Chapter-05 Roots of Equations

Date	@September 20, 2024
tag	done

Bisection Method
Fixed Point Method
Newton-Raphson Method
Secant Method
Module Scipy.optimize

$f(x) = 0$

if equation is nonlinear, we cannot find analytical solution.

Question # 3 [Marks = 4+4]

- Apply appropriate theorem to test the existence of a root for each of the following functions.
  (a) $f_1(x) = \ln(x) + e^x$ in the interval $[-6, 6]$
  (b) $f_2(x) = e^x + \cos(x) - x^3$ in the interval $[-4, 4]$

Solution:

We will apply the **Intermediate Value Theorem (IVT)**. Two conditions must be satisfied: 1. The function must be continuous on the given interval. 2. There must be a sign change between the function values at the endpoints of the interval.

(a) **For $f_1(x) = \ln(x) + e^x$ in the interval** $[-6, 6]$:
Continuity: - $\ln(x)$ is only defined for $x > 0$, so $f_1(x)$ is not continuous in the interval $[-6, 6]$ because it includes negative values where the logarithm is undefined.
Conclusion: - Since the function is not continuous in this interval, IVT cannot be applied, and we cannot test for a root in the interval $[-6, 6]$.

(b) **For $f_2(x) = e^x + \cos(x) - x^3$ in the interval** $[-4, 4]$:
Continuity: - $f_2(x)$ is the sum of continuous functions (e^x, $\cos(x)$, and x^3), so it is continuous over the entire interval $[-4, 4]$.
Sign Change: - At $x = -4$:

$$f_2(-4) = e^{-4} + \cos(-4) - (-4)^3 \approx 0.0183 - 0.6536 + 64 = 63.3647 > 0$$

- At $x = 4$:

$$f_2(4) = e^4 + \cos(4) - 4^3 \approx 54.5981 - 0.6536 - 64 = -9.2045 < 0$$

Conclusion: - Since $f_2(-4) > 0$ and $f_2(4) < 0$, there is a sign change. Therefore, by the IVT, there exists at least one root in the interval $(-4, 4)$.

CLO-3 Implement a numerical method in Python/Numpy/Scipy.

▼ Bisection Method

based on Bolzano's Intermediate Value Theorem

If a function f(x) satisfies the following conditions:

- f is continuous in $[a, b]$
- $f(a) \cdot f(b) < 0$

then there exists at least one solution $x^* \in (a, b)$ of the equation $f(x) = 0$
This means that the function changes sign in the interval
$[a, b]$ and $\text{sign}(f(a)) \neq \text{sign}(f(b))$

The **Bisection method** is used to find a very small interval $[a, b]$ that contains the root of the equation $f(x) = 0$ and the approximate solution is defined to be the midpoint of that interval $(a + b)/2$

```
def bisection(f, a, b, tol = 1.e-6):
    iteration = 0 #initialize counter iteration
    if (f(a) * f(b) < 0.0): # check if there is a root
        while ((b-a) > tol): # check if the end-points converge
            iteration = iteration + 1
            x = (a + b)/2
            if (f(a) * f(x) < 0.0):
                b = x
            elif (f(x) * f(b) < 0.0):
                a = x
            else:
                break
            print(iteration, x)
    else:
        print('failure')
    return x
# returns the midpoint of the final interval
```

$\ln x + x = 0 in the inteval [0.1, 1]$

```
import numpy as np

def f(x):
    y = np.log(x) + x
    return y

a = 0.1
b = 1.0
tol = 1.e-4
x = bisection(f, a, b, tol)
print('The aproximate solution is: ', x)
print('And the error is: ', f(x))
```

Output:

1 0.55
2 0.775
3 0.6625000000000001
4 0.6062500000000001
5 0.578125
6 0.5640625
7 0.57109375
8 0.567578125
9 0.5658203125000001
10 0.5666992187500001
11 0.567138671875
12 0.5673583984375
13 0.56724853515625
14 0.567193603515625
The approximate solution is: 0.567193603515625
And the error is: 0.0001390223881425623

Evaluating the value $f(x^*)$ we observe that the approximate solution satisfies the equation $f(x) = 0$ with 4 decimal digit is correct. This is because we chose `tol = 1.e-4`. Try using smaller values of the variable `tol`.

We also observe that to achieve the requested accuracy the method required 14 iterations

Experimental Convergence Rate

In order to estimate the convergence rate, we compute the errors for three subsequent iterations, let's say e_1, e_2, e_3. Then we assume that

$|e_2| = C|e_1|^r$

$|e_3| = C|e_2|^r$

Dividing the previous equations we have

$$\frac{|e_2|}{|e_3|} = \left(\frac{|e_1|}{|e_2|}\right)^r$$

We then we solve for n to obtain the formula

$$r = \frac{\log \frac{|e_2|}{|e_3|}}{\log \frac{|e_1|}{|e_2|}}$$

We modify the bisection function in order to compute the convergence rates numerically, and we try using the same problem as before.

```
def bisection_rates(f, a, b, tol = 1.e-6):
    iteration = 0
    if (f(a) * f(b) < 0.0):
        e1 = abs(b-a) #initialize e1 arbitrarily
        e2 = e1*2 #initialize e2  arbitrarily
        e3 = e1 #initialize e3 arbitrarily
```

```
        while ((b-a)>tol):
            e1 = e2
            e2 = e3
            iteration = iteration + 1
            x = (a + b)/2
            if (f(a) * f(x) < 0.0):
                b = x
            elif (f(x) * f(b) < 0.0):
                a = x
            else:
                break
            e3 = np.abs(b-a)
            rate = np.log(e2/e3)/np.log(e1/e2)
            print('iteration = ', iteration, 'rate =', rate)
    else:
         print('failure')

    return x
```

```
def f(x):
    y = np.log(x) + x
    return y

a = 0.1
b = 1.0
tol = 1.e-4

x = bisection_rates(f, a, b, tol)
print('The approximate solution x is: ', x)
print('And the value f(x) is: ', f(x))
```

iteration = 1 rate = 1.0000000000000002
iteration = 2 rate = 0.9999999999999997
iteration = 3 rate = 0.9999999999999993
iteration = 4 rate = 1.0000000000000007
iteration = 5 rate = 1.0000000000000029
iteration = 6 rate = 0.9999999999999971
iteration = 7 rate = 1.0000000000000115
iteration = 8 rate = 0.9999999999999657
iteration = 9 rate = 1.0000000000000682
iteration = 10 rate = 0.9999999999999545
iteration = 11 rate = 0.9999999999998177
iteration = 12 rate = 1.0000000000001823
iteration = 13 rate = 1.0000000000007292
iteration = 14 rate = 0.9999999999992709
The approximate solution x is: 0.567193603515625
And the value f(x) is: 0.0001390223881425623

Verify theoretically that for the previous example we need 14 iterations.

Manual Method:

	a	b	c	f(a)	f(b)	f(C)
1	0.1	1	0.55	−2.202585093	1	−0.047837001
2	0.55	1	0.775	−0.047837001	1	0.52010775
3	0.55	0.775	0.6625	−0.047837001	0.52010775	0.250765279
4	0.55	0.6625	0.60625	−0.047837001	0.250765279	0.105787163
5	0.55	0.60625	0.578125	−0.047837001	0.105787163	0.030159829
6	0.55	0.578125	0.5640625	−0.047837001	0.030159829	−0.008527718
7	0.5640625	0.578125	0.57109375	−0.008527718	0.030159829	0.010891853
8	0.5640625	0.57109375	0.567578125	−0.008527718	0.010891853	0.001201251
9	0.5640625	0.567578125	0.565820313	−0.008527718	0.001201251	−0.003658406
10	0.565820313	0.567578125	0.566699219	−0.003658406	0.001201251	−0.001227375
11	0.566699219	0.567578125	0.567138672	−0.001227375	0.001201251	−0.000012762
12	0.567138672	0.567578125	0.567358399	−0.000012762	0.001201251	0.000594321
13	0.567138672	0.567358399	0.567248536	−0.000012762	0.000594321	0.0002908
14	0.567138672	0.567248536	0.567193604	−0.000012762	0.0002908	0.000139024

So the approximate solution value is c and the error is f(c)

> ✨ how to decide how many iterations? when f(c) < tolerance

▼ Fixed Point Method

Given an interval, bisection is guaranteed to converge to a root
However bisection uses almost no information about
`f(x)` beyond its sign at a point

Basic Idea: Every equation of the form `f(x)=0` can be written equivalently in the form `x=g(x)` in many different ways

Date________ 20___ M T W T F S

$f(x) = \cos(x) + x^3 - 0.5$

$= -\sin x + 3x^2$

$f(x) = 0$

$\cos(x) + x^3 - 0.5 = 0$

$f(-5) = -125.216$

$f(5) = 124.78$

$x^3 = 0.5 - \cos x$

$f(x) = x^3 - x^2 - 3x - 3$ $[1, 3]$ a b

$f(x)=0$ has 3 forms

$x^3 = x^2 + 3x + 3$ $x^2 = x^3 - 3x - 3$ $x = \frac{x^3 - x^2 - 3}{3}$

$x = \sqrt[3]{x^2 + 3x + 3}$ $x = \sqrt{x^3 - 3x - 3}$

Find derivatives

$g_1'(x) = \frac{3}{2}(2x-3)(x^2+3x+3)^{-2/3}$ g

$g_2'(x) = \frac{1}{2}(3x^2-3)(x^3-3x-3)^{-1/2}$

$g_3'(x) = \frac{1}{3}(3x^2 - 2x)$

$g_1'(1.5) = \cancel{2.136}$ 0 ✓ selected

$g_2'(1.5) = \cancel{2.408}$ Math error

$g_3'(1.5) = 1.25$

Page #

```
def fixedpoint(g, x0, tol = 1.e-6, maxit = 100):
    # g = the function g(x)
    # x0 = the initial guess of the fixed point x=g(x)
    # tol = tolerance for the absolute error
    #         of two subsequent approximations
    # maxit = maximum number of iterations allowed
    error = 1.0
    iteration = 0
    xk = x0
    while (error > tol and iteration < maxit):
        iteration = iteration + 1
        error = xk
        xk = g(xk)
        error = np.abs(error - xk)
        print ('iteration =', iteration, ', x =', xk)
    return xk
```

```
def f(x):
    y = x**2-x-1.0
    return y
def g(x):
    y = np.sqrt(x+1.0)
    return y
tol = 1.e-4
maxit = 50
x0 = 0.0
x = fixedpoint(g, x0, tol, maxit)
print('The approximate solution x is: ', x)
print('And the value f(x) is: ', f(x))

#Output
iteration = 1 , x = 1.0
iteration = 2 , x = 1.4142135623730951
iteration = 3 , x = 1.5537739740300374
iteration = 4 , x = 1.5980531824786175
iteration = 5 , x = 1.6118477541252516
iteration = 6 , x = 1.616121206508117
iteration = 7 , x = 1.6174427985273905
iteration = 8 , x = 1.617851290609675
iteration = 9 , x = 1.6179775309347393
iteration = 10 , x = 1.6180165422314876
The approximate solution x is:  1.6180165422314876
And the value f(x) is:  -3.9011296748103774e-05
```

Newton-Raphson Method

Basic Idea: Given $f(x)$ and $f'(x)$ and and initial guess x_0, find the root of the tangent line to $(x_0, f(x_0))$ to find $x_1 \approx x^*$. Continue using x_1 to compute x_2, etc.

Given
x_k, then the next approximation of the root x^* defined by Newton's method is:

$$x_{k+1} = x_k - \frac{f(x_k)}{f'(x_k)}$$

Date______ 20___
M T W T F S

$f(x) = \ln x + x$

$f'(x) = \frac{1}{x} + 1$

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

let $x_0 = 1$ $\quad$ $err = tol + 1 = 1\times e^{-4} + 1 = 1.0001$

1) $x_1 = x_0 - \frac{f(x_0)}{f'(x_0)}$ $\quad$ $err = 1$

$f(x_0) = f(1) = \ln 1 + 1 = 0 + 1 = 1$

$f'(x_0) = f'(1) = 2$

$x_1 = 1 - \frac{1}{2} = \frac{1}{2}$ $\quad$ $err = |err - x_k| = |1 - \frac{1}{2}| = 0.5$

2) $x_2 = x_1 - \frac{f(x_1)}{f'(x_1)}$ $\quad$ $err = x_k = 0.5$

$f(x_1) = f(1/2) = \ln(0.5) + 1/2 = -0.193147$

$f'(x_1) = f'(1/2) = \frac{1}{1/2} + 1 = 3$

$x_2 = \frac{1}{2} - \frac{(-0.193147)}{3} = 0.564382$ $\quad$ $err = 0.5 - 0.564382$

$= -0.064382$

3) $x_3 = x_2 - \frac{f(x_2)}{f'(x_2)}$ $\quad$ $err = x_k = 0.564382$

$f(x_2) = f(0.564382) = \ln(0.564382) + 0.564382$

$= -7.641952\times 10^{-3}$

$f'(x_2) = 1/0.564382 + 1 = 2.771849$

$x_3 = 0.564382 - \frac{\cancel{0.564382}\,(-7.641952\times 10^{-3})}{2.771849}$

$= 0.5671389871$

$err = 0.564382 - 0.5671389871$

$= -2.7569871\times 10^{-3}$

TICK® INDUSTRIES PRIVATE LIMITED $\quad$ Page #

Date______20___
M T W T F S

4) $x_4 = x_3 - \frac{f(x_3)}{f'(x_3)}$ $err = xk = 0.5671389871$

$f(x_3) = -1.189103265 \times 10^{-5}$

$f'(x_3) = 2.763236213$

$x_4 = 0.5671432904$

$err = 0.5671389871 - 0.5671432904$
$= -4.3033 \times 10^{-6}$

Now err has become < tol so we break the loop

Total Error: $\ln(0.5671432904) + 0.5671432904$
$= -2.8778 \times 10^{-11}$

```
def newton(f, df, x0, tol = 1.e-6, maxit = 100):
    # f = the function f(x)
    # df = the derivative of f(x)
    # x0 = the initial guess of the solution
    # tol = tolerance for the absolute error
    # maxit = maximum number of iterations
    err = tol+1.0
    iteration = 0
    xk = x0
    while (err > tol and iteration < maxit):
        iteration = iteration + 1
        err = xk # store previous approximation to err
        xk = xk - f(xk)/df(xk) # Newton's iteration
        err = np.abs(err - xk) # compute the new error
        print(iteration, xk)
    return xk



def f(x):
    y = np.log(x) + x
    return y
def df(x):
    y = 1.0 / x + 1.0
    return y
tol = 1.e-4
maxit = 50
x0 = 1.0
```

```
x = newton(f, df, x0, tol, maxit)
print('The aproximate solution is: ', x)
print('And the error is: ', f(x))
#Output
1 0.5
2 0.5643823935199818
3 0.5671389877150601
4 0.5671432903993691
The aproximate solution is:  0.5671432903993691
And the error is:  -2.877842408821607e-11
```

Secant Method

The secant method can be obtained from Newton's method with the approximation of the first derivative

$f'(x_k) = \frac{f(x_k)-f(x_{k-1})}{x_k-x_{k-1}}$

The new iteration is the defined

Given x_k and x_{k-1}

$x_{k+1} = x_k - \frac{x_k-x_{k-1}}{f(x_k)-f(x_{k-1})} f(x_k)$

```
def secant(f, x1, x2, tol = 1.e-6, maxit = 100):
    # f = the function f(x)
    # x1 = an initial guess of the solution
    # x2 = another initial guess of the solution
    # tol = tolerance for the absolute error
    # maxit = maximum number of iterations
    err = 1.0
    iteration = 0
    while (err > tol and iteration < maxit):
        xk1 = x1
        xk = x2
        iteration = iteration + 1
        err = xk1
        xk1 = xk - (xk-xk1)/(f(xk)-f(xk1))*f(xk)
        err = np.abs(err - xk1)
        x1 = x2
        x2 = xk1
        print(iteration, xk1)
    return xk1

def f(x):
    y = np.log(x) + x
    return y
tol = 1.e-4
maxit = 50
x1 = 1.0
x2 = 2.0
```

```
x = secant(f, x1, x2, tol, maxit)
print('The approximate solution is: ', x)
print('And the error is: ', f(x))

#Output
1 0.40938389085035864
2 0.651575386390747
3 0.5751035382227284
4 0.5667851889083253
5 0.5671448866112347
6 0.5671432907314143
7 0.5671432904097836
The approximate solution is:  0.5671432904097836
And the error is:  -6.661338147750939e-16
```

$f(x) = \ln x + x \qquad tol = 1 \times 10^{-4}$

$x_1 = 1$

$x_2 = 2$

$n = 2$

$$\Rightarrow x_3 = x_2 - \left(\frac{x_2 - x_1}{f(x_2) - f(x_1)} \right) f(x_2)$$

$f(x_2) = f(2) = 2.693147181$

$f(x_1) = f(1) = 1$

$$x_3 = 2 - \frac{(2-1)}{1.693147181} (2.693147181)$$

$x_3 = 0.409383891$

$$\Rightarrow x_4 = x_3 - \frac{(x_3 - x_2)}{f(x_3) - f(x_2)} f(x_3)$$

$f(x_3) = f(0.409383891) = -0.4837180634$

$\cancel{f(x}$

$x_4 = x_3 - (-0.2421914953)$

$x_4 = 0.6515753863$

$\vdots$

$\vdots$ until error > tol

$\Rightarrow x_9 = 0.5671432904097836.$

error $= \ln(x_9) + x_9$;

▼ Module Scipy.optimize

The bisection method is implemented in the function `bisect` of the module `scipy.optimize`

```
import scipy.optimize as spo
def f(x):
    y = np.log(x)+x
    return y
a = 0.1
b = 1.0
tol = 1.e-4
x = spo.bisect(f, a, b, () , tol)
print('The approximate solution x is: ', x)
```

```
print('And the value f(x) is: ', f(x))
#Output
The approximate solution x is:  0.567193603515625
And the value f(x) is:  0.0001390223881425623
```

The generic fixed-point method is also implemented in `scipy.optimize` in the function `fixed_point`

```
import scipy.optimize as spo
def f(x):
    y = x**2-x-1.0
    return y
def g(x):
    y = np.sqrt(x+1.0)
    return y
x0 = 1.0
tol = 1.e-4
maxit = 50
x = spo.fixed_point(g, x0, (), tol, maxit)
print('The approximate solution x is: ', x)
print('And the value f(x) is: ', f(x))

#Output
The approximate solution x is:  1.6180339887498991
And the value f(x) is:  9.547918011776346e-15
```

Both the Newton and Secant methods are implemented in the function `newton` of `scipy.optimize`.

```
import scipy.optimize as spo
def f(x):
    y = np.log(x)+x
    return y
def df(x):
    y = 1.0/x+1.0
    return y
x0 = 1.0
x = spo.newton(f, x0, df, tol=1.e-4, maxiter=50)
print('The approximate solution x is: ', x)
print('And the value f(x) is: ', f(x))

#output
The approximate solution x is:  0.5671432903993691
And the value f(x) is:  -2.877842408821607e-11
```

In `scipy.optimize` one can find also a hybrid method that works in a more broader spectrum of problems compared to the previous implementation. This method is implemented in the function `fsolve`.

```
import scipy.optimize as spo
def f(x):
    y = np.log(x)+x
```

```
    return y
def df(x):
    y = 1.0/x+1.0
    return y
x0 = 1.0
x = spo.fsolve(f, x0, fprime=df, xtol=1.e-4)
print('The approximate solution x is: ', x)
print('And the value f(x) is: ', f(x))

#output
The approximate solution x is:  [0.56714329]
And the value f(x) is:  [3.4803842e-09]
```

Application in Astrophysics:

If ψ is the mean anomaly of the orbit of a plant, then θ, the eccentric anomaly, can be computed by solving the fixed point equation

$\theta = \psi + e \sin\theta$
where e is the eccentricity of the elliptical orbit.

This equation can be solved seamlessly using the function `fixed_point` of `scipy.optimize`.

```
import numpy as np
import scipy.optimize as spo
def g(theta):
    e = 1.e-6
    psi = np.pi/6.0
    return psi+e*np.sin(theta)
theta0 = np.pi/6.0
theta = spo.fixed_point(g, theta0)
print('eccentric anomaly=', theta)

#Output
eccentric anomaly= 0.5235992755987319
```

Knowing the eccentric anomaly can lead to the estimation of the heliocentric distance $r = a(1 - e\cos\theta)$ where `a` is the semi-major axis.