Siddhanth Kumar Section 102

Workshop 9: Approximate Root-Finding Methods

Submit this notebook to bCourses to receive a grade for this Workshop.

Please complete workshop activities in code cells in this iPython notebook. The activities titled **Practice** are purely for you to explore Python, and no particular output is expected. Some of them have some code written, and you should try to modify it in different ways to understand how it works. Although no particular output is expected at submission time, it is *highly* recommended that you read and work through the practice activities before or alongside the exercises. However, the activities titled **Exercise** have specific tasks and specific outputs expected. Include comments in your code when necessary. Enter your name in the cell at the top of the notebook.

The workshop should be submitted on bCourses under the Assignments tab (both the .ipynb and .pdf files).

In lecture, you discussed a few different ways to estimate roots of nonlinear functions of one variable. Here we will expand on the details and use some of those techniques.

Root finding

Iterative Relaxation Method

A fixed point of a function g is defined to be a point x_0 where $g(x_0) = x_0$. One method of finding a fixed point is to simply guess a value, and keep applying g to it. If at some iteration we reach the fixed point, then every iteration after it will return the same value, and we know we have solved the equation. So we can try applying g for many iterations and watch for convergence. Doing so solves the equation

$$g(x) = x$$

But what we want to solve is, for a function $f(x)$,

$$f(x) = 0$$

So this method works automatically if the function $f(x)$ whose root we want to find contains a term $-x$ in it, such as

$$f(x) = 2 - e^{-x} - x$$

Then we can use the method above to solve

$$g(x) = 2 - e^{-x} = x$$

But we can also try to solve functions which don't contain $-x$ by simply adding x to both sides of the root equation:

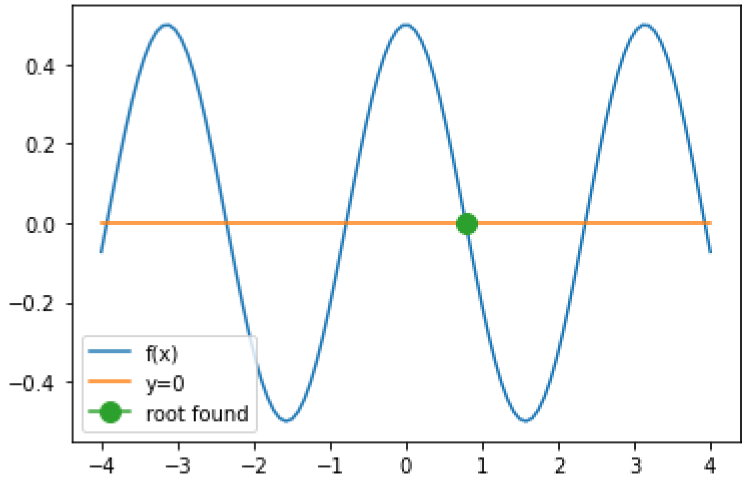
$$f(x) + x = x$$

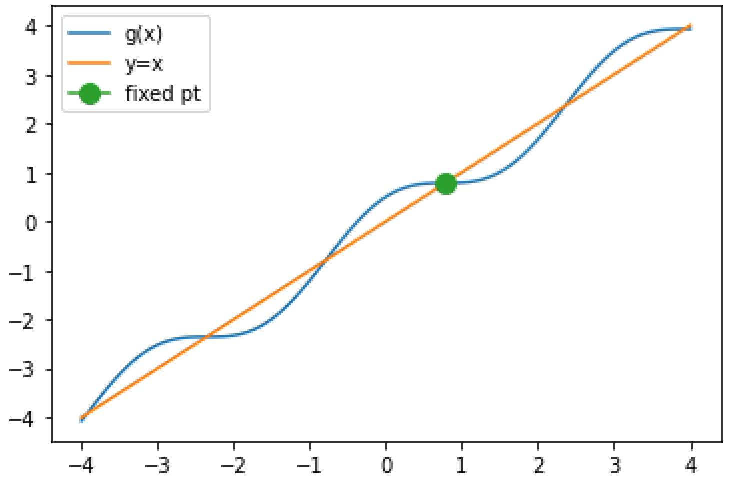
So $g(x) = f(x) + x$.

In [2]:
```python
import numpy as np
import scipy as sc
import matplotlib.pyplot as plt
%matplotlib inline

def fixedpt_finding(func, x0, nIter=50):
    '''
        fixedpt_finding uses an iterative relaxation method to solve the problem func(x)

        Inputs:
            func - name of function. func must be a python function which
                   takes only required argument and returns one value

            x0 - initial guess for the location of the fixed point
            nIter - number of iterations. Default value is 50.

        Outputs:
            x - the final estimate of the location of the fixed point
            prec - an estimate of the precision of the location, evaluated
                as the difference between the last and second last estimates
                of the location of the fixed point
            xa - numpy array of the nIter estimates of the location of the fixed point
    '''
    xa = []
    x = x0
    for i in range(nIter):
        xa.append(x)
        x = func(x)

    prec = xa[-1]-xa[-2]

    return x, prec, np.array(xa)
```

In [3]:
```python
# Example

def f(x):
    return np.cos(x)**2 - 0.5

def g(x):
    return f(x) + x

x_root, precision, _ = fixedpt_finding(g, 0.0, nIter=50)

print("Root:\t\tx= ", x_root)
print("Precision:\tdx=", precision)

xd = np.linspace(-4,4,100)

# Plot f(x) and see if the estimated root is in the right place
plt.figure()
plt.plot(xd, f(xd),label='f(x)')
plt.plot(xd, np.zeros(len(xd)),label='y=0')
plt.plot(x_root, f(x_root),marker='o',markersize=10, label='root found')
plt.legend()

# Plot g(x) and visualize the fixed point
plt.figure()
plt.plot(xd, g(xd),label='g(x)')
plt.plot(xd, xd,label='y=x')
plt.plot(x_root, g(x_root),marker='o',markersize=10, label='fixed pt')
plt.legend()

plt.show()
```

```
Root:           x=  0.7853981633974484
Precision:      dx= -1.1102230246251565e-16
```





Exercise 1

a. Use the iterative relaxation method to find the root of $f(x) = \cos(x) - x$. Print your solution and precision. Make a plot of the guessed value of the root vs. iteration number to demonstrate the convergence to the solution. Feel free to use the function defined above.

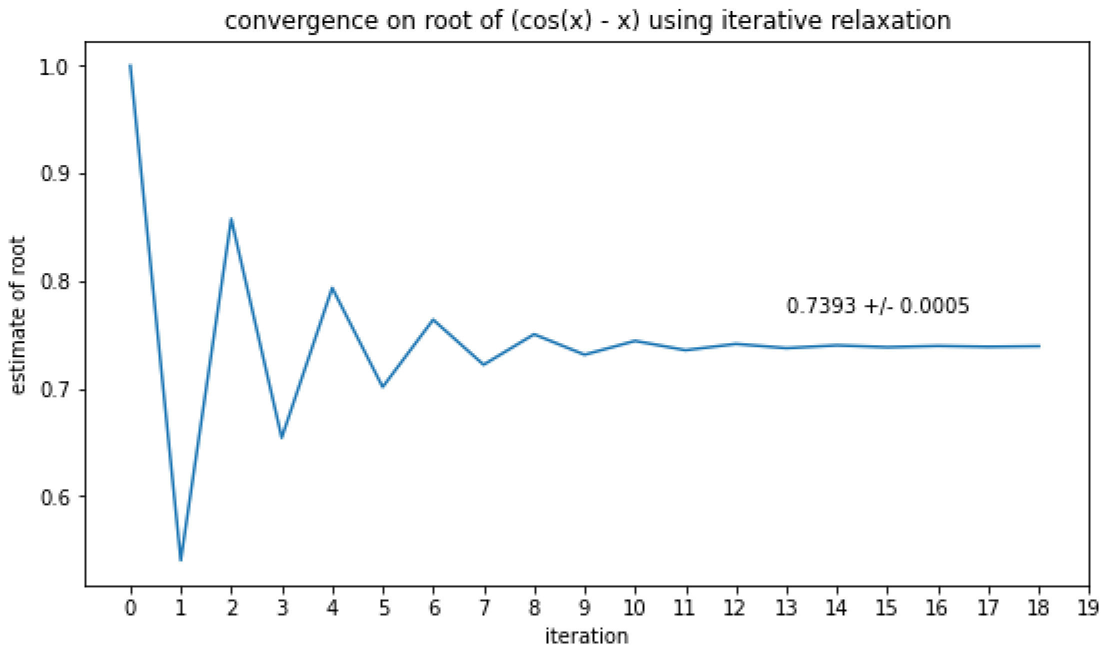
b. Now use the iterative relaxation method to find a root of $f(x) = \cos(x)$. Print your solution and precision. Plot the value vs. iteration number. Compare your answer to the analytical value.

c. Now find the root of $f(x) = e^x - x - 2$. Print your solution and precision, and plot the value vs. iteration number. Try a few different guesses and numbers of iterations. How many roots does this function have? How many were you able to find using this method?

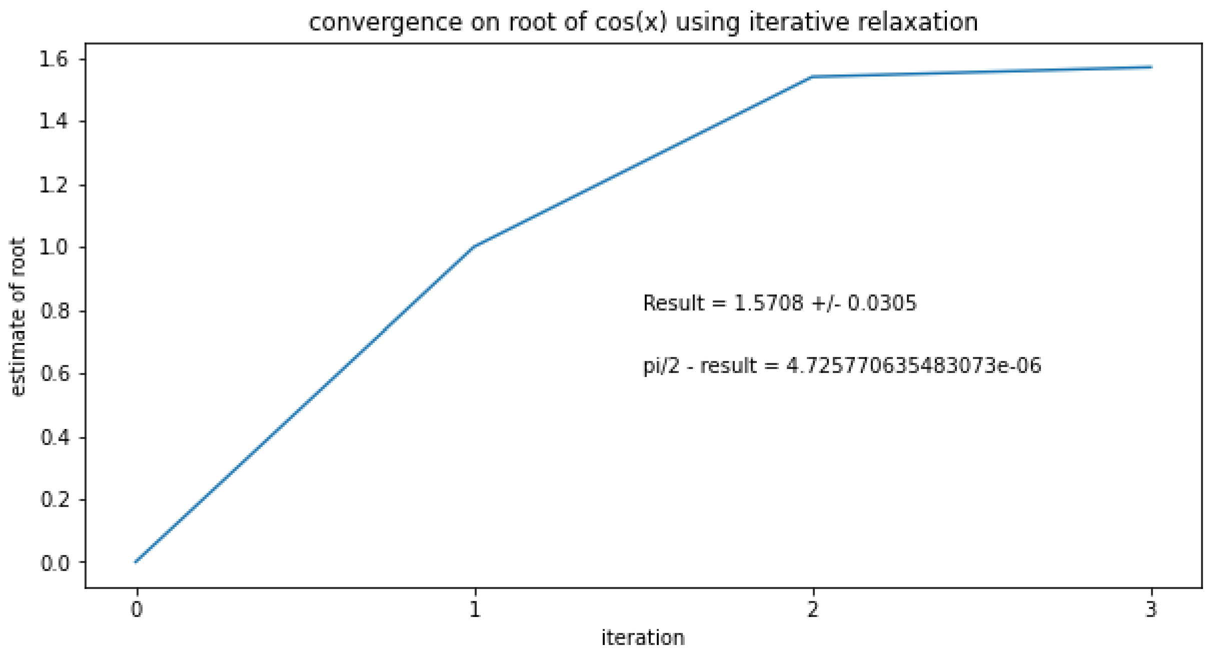
d. Lastly, use this method to find the root of $f(x) = \cos(\frac{\pi x}{2}) - x$. Print your solution and precision. Plot the value vs. iteration number. Does the method converge to a solution? Does a root exist for this function?

In [4]:
```python
# Code for Exercise 1.a

def f(x):
    return np.cos(x) - x

def g(x):
    return f(x) + x


Ns = [0, 1, 2 ,3 , 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19]
```

```
precisions = []

x_root, precision, roots = fixedpt_finding(g, 1.0, nIter=19)


plt.rcParams["figure.figsize"] = (9,5)
plt.plot(roots)
plt.text(13, 0.77, '{0:.4f} +/- {1:.4f}'.format(roots[-1], precision))
plt.title('convergence on root of (cos(x) - x) using iterative relaxation')
plt.xlabel('iteration')
plt.xticks(np.arange(min(Ns), max(Ns)+1, 1.0))
plt.ylabel('estimate of root')
plt.show()
```

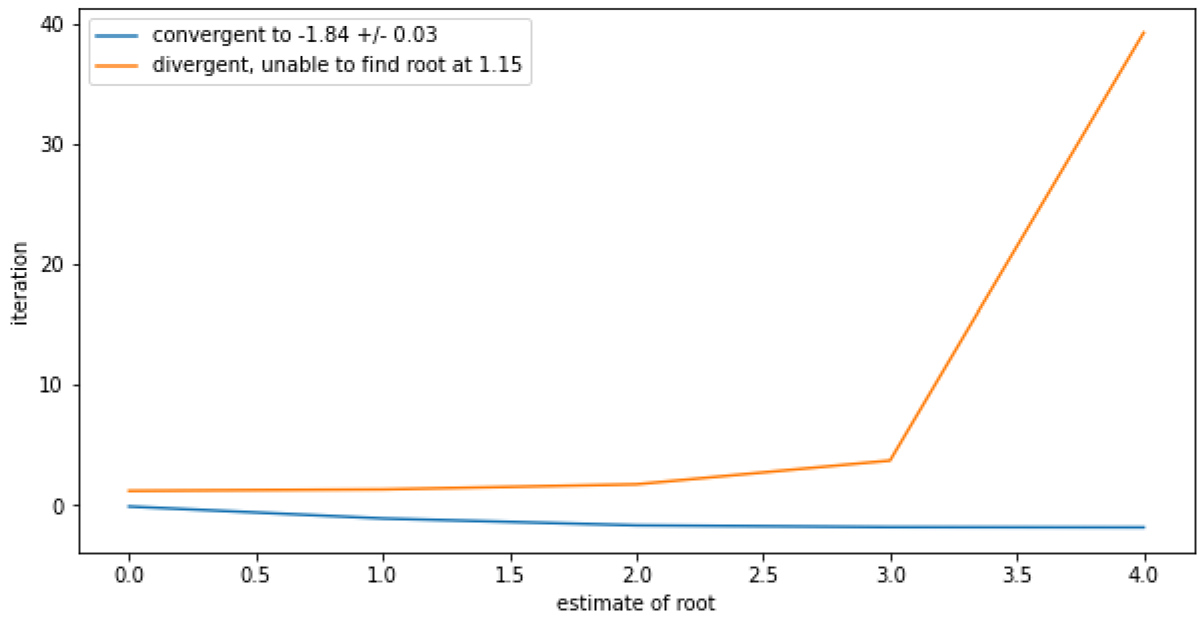


In [5]:
```
#exercise 1.b

def f(x):
    return np.cos(x)

def g(x):
    return f(x) + x


Ns = [0, 1, 2 ,3]
precisions = []


x_root, precision, roots = fixedpt_finding(g, 0.0, nIter=4)


plt.rcParams["figure.figsize"] = (10,5)
plt.plot(roots)
plt.text(1.5, 0.8, 'Result = {0:.4f} +/- {1:.4f}'.format(roots[-1], precision))
plt.text(1.5, 0.6, 'pi/2 - result = {0} '.format(np.pi/2 - roots[-1]))
plt.title('convergence on root of cos(x) using iterative relaxation')
plt.xlabel('iteration')
plt.xticks(np.arange(min(Ns), max(Ns)+1, 1.0))
plt.ylabel('estimate of root')
plt.show()
```



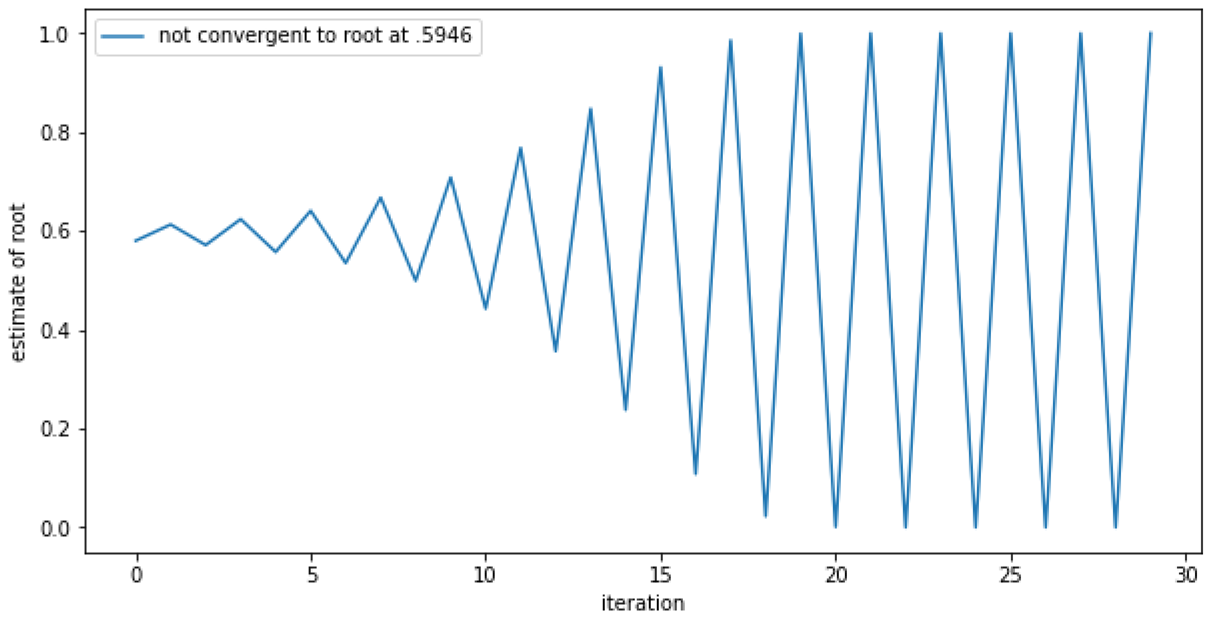
```
In [6]: #exercise 1.c

        def f(x):
            return ((np.e**x) - x - 2)

        def g(x):
            return f(x) + x


        print(np.e)

        x_root, precision, roots = fixedpt_finding(g, 1.2, nIter=5)
        x_root1, precision1, roots1 = fixedpt_finding(g, -.1, nIter=5)

        plt.plot(roots1)
        plt.plot(roots)
        plt.xlabel('estimate of root')
        plt.ylabel('iteration')

        plt.legend(['convergent to -1.84 +/- 0.03', 'divergent, unable to find root at 1.15'])
```

```
        2.718281828459045
Out[6]: <matplotlib.legend.Legend at 0x7f2278a59f10>
```



In [8]:
```
#exercise 1.d

def f(x):
    return (np.cos(np.pi * x / 2) - x)

def g(x):
    return f(x) + x

roots = [0.5946, 0.6324]
Ns = range(30)


x_root, _, roots = fixedpt_finding(g, 0.58, nIter=30)


plt.plot(roots)
plt.xlabel('iteration')
plt.ylabel('estimate of root')
plt.legend(['not convergent to root at .5946'])
```

Out[8]:
```
<matplotlib.legend.Legend at 0x7f2278acd0d0>
```



Root finding using the bisection method

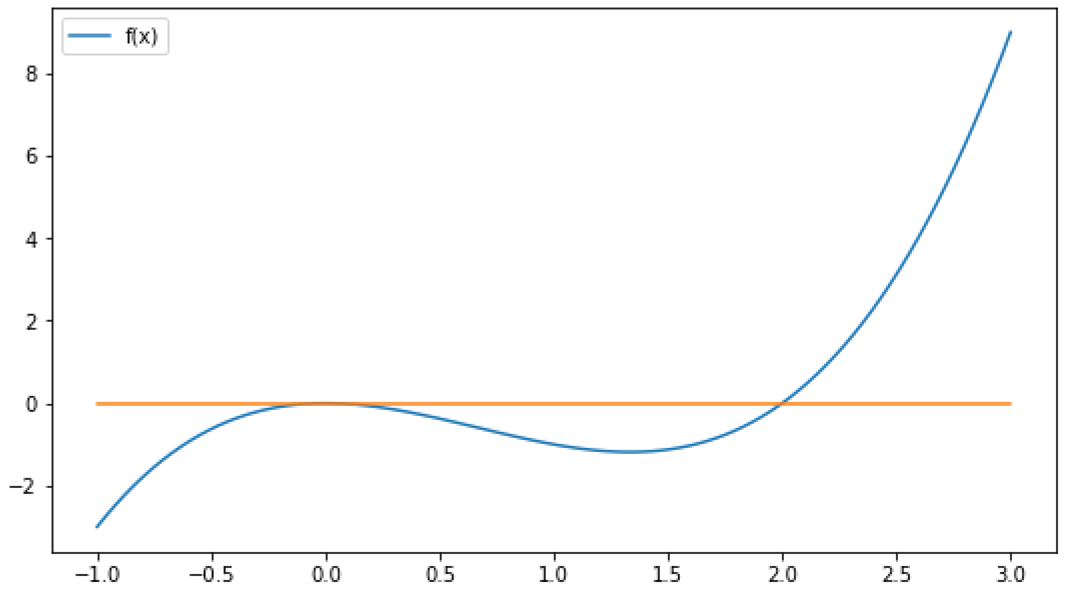
First we introduce the `bisect` algorithm which is (i) robust and (ii) slow but conceptually very simple.

Suppose we need to compute the roots of

$$f(x) = x^3 - 2x^2$$

This function has a roots at $x = 0, 2$. Run the cell below to generate a plot of $f(x)$. What do you notice about how the function behaves around each of these two zeros?

In [122…
```
def f(x):
    return x ** 3 - 2 * x ** 2

# Visualize f(x) and see the roots
xd = np.linspace(-1,3,100)
yd = f(xd)
plt.figure()
plt.plot(xd,yd, label='f(x)')
plt.plot(xd,np.zeros(len(xd)))
plt.legend()
plt.show()
```



Indeed, the `bisect` method operates on the "Intermediate Value Theorem" which just makes the observation that if a continuous function $f(x)$ changes sign over an interval $x \in [a, b]$, then there must exist at least one value of x in that interval for which $f(x) = 0$. As a result, this method cannot find the root at $x = 0$. So to use `scipy.optimize.bisect` you need to give it three arguments: the name of the python function which encodes $f(x)$, the left end of your interval (a) and the right end of your interval b. But further more, a and b must be such that $f(a)$ and $f(b)$ have opposite sign. Try changing some of these values in the call to `bisect` below!

In [116…
```
from scipy.optimize import bisect

x_root = bisect(f, 1.5, 3, xtol=1e-6) #xtol is an optional argument specifying how preci

print("The root x is approximately x=%14.12g,\n"
      "the error is less than 1e-6." % (x_root))
print("The exact error is %g." % (2 - x_root))
```

```
The root x is approximately x= 2.00000023842,
the error is less than 1e-6.
The exact error is -2.38419e-07.
```

Exercise 2

a. Use the built-in `bisect` function to compute the roots of the four functions from Exercise 2, and compare the results. Was the bisect method able to find all of the roots of $f(x) = e^x - x - 2$? What about for $f(x) = \cos(\frac{\pi x}{2}) - x$?

In [135…
```
# Code for Exercise 2

def a(x):
    return np.cos(x) - x

def b(x):
    return np.cos(x)

def c(x):
    return np.exp(x) - x - 2
```

```
def d(x):
    return (np.cos(np.pi * x / 2) - x)

x_root = bisect(a, 0, 1 , xtol=1e-6)#xtol is an optional argument specifying how precise
x_root1 = bisect(b, 1, 2 , xtol=1e-6)
x_root2 = bisect(c, -2, 0, xtol=1e-6)
x_root3 = bisect(c, 0, 2, xtol=1e-6)
x_root4 = bisect(d, 0, 1, xtol=1e-6)

print('Able to find (cos(x)-x) root at {0:.6f} +- 1e-6'.format(x_root))
print('Able to find cos(x) root at {0:.6f} +- 1e-6'.format(x_root1))
print('Able to find (e^x - x -2) root at {0:.6f} +- 1e-6'.format(x_root2))
print('Able to find (e^x - x - 2) root at {0:.6f} +- 1e-6'.format(x_root3))
print('Able to find (cos(x*pi/2)-x) root at {0:.6f} +- 1e-6'.format(x_root4))
```

```
Able to find (cos(x)-x) root at 0.739085 +- 1e-6
Able to find cos(x) root at 1.570796 +- 1e-6
Able to find (e^x - x -2) root at -1.841405 +- 1e-6
Able to find (e^x - x - 2) root at 1.146194 +- 1e-6
Able to find (cos(x*pi/2)-x) root at 0.594611 +- 1e-6
```

Root finding using Brent method

This is another method to find a root of the function $f(x)$ on the sign changing interval $x \in [a, b]$. It is a safe version of the secant method that uses inverse quadratic extrapolation. Brent's method combines a few other elementary root-finding techniques: root bracketing, interval bisection, and inverse quadratic interpolation. But again, this still requires that $f(a)$ and $f(b)$ have opposite signs.

In [136…

```
from scipy.optimize import brentq

def f(x):
    return x ** 3 - 2 * x ** 2

x = brentq(f, 0.5,3, xtol=1e-6)

print("The root x is approximately x=%14.12g,\n"
      "the error is less than 1e-6." % (x))
print("The exact error is %g." % (2 - x))
```

```
The root x is approximately x= 2.00000000005,
the error is less than 1e-6.
The exact error is -4.75548e-11.
```

Root finding using the `fsolve` function

A (often) better (in the sense of "more efficient") algorithm than the bisection algorithm is implemented in the general purpose `fsolve()` function for root finding of (multidimensional) functions. This algorithm needs only one starting point close to the suspected location of the root (but is not garanteed to converge).

Here is an example:

In [10]:

```
from scipy.optimize import fsolve

def f(x,a,b):
    return a*x**3 - b*x**2

a = 1
b = 2
x = fsolve(f, x0=[-0.5,3], args=(a,b))           # look for two roots starting with 0 an
```

```
print("Number of roots is", len(x))
print("The root(s) are ", x)
```

```
Number of roots is 2
The root(s) are  [-1.86238792e-26  2.00000000e+00]
```

The input to `fsolve` is the name of the python function and the array of initial locations for the roots you are trying to find. You can optionally pass additional arguments (parameters) to the function as a list with the keyword `args`. The return value of `fsolve` is a numpy array of the best estimates of the locations of the roots found for each initial guess given. If n initial guesses are given, n estimates are returned.

Exercise 3

Find the solutions of the quadratic equation $ax^2 + bx + c = 0$ for an arbitrary set of coefficients a, b, c using `fslove`, and compare to the exact solution.

In [11]:

```
# Code for Exercise 3

def f(x, a, b, c):
    return a*x**2 + b*x + c

a = 4
b = 6
c = -3

sol = fsolve(f, x0=[-2,1], args=(a,b,c))
print('Fsolve solutions: {0}'.format(sol))
print('Analytical solutions: {0}'.format([-1.896, 0.396]))
```

```
Fsolve solutions: [-1.89564392  0.39564392]
Analytical solutions: [-1.896, 0.396]
```

Exercise 4

(Newman 6.15)

Consider a sixth-order polynomial

$$P(x) = 924x^6 - 2772x^5 + 3150x^4 - 1680x^3 + 420x^2 - 42x + 1$$

There is no general formula for the roots of a polynomial of degree 6, but you can compute the roots numerically.

1. Make a plot of $P(x)$ from $x = 0$ to $x = 1$ and by inspecting it find rough values for the six roots of the polynomial.
2. Write the code to solve for the positions of all six roots to at least ten decimal places using at least one of the methods above (you can use the built-in functions).

In [16]:

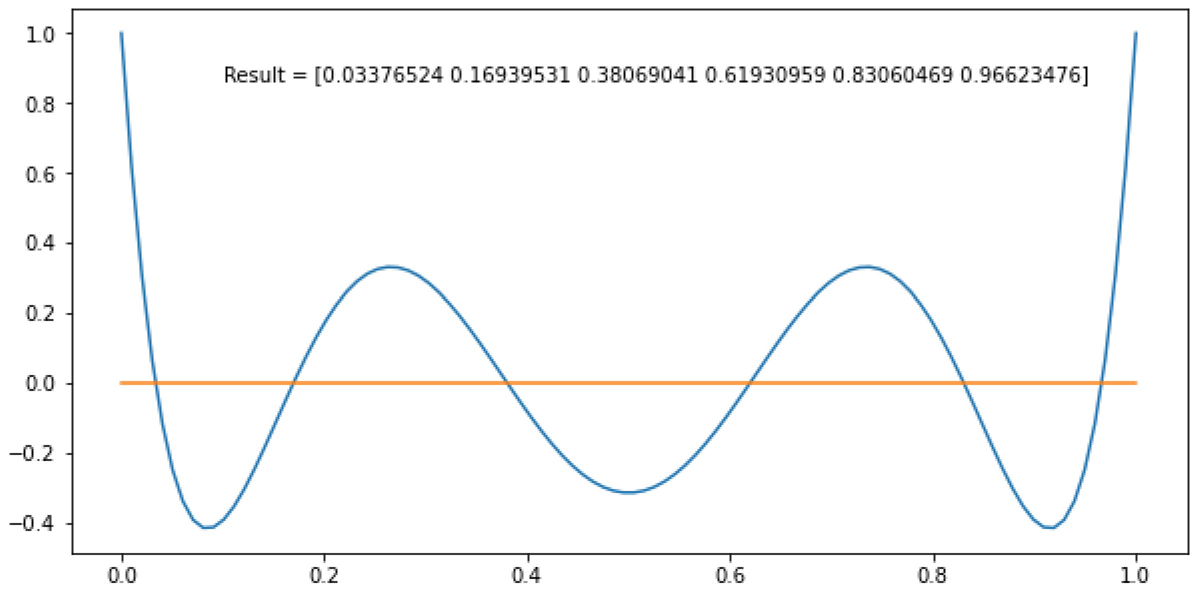
```
# Code for Exercise 4

def p(x):
    return ((924*x**6) - (2772*x**5) + (3150*x**4) - (1680*x**3) + (420*x**2) - (42*x) +

xs = np.linspace(0, 1, 100)
ys = []

for value in xs:
    ys.append(p(value))
```

```
sol = fsolve(p, x0=[0,0.2, 0.4, 0.6, 0.8, 1])

print('roots:')
for value in sol:
    print(value)

plt.plot(xs, ys)
plt.text(0.1, 0.8, 'Result = {0} \n'.format(sol))
plt.plot(xs,np.zeros(len(xs)))
```

```
roots:
0.03376524290243354
0.16939530672083822
0.3806904068103104
0.6193095931897283
0.8306046932875865
0.9662347570906943
```

Out[16]:

```
[<matplotlib.lines.Line2D at 0x7f227707ac70>]
```



In [ ]: