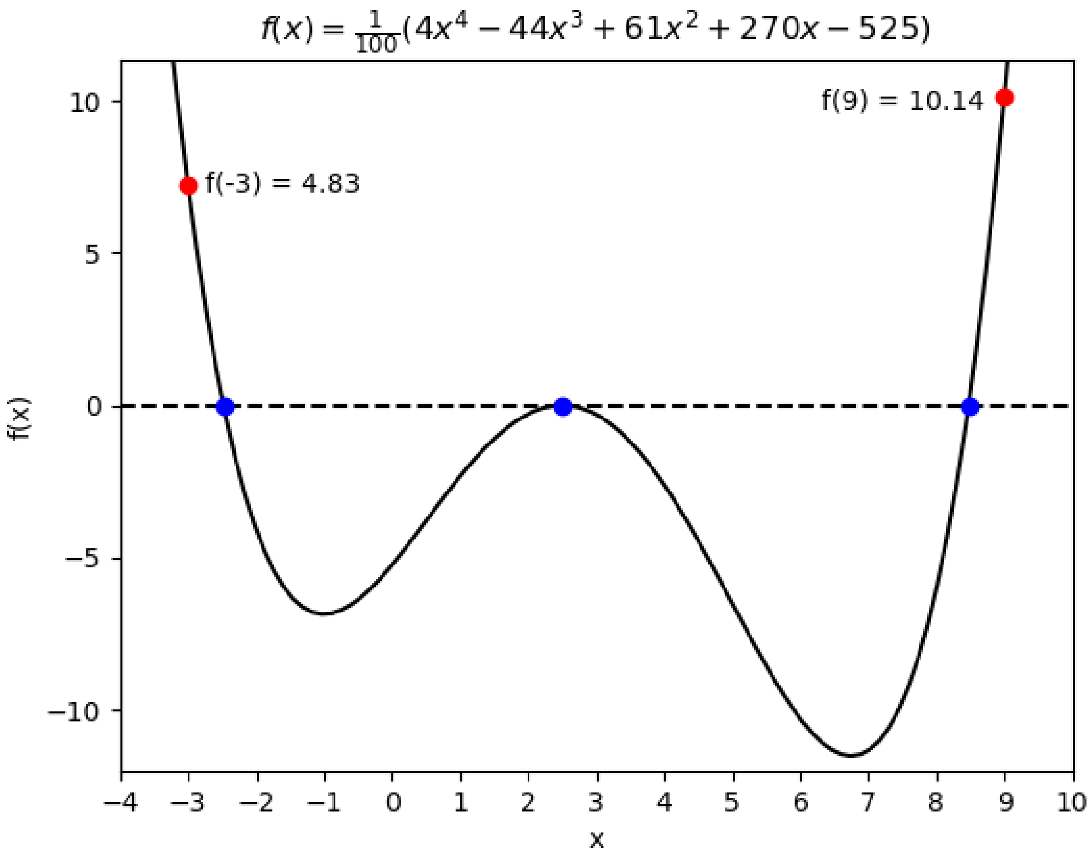
[1]:
```
import matplotlib.pyplot as plt
import numpy as np
import math
import sympy as sp
```

[2]:
```
# implementation of function f
def f(x):
    return (1/100)*(4*math.pow(x,4) - (44*math.pow(x,3)) + (61*math.pow(x,2))
        + (270*x) - 525)
```

[3]:
```
# implementation of f'
def fp(x):
    return (1/100)*(16*math.pow(x,3))-(132*math.pow(x,2)) + (122*x) + 270
```

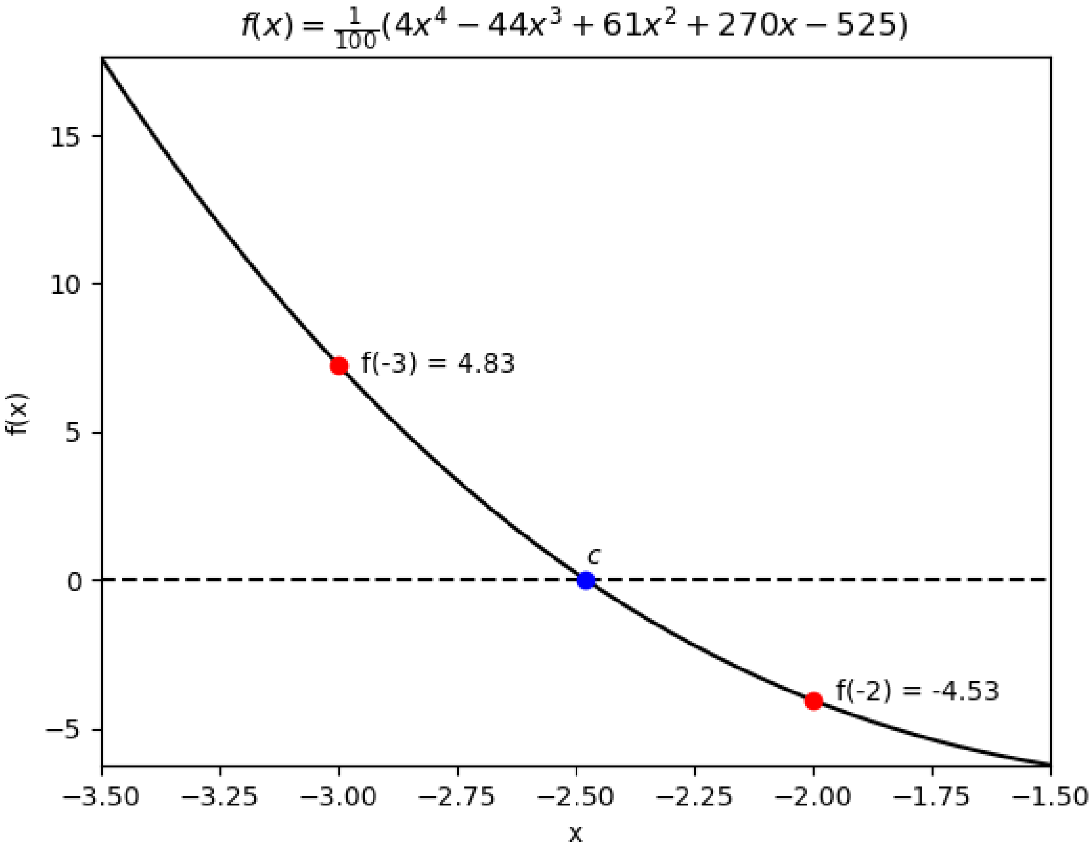
[4]:
```
# graph f(x) on [-3,9]
x = np.linspace(-4, 10, 100)    # generate 1000 evenly spaced points on (-10,10)
vf = np.vectorize(f)    # create vectorized implementation of this function

# plot graphic
plt.plot(x, vf(x), 'k')
plt.plot([-4,10], [0,0], 'k--')
plt.axis([-4, 10, -12 , f(9.05)]);



# other plot asthetics
plt.title(r'$f(x) = \frac{1}{100}(4x^4-44x^3+61x^2+270x-525)$')
```

```
plt.xlabel('x')
plt.ylabel('f(x)');

# plot points of interest
plt.plot(-3, f(-3), 'ro')
plt.text(-2.75, 7, r'f(-3) = 4.83')
plt.plot(9, f(9),'ro')
plt.text(6.3, 9.75, r'f(9) = 10.14')
plt.plot(-2.477, f(-2.477), 'bo')
plt.plot(2.5, f(2.5), 'bo')
plt.plot(8.477, f(8.477), 'bo')

# adjust ticks
plt.xticks(np.arange(-4,11,1));
```



(a) From the graph of f, we see that f has a root on the interval $[-3,-2]$. Will the bisection method starting with the interval $[-3,-2]$ converge to this root? Why or why not?

[5]:
```
# narrow graph to specified interval of [-3,-2]
x = np.linspace(-3.5, -1.5, 1000)    # generate 1000 points
vf = np.vectorize(f)    # create vectorized implementation of this function

plt.plot(x, vf(x), 'k')    # plot vf(x) vs x
plt.plot([-3.5,-1.5], [0,0], 'k--')    # plot horizontal line across
plt.axis([-3.5,-1.5, f(-1.5), f(-3.5)]);   # adjust axes to fit only interval

# other plot asthetics
plt.title(r'$f(x) = \frac{1}{100}(4x^4-44x^3+61x^2+270x-525)$')
plt.xlabel('x')
plt.ylabel('f(x)');

# points of interest
plt.plot(-3, f(-3), 'ro')
plt.text(-2.95, 7, r'f(-3) = 4.83')
plt.plot(-2, f(-2), 'ro')
plt.text(-1.95, -4, r'f(-2) = -4.53');
plt.plot(-2.477, f(-2.477), 'bo');
plt.text(-2.477, 0.5, r'$c$');
```

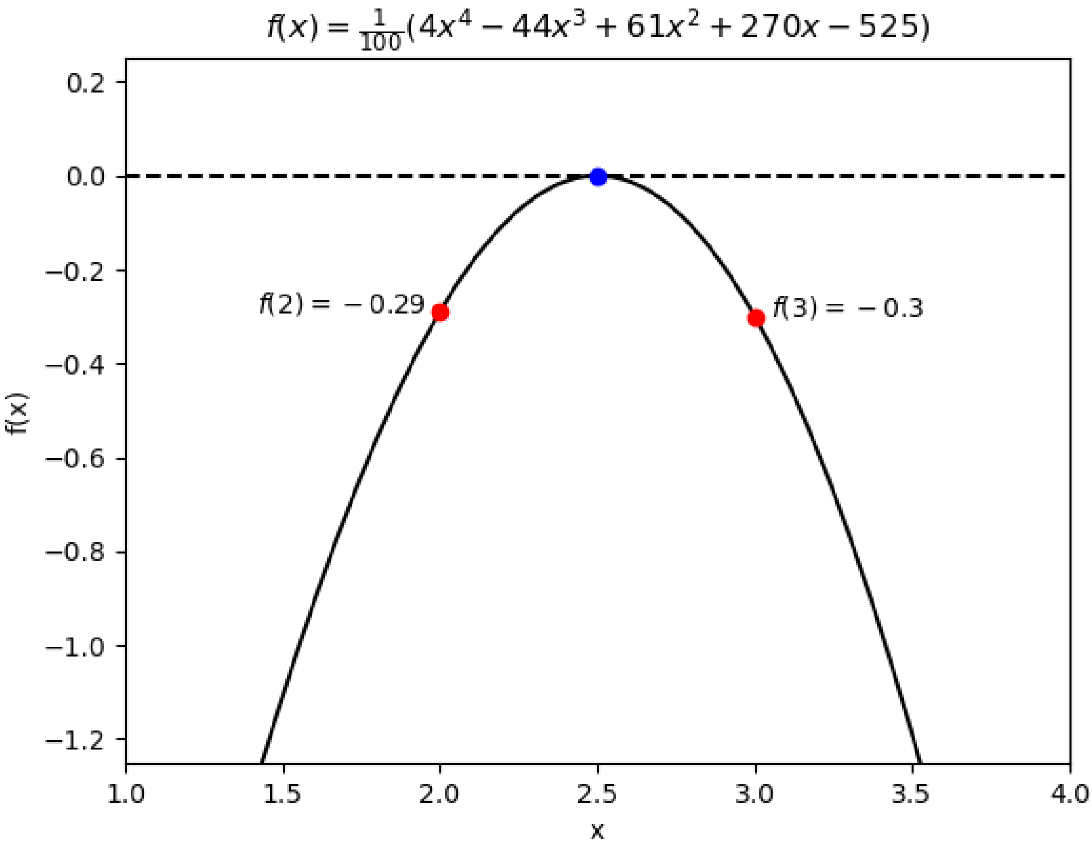


Based on the graph above, the bisection method will converge to the root on $[-3, -2]$. Observe $f \in C[-3, -2]$ and $f(-3)f(-2) < 0$. Thus by the Intermediate Value Theorem, $\exists c \in [-3, -2] \ni f(c) = 0$. Also, by inspection there only exists one root in this interval. Therefore, the bisection method will converge to this root, c.

(b) From the graph of f we see that f has a root on the interval $[2, 3]$. Will the bisection method starting with the interval $[2, 3]$ converge to this root? Why or why not?

[6]:
```
# narrow graph to specified interval of [-2,-3]

x = np.linspace(1, 4, 1000)     # generate 1000 evenly spaced points on (-10,10)
vf = np.vectorize(f)     # create vectorized implementation of this function

plt.plot(x, vf(x), 'k')     # plot vf(x) vs x
plt.plot([1, 4], [0,0], 'k--')     # plot horizontal line across
plt.axis([1, 4, -1.25, 0.25]);    # adjust axes to fit only interval

# other plot asthetics
plt.title(r'$f(x) = \frac{1}{100}(4x^4-44x^3+61x^2+270x-525)$')
plt.xlabel('x')
```

```python
plt.ylabel('f(x)');

# points of interest
plt.plot(2, f(2), 'ro')
plt.text(1.42, f(2), r'$f(2) = -0.29$')
plt.plot(3, f(3), 'ro')
plt.text(3.05, f(3), r'$f(3) = -0.3$')
plt.plot(2.5, f(2.5), 'bo');
```



While there is a unique root in this interval, the Intermediate Value Theorem doesn't hold. The function f does not change sign over the given interval, therefore the bisection method will not converge.

(c) For the root on the interval $[-3, -2]$, do you expect linear or quadratic convergence of Newton's method? Why?

For the root on $[-3, -2]$, I expect quadratic convergence using Newton's Method because the root is a simple root and is the only one within the interval.

(d) For the root on the interval $[2, 3]$, do you expect linear or quadratic convergence of Newton's method? Why?

For the root on the interval $[2, 3]$, I expect the root should converge linearly. This is because the root is of multiplicity 2 (i.e. a double root), thus the Newton method will not converge quadratically.

Problem 2

Create you own code to implement the bisection method. Include the code in your write-up and perform the following tasks.

[7]:
```python
# Bisection Method
def bisection(func, a, b, *, tol = None, maxiter = None):
    '''
    Function accepts as input a function f that implements f(x), the endpoints
    a and b of an interval [a,b]. Function allows for Bisection method
    implementation using only a maximum number of iterations,
    a maximum absolute error tolerance, or the option to use
    both as potential stopping points for execution. No return value,
    function prints to standard out the number of iterations,
    a, b, current approximation, and absolute error.
    '''
    if (f(a)*f(b)>0):    # ensure IVT holds before executing function
        raise ValueError(f"Error: f(x) does not change signs on [{a},{b}] "
                         f"or more than one root exists in [{a},{b}].")
    counter = 0
    fa = func(a)
    p = a + (b-a)/2
    l = a
    u = b
    print(f"n\ta\t\t\tb\t\t\t\tpn (midpoint)\t\t\t\tAbs. Error")
    print("_"*120)
    if (tol and maxiter == None):    # executed if no max iteration entered
        while True:
            p = a + (b-a)/2
            fp = func(p)
            if (fp == 0) or ((b-a)/2 < tol):    # exit condition
                counter += 1
                print(f"{counter:>02}\t{a:<16}\t{b:<16}\t\t{p:<16}\t\t\t"
                      "{b-p:<16}")
                break
            elif (fa * fp > 0):    # same signs, shift right
                print(f"{counter+1:>02}\t{a:<16}\t{b:<16}\t\t{p:<16}\t\t\t"
                      f"{b-p:<16}")
                a = p
                fa = fp
            else:    # different signs, shift left
```

```python
                print(f"{counter+1:>02}\t{a:<16}\t{b:<16}\t\t{p:<16}\t\t\t"
                      f"{b-p:<16}")
                b = p
            counter += 1
        print(f"\nAfter {counter} iterations, the approximation for the root"
              f" in [{l},{u}] is ~ {p}\nwith error {b-p}")
    elif (maxiter and tol == None):    # tolerance only
        while counter < maxiter:
            p = a + (b-a)/2
            fp = func(p)
            if (fa * fp > 0):    # same signs, shift right

                print(f"{counter+1:>02}\t{a:<16}\t{b:<16}\t\t{p:<16}\t\t\t"
                      "{b-p:<16}")
                a = p
                fa = fp
            else:    # different signs, shift left
                print(f"{counter+1:>02}\t{a:<16}\t{b:<16}\t\t{p:<16}\t\t\t"
                      f"{b-p:<16}")
                b = p
            counter += 1
        print(f"\nAfter {counter} iterations, the approximation for the root"
              f" in [{l},{u}] is ~ {p}\nwith error {b-p}")
    else:    # max iterations and error tolerance
        while counter < maxiter:
            p = a + (b-a)/2
            fp = func(p)
            if (fp == 0) or ((b-a)/2 < tol):
                counter += 1
                print(f"{counter:>02}\t{a:<16}\t{b:<16}\t\t{p:<16}\t\t\t"
                      f"{b-p:<16}")
                break
            if (fa * fp > 0):    # same signs, shift right

                print(f"{counter+1:>02}\t{a:<16}\t{b:<16}\t\t{p:<16}\t\t\t"
                      f"{b-p:<16}")
                a = p
                fa = fp
            else:    # different signs, shift left
                print(f"{counter+1:>02}\t{a:<16}\t{b:<16}\t\t{p:<16}\t\t\t"
                      f"{b-p:<16}")
                b = p
            counter += 1
        print(f"\nAfter {counter} iterations, the approximation for the root in"
              f" [{l},{u}] is ~ {p}\nwith error {b-p}")
```

(a) Apply the bisection method starting with the interval $[-3, -2]$ to find the root with accuracy 10^{-5}.

[8]:
```python
# apply bisection method to [-3,-2] with accuracy 10^-5
try:
    bisection(f,-3,-2,tol=0.00001)

except ValueError as err:
```

(b) Record the data from the bisection method in a table of the given form.

Iteration	**a**	**b**	$\mathbf{p_n}$ **(midpoint)**	**Abs.Error**
1	-3	-2	-2.5	0.5
2	-2.5	-2	-2.25	0.25
3	-2.5	-2.25	-2.375	0.125
4	-2.5	-2.375	-2.4375	0.0625
5	-2.5	-2.4375	-2.46875	0.03125
6	-2.5	-2.46875	-2.484375	0.015625
7	-2.484375	-2.46875	-2.4765625	0.0078125
8	-2.484375	-2.4765625	-2.48046875	0.00390625
9	-2.48046875	-2.4765625	-2.478515625	0.001953125
10	-2.47851625	-2.4765625	-2.4775390625	0.0009765625
11	-2.4775390625	-2.4765625	-2.47705078125	0.00048828125
12	-2.4775390625	-2.47705078125	-2.477294921875	0.000244140625
13	-2.477294921875	-2.47705078125	-2.4771728515625	0.0001220703125
14	-2.477294921875	-2.4771728515625	-2.47723388671875	$6.103\,515\,624 \times 10^{-5}$
15	-2.47723388671875	-2.4771728515625	-2.477203369140625	$3.051\,757\,812\,5 \times 10^{-5}$
16	-2.47723388671875	-2.477203369140625	-2.4772186279296875	$1.525\,878\,906\,25 \times 10^{-5}$
17	-2.47723388671875	-2.4772186279296875	-2.4772262573242188	$7.629\,394\,531\,25 \times 10^{-6}$

Problem 3

Create your own code to implement Newton's Method. Include the code in the write-up and perform the following tasks.

[9]:
```python
# Newton's Method Implementation
def newtonMethod(func, p0, *, error = None, maxiter = None):
    '''
    Implementation of Netwon's method for approximating roots of continous
    functions. Function takes as arguments the implementation of a
    mathematical function, the initial guess for the approximation of the root,
    and optionally as keyword arguments the absolute error tolerance and/or
    the maximum number of iterations to perform. Function prints out number of
    iterations, the nth approximation of the root and the absolute error of
    the nth iteration. Function also return a vector containing the sequence
    of approximations.
    '''
    x = sp.symbols('x')
    counter = 0
    func_diff = sp.Derivative(func, x)     # compute derivative
    approxs = np.array([p0], dtype=np.longdouble)    # approximations array
    if (error and maxiter == None):     # Absolute error only as exit condition
        print("N\tPn\t\t\tAbs. Err.")
        print("_"*60)
        while True:
            counter += 1
            p = p0 - (func.subs(x,p0)/func_diff.doit_numerically(p0))
            p = float(p)     # convert back to a float
            if (np.fabs(p-p0) < error):     # absolute error < error tolerance
                print(f"\nAfter {counter} iterations and error tolerance"
                      f" {error:f}\nthe approximated root is {p0}.")
                return approxs
            else:
                err = np.fabs(p - p0)     # absolute error
                p0 = p
                approxs = np.append(approxs, p)
            print(f"{counter}\t{p}\t{err}")
    elif (maxiter and error == None):     # Fixed number of iterations
        print("N\tPn\t\t\tAbs. Err.")
        print("_"*60)
        while counter < maxiter:
            p = p0 - (func.subs(x,p0)/func_diff.doit_numerically(p0))
            p = float(p)     # convert back to a float
            err = np.fabs(p - p0)
            counter = counter + 1
            p0 = p
```

```
            approxs = np.append(approxs, p)
            print(f"{counter}\t{p}\t{err}")
        print(f"\nAfter {counter} iterations the approximated root is {p}\n"
              f"with error of {err}.")
        return approxs
    else:    # both absolute error tolerance and maximum number of iterations
        print("N\tPn\t\t\tAbs. Err.")
        print("_"*60)
        while counter < maxiter:
            p = p0 - (func.subs(x,p0)/func_diff.doit_numerically(p0))
            p = float(p)    # convert back to a float
            if (np.fabs(p-p0) < error):    # absolute error < error tolerance
                print(f"\nAfter {counter} iterations and error tolerance"
                      f" {error:f}\nthe approximated root is {p0}.")
                return approxs
            else:
                counter = counter + 1
                err = np.fabs(p - p0)
                p0 = p
                approxs = np.append(approxs, p)
                print(f"{counter}\t{p}\t{err}")
        print(f"\nAfter {counter} iterations and error tolerance {error:f}\n"
              f" the approximated root is {p0}.")
        return approxs
```

[10]:
```
# define f(x) symbolically to use CAS for derivative computation in newtonMethod
x = sp.symbols('x')
f_sym = (1/100)*(4*x*x*x*x - 44*x*x*x + 61*x*x + 270*x - 525)
```

[11]:
```
f_sym
```

[11]:

$$0.04x^4 - 0.44x^3 + 0.61x^2 + 2.7x - 5.25$$

(a) Use Newton's method with starting guess $p_0 = -2$ to find the root of $f(x)$ on the interval $[-3, -2]$ with accuracy 10^{-5}.

[12]:
```
approximations = newtonMethod(f_sym, -2, error=0.00001)
```

```
N       Pn                      Abs. Err.
____________________________________________________________
1       -2.6428571428571432     0.6428571428571432
2       -2.4893961221453367     0.15346102071180656
3       -2.4772980226580685     0.012098099487268144
4       -2.477225577639738      7.244501833048034e-05

After 4 iterations and error tolerance 0.000010
the approximated root is -2.477225577639738.
```

In order to see if the if Newton's method converges linearly or quadratically, use the vector of approximations to see if the order of convergence is converging to some asymptotic error constant, C.

[13]:
```
# implement function to check if sequence converges linearly or quadratically
def asymptoticError(approximations):
    '''
    Function takes a vector of approximations computed using the Newton method
    and outputs a table for the order/rate of convergence to determine if the
    error is converging to some asymptotic error constant. This can be use
    to determine if the given sequence generated from the Newton method
    converges linearly or quadratically.
    '''
    print("n\tpn\t\t\t(pn - pn-1)\t\t(pn-1 - pn-2)\t\tLinear\t\t\tQuadratic")
    print("_"*124)
    for idex, p in enumerate(approximations):
        if idex==0:
            print(f"{idex}\t{p}\t\t\t-\t\t\t-\t\t\t-\t\t\t-")
        elif idex == 1:
            abserr = math.fabs(approximations[idex] - approximations[idex - 1])
            print(f"{idex}\t{p:}\t{abserr}\t-\t\t\t-\t\t\t-")
        else:
            abserr = math.fabs(approximations[idex] - approximations[idex - 1])
            prevAbserr = math.fabs(approximations[idex - 1] -
                                   approximations[idex - 2])
            print(f"{idex}\t{p}\t{abserr}\t{prevAbserr}\t"
                  f"{abserr/prevAbserr}\t"
                  f"{abserr/math.pow(prevAbserr,2)}")
```

[14]:
```
# compute error and asymptotic error constant
asymptoticError(approximations)
```

(b) Record the data from Newton's Method.

The output of the previous function call is contained in the following table.

Iteration	$\mathbf{p_n}$	$\mathbf{p_n - p_{n-1}}$	$\mathbf{p_{n-1} - p_{n-2}}$	$\frac{\lvert\mathbf{p_n - p_{n-1}}\rvert}{\lvert\mathbf{p_{n-1} - p_{n-2}}\rvert}$	$\frac{\lvert\mathbf{p_n - p_{n-1}}\rvert}{\lvert\mathbf{p_{n-1} - p_{n-2}}\rvert^2}$
0	-2.0	-	-	-	-
1	-2.64285	0.64285	-	-	-
2	-2.48939	0.15346	0.64285	0.23871	0.37133
3	-2.47729	0.01209	0.15346	0.07883	0.51371
4	-2.47722	7.2445×10^{-5}	0.01209	0.00598	0.49496

In order to determine if Newton's method converges quadratically, I decided to run Newton's method using one more iteration.

[15]:
```python
approximations2 = newtonMethod(f_sym, -2, maxiter=5)
```

[16]:
```python
asymptoticError(approximations2)
```

The following table contains the results of the previous function call.

Iteration	$\mathbf{p_n}$	$\mathbf{p_n - p_{n-1}}$	$\mathbf{p_{n-1} - p_{n-2}}$	$\frac{\lvert\mathbf{p_n - p_{n-1}}\rvert}{\lvert\mathbf{p_{n-1} - p_{n-2}}\rvert}$	$\frac{\lvert\mathbf{p_n - p_{n-1}}\rvert}{\lvert\mathbf{p_{n-1} - p_{n-2}}\rvert^2}$
0	-2.0	-	-	-	-
1	-2.64285	0.64285	-	-	-
2	-2.48939	0.15346	0.64285	0.23871	0.37133
3	-2.47729	0.01209	0.15346	0.07883	0.51371
4	-2.47722	7.2445×10^{-5}	0.01209	0.00598	0.49496
5	-2.47723	2.58807×10^{-9}	7.24450×10^{-5}	3.57247×10^{-5}	0.49313

(c) Use table to decide if Newton's method starting with $p_0 = -2$ converged linearly or quadratically to the root.

From these results, we can conclude using Newton's method on the given interval will converge to a root of f quadratically since the formula for order 2 appears to be converging to ≈ 0.49.

Problem 4

Use your Newton's method created in the previous problem to perform the following tasks:

(a) Use Newton's method with starting guess $p_0 = 2$ to find the root of $f(x)$ on the interval $[2, 3]$ with accuracy 10^{-5}.

[17]:
```
# approximate root on [2,3]
approxs = newtonMethod(f_sym, 2, error=0.00001)
```

```
N       Pn                      Abs. Err.

------------------------------------------------------------
1       2.2543859649122804      0.2543859649122804
2       2.3779615859713696      0.12357562105908926
3       2.439137616668518       0.0611760306971485
4       2.469603841591536       0.03046622492301765
5       2.4848101700354115      0.015206328443875705
6       2.4924070843990016      0.007596914363590113
7       2.4962040342205762      0.0037969498215746356
8       2.4981021391400557      0.001898104919479504
9       2.499051099955795       0.0009489608157391416
10      2.499525557558671       0.0004744576028761216
11      2.499762780673199       0.00023722311452800682
12      2.4998813908102884      0.00011861013708935886
13      2.4999406955217203      5.930471143189564e-05
14      2.4999703477904034      2.965226868312243e-05
15      2.499985173889031       1.482609862746287e-05

After 15 iterations and error tolerance 0.000010
the approximated root is 2.499985173889031.
```

[18]:
```
# linear or quadratic convergence?
asymptoticError(approxs)
```

(b) Record the data from Newton's method in a table.

Iteration	$\mathbf{p_n}$	$\mathbf{p_n - p_{n-1}}$	$\frac{\lvert\mathbf{p_n - p_{n-1}}\rvert}{\lvert\mathbf{p_{n-1} - p_{n-2}}\rvert}$	$\frac{\lvert\mathbf{p_n - p_{n-1}}\rvert}{\lvert\mathbf{p_{n-1} - p_{n-2}}\rvert^2}$
0	2.0	-	-	-
1	2.2543859649122804	0.2543859649122804	-	-
2	2.3779615859713696	0.12357562105908926	0.4857800276115929	1.9096180395766087
3	2.439137616668518	0.0611760306971485	0.49504934851103355	4.006043783298645
4	2.469603841591536	0.03046622492301765	0.4980091806518222	8.140593218890142
5	2.4848101700354115	0.015206328443875705	0.4991208619479178	16.382760358695613
6	2.4924070843990016	0.007596914363590113	0.4995889962280634	32.85401851419769
7	2.4962040342205762	0.0037969498215746356	0.49980158257046503	65.79007721422718
8	2.4981021391400557	0.001898104919479504	0.4999025556498768	131.65898395848745
9	2.499051099955795	0.0009489608157391416	0.49995171815863815	263.39519645506965
10	2.499525557558671	0.0004744576028761216	0.4999759684561565	526.8668212256239
11	2.499762780673199	0.00023722311452800682	0.499988013870956	1053.8096783360013
12	2.4998813908102884	0.00011861013708935886	0.4999940133378344	2107.6951726759517
13	2.4999406955217203	$5.930\,471\,143\,189\,564 \times 10^{-5}$	0.4999969891883396	4215.465907535799
14	2.4999703477904034	$2.965\,226\,868\,312\,243 \times 10^{-5}$	0.49999853244669284	8431.008605798232
15	2.499985173889031	$1.482\,609\,862\,746\,287 \times 10^{-5}$	0.49999879556944776	16862.07557717291

(c) Use the results from part (b) to decide if Newton's method starting with $p_0 = 2$ converged linearly or quadratically to the root on $[2, 3]$

Based on the above results, it appears as though the sequence converged linearly to the root.