II. Roots of High-Degree Equations

High-degree equations can be polynomials or equations containing radicals and/or transcendental (trigonometric and logarithmic) functions. The simplest example of high-degree equations is the quadratic equation:

$$ax^2 + bx + c = 0$$

Its roots can easily be obtained by the equation:

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

But in case of equations of higher degrees (power) or when terms of transcendental functions exist, numerical methods become the only way to obtain their roots.

Simple Iterations Method

The simple iterations is the simplest method used in finding roots of high-degree equation. In this method, the equations is rearranged to have the variable on the left side or the equation. An initial value of the variable, known as initial guess, substituted in the right side and a new value for the variable calculated. In the second iteration (repeat) the new value substituted on the right side. This cycle will continue until the old value and resulting new value of the variable become, theoretically, equal. Thus, this value will be one of the roots of the equations. Other root can be found with the same manner just by changing the initial guess.

Now, let's write solution steps in the following algorithm:

1- Rearrange the equation so that the variable is put on the left side
2- Assume (guess) an initial value of the variable to start the first iteration
3- Substitute the value of the variable in the right side of the equation and calculate an *new* value for the variable
4- If the new value of the variable is not equal to the previous value, consider the new one as the value of the variable.
5- Repeat steps 3 and 4 until the new value is equal to the old value of the variable.
6- In case the new value does not approach the old value (difference increases at each iteration) stop calculation and try another initial value or another rearrangement of the given equation.

Example 1: Find the roots of the following equation:

$$2x^2 - 5x + 3 = 0$$

(Analytical solutions: $x = 1$ and $x = 1.5$)

Solution: The first step in algorithm should be performed manually. So, the equation can be rearranged in the forms:

$$x = \frac{2x^2 + 3}{5}$$

or

$$x = \sqrt{\frac{5x - 3}{2}}$$

Let's **try** to code the algorithm as following:

```
x = 0                        # the initial guess
for iteration in range(1,101):   # assume 100 iterations are enough
    print(iteration, x)      # print number of iterations and x
    xnew = (2*x**2 + 3)/5    # calculates and outputs value of xnew
    if x == xnew:            # condition of equality
        break                # break the for-loop
    x = xnew                 # assign the value of xnew to x
print(iteration, xnew)       # print number of iterations and xnew at
                             end of the loop
```

To print the number of iterations, one is added to `iteration` because the for-loop variable will count from 0 to 99.

Accuracy of a solution: The last two lines of the output show the values at the last iteration for x an xnew, respectively,

```
100 0.9999999999394882
100 0.9999999999515905
```

Since the solution is obtained by using the computer, the values are directly affected by the precision of the data types of the computer variables. For example, let's type this simple summation on Python Shell and print the result:

```
>>> s = 0.1 + 0.2
>>> print(s)
0.30000000000000004
```

Now, let's check the type of the variable `s`

```
>>> type(s)
<class 'float'>
```

This shows that the `float` data type which is equivalent to `double` in some other languages is accurate up to the 16th decimal digit in this test. This means that to satisfy the condition of *exact equality* is extremely difficult if not impossible. So, the normal practice in numerical methods is to specify the **degree of accuracy** required in the solution. For example, accuracy up to the third decimal digit can be acceptable for an architect when the values are in meter.

According to this concept, the condition should be modified in a way that can consider the required degree of the accuracy for the problem. In many references, it is called the **tolerance** and it represents the acceptable **absolute maximum difference** between values of the variable of the equation resulted from the two successive iterations, in other words, the difference between the old and new values. Therefore, the code can be as following

```
x = 0                            # the initial guess
for iteration in range(1,101):   # assumes 100 iterations are enough
    print(iteration, x)          # print iteration number and x
    xnew = (2*x**2 + 3)/5        # calculates value of xnew
```

```
    if abs(x - xnew) < 0.000001: # degree of accuracy condition
        break                    # exit the for-loop
    x = xnew                     # assign the value of xnew to x
print(iteration, xnew)           # print number of iterations and
                                 xnew at end of the loop
```

Finally, instead of cluttering the output window by the output at each iteration, it is preferred to displays the final solution in smaller number of decimal digits and the corresponding number of iterations at the end of the program.

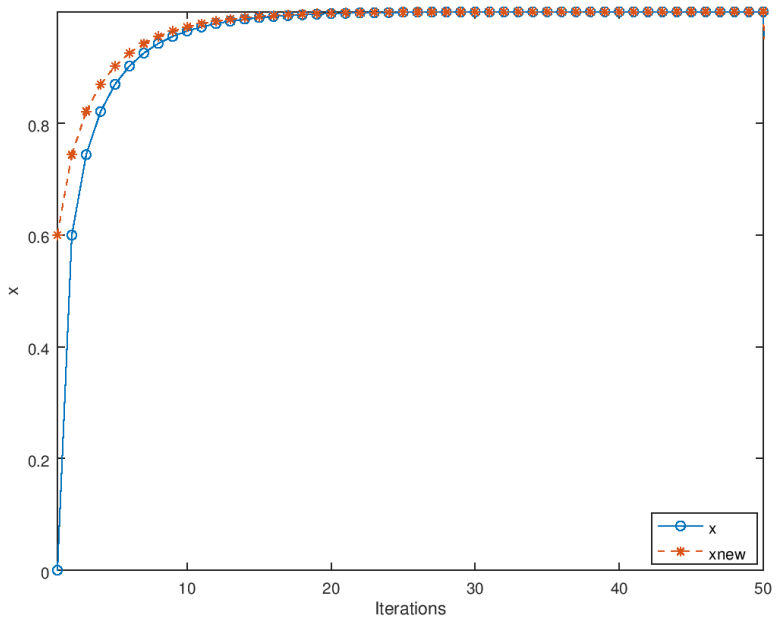
```
x = 0
for iteration in range(1,101):
    xnew = (2*x**2 + 3)/5
    if abs(x - xnew) < 0.000001:
        break
    x = xnew
print('The root: %.5f' % xnew)
print('Number of iterations: %d' % iteration)
```

the output will be:

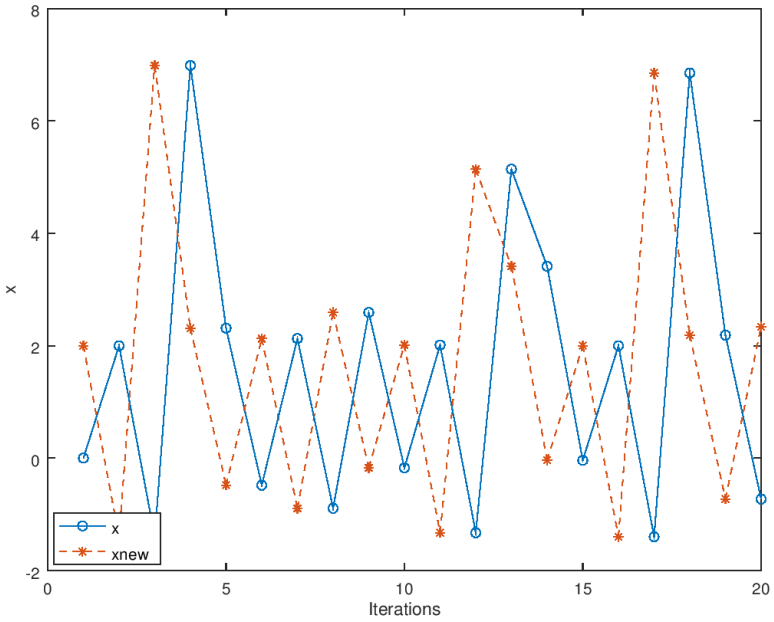
```
The root: 1.00000
Number of iterations: 50
```

To find the second root, different initial guesses can be tested. If the solution is not approached, the second rearrangement of the given equation should be applied.

Convergence and Divergence: The solution obtained for the quadratic equation given above showed the two values of the variable *get closer* at each iteration and at the iteration number 50 both values are considered equal when required condition of accuracy was satisfied. This behavior is known as the convergence of values to a specific solution. The following graph shows the convergence of the values to 1.



On the contrary, the divergence occurs when the difference between the values of the variable gets larger at each iteration until the final value of iterations loop is reached. The divergence can be resulted from many factors according to the type of the equation, the initial guess and/or method of rearrangement. The following graph shows the behavior of the variables in case of divergence for the equation $x\cos(x) - 1 = 0$



Newton-Raphson Method

The Newton-Raphson method has the same procedure of the simple iteration with a main difference in the step one. Instead of simple rearrangement of the given equation, a new equation should be formulated by using the given equation and its first derivative with respect to its variable. The formula to be put in the code is as following:

$$x_{new} = x - \frac{f(x)}{f'(x)}$$

Example 2: Find the roots of the following equation:

$$2x^2 - 5x + 3 = 0$$

Solution: Let's write the equation as a function then find its first derivative:

$$f(x) = 2x^2 - 5x + 3$$

So,

$$f'(x) = 4x - 5$$

Now, the formula that will be "plugged" in the code will be

$$x_{new} = x - \frac{2x^2 - 5x + 3}{4x - 5}$$

The code will be

```
x = 0
for iteration in range(1,101):
    xnew = x-(2*x**2-5*x+3)/(4*x-5) # Newton-Raphson's formula
    if abs(x - xnew) < 0.000001:
        break
    x = xnew
print('The root: %.5f' % xnew)
print('Number of iterations: %d' % iteration)
```

the output will be:

```
The root: 1.00000
Number of iterations: 6
```

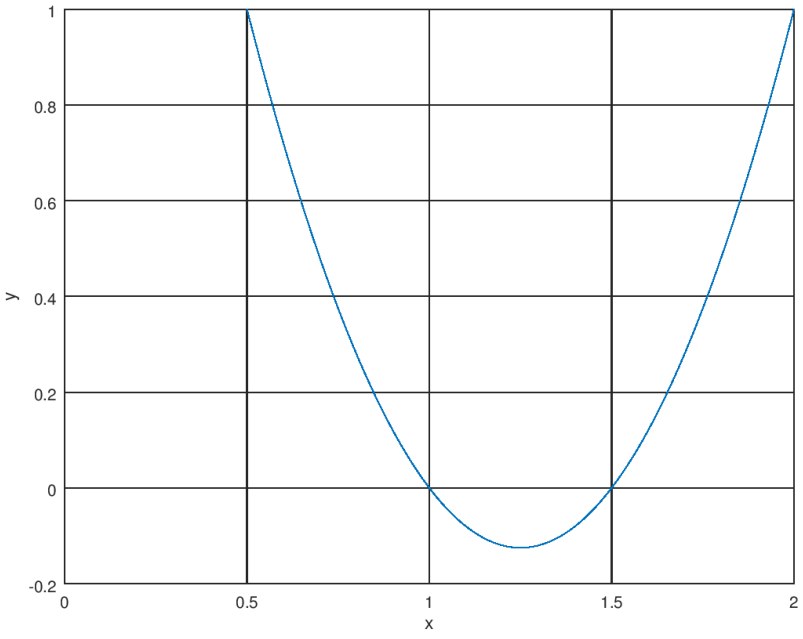
Amazing! The root have been reached at six iterations only. This demonstrates the efficiency of the Newton's method since it could converge about 8 times faster than the simple iterations method.

Another advantage is that only on formulation of the given equation can be used in computing the other root(s). For the equation given in the example, the second root, 1.5, can be simply obtained by setting the initial guess value to 2 (x = 2) and the output will be

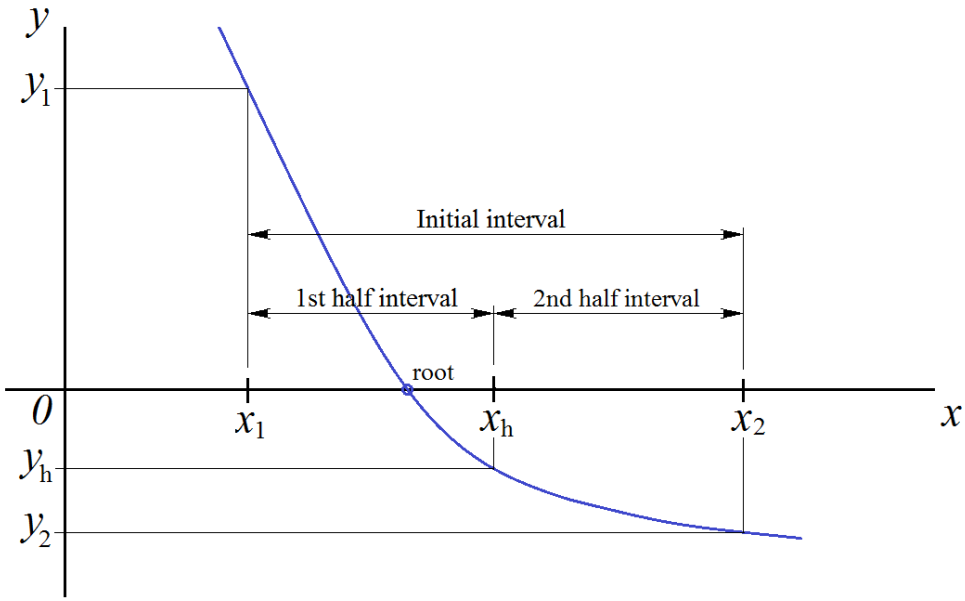
```
The root: 1.50000
Number of iterations: 5
```

Bisection Method

This method is based on the fact that a root of an equation is point where its curve crosses the *x*-axis. For example, the graph of the equation $y = 2x^2 - 5x + 3$ is



The curve crosses the x-axis at 1 and 1.5 (i.e. when $y = 0$) which are the roots of the equation. Accordingly, we can simply imagine that the bisection method as a search for these points along the x-axis. It actually searches for the point that is embraced by two values of x having corresponding values of y of different signs. Then interval between the x values is halved and the half containing the root is halved again and so on. By successive bisection operations, the interval becomes smaller at each time and x_1 and x_2 approach the root. At that point, y_1 and y_2 approach to zero. By changing starting search points the other roots can be determined in the same way.



The algorithm:

1- Input the values of x that embrace the interval where the root is expected
2- Calculate corresponding values for y
3- Check for the sign difference between y-values
4- In case of same signs, stop
5- Calculate the value of x in the half of the interval
6- Check for the sign difference between the y-values first half interval
7- In case of opposite signs, let x_1 and x_2 be the limits of the first half interval
8- In case of similar signs, let x_1 and x_2 be the limits of the second half interval
9- If the values of y approaches zero, print the x-value and stop
10- Else repeat steps 5 to 10

Example 3: Find the roots of the following equation:

$$2x^2 - 5x + 3 = 0$$

Solution: Let's write the equation as a function

$$y = 2x^2 - 5x + 3$$

Let's "translate" the algorithm to a computer code to solve the equation:

```
x1 = 0                          # first value of the interval
x2 = 1.2                        # second value of the interval
```

```
y1 = 2*x1**2-5*x1+3        # calculation of y1
y2 = 2*x2**2-5*x2+3        # calculation of y2
if y1*y2 > 0:              # test if signs are similar
    print('No roots exist within given interval')
    exit                   # terminate the program
for i in range(1,101):     # assumed 100 bisections are enough
    xh = (x1+x2)/2         # calculation the half value
    yh = 2*xh**2-5*xh+3    # calculation of yh
    y1 = 2*x1**2-5*x1+3    # calculation of y1
    if abs(y1) < 1.0e-6:   # condition of approach to solution
        break              # exit the loop
    elif y1*yh < 0:        # check for the sign change in first half
        x2 = xh            # let x2 be the midpoint value
    else:                  # in case sign change in the second half
        x1 = xh            # let x1 be the midpoint value
print('The root: %.5f' % x1)
print('Number of bisections: %d' % i)
```

The run of the program displays:

```
The root: 1.00000
Number of bisections: 21
```

Changing the initial interval to `x1 = 1.1` and `x2 = 2`, gives the second root:

```
The root: 1.50000
Number of bisections: 21
```

This could be the simplest working program of bisection method, but there are three modifications that should be made to improve the program:

1- The calculation of the values of y was repeated four times in the code without any change in the formula except substitution of different values of x. This can cause computation errors, especially in large equations. The solution is to define a function that includes the equation and calling it with passing the suitable value of x at each time. There two simple ways in Python to construct a one-line function:

   i- Lambda function: for example: `y = lambda x: 2*x**2 - 5*x + 3`

   ii- General function definition, for example:

```
def y(x):
    y = 2*x**2 - 5*x + 3
    return y
```

   or in simply:

```
def y(x):
    return 2*x**2 - 5*x + 3
```

2- Since the search of the interval that contains the root requires many trials with different values of x_1 and x_2, it is better to use the run-time **input** function instead of defining the values inside the code.

The modifications are applied on following program

```
def y(x):                          # define the function y(x)
    y = 2*x**2 - 5*x + 3
    return y

x1 = float(input('Enter the value of x1: '))    # input x1
x2 = float(input('Enter the value of x2: '))    # input x2
y1 = y(x1)                         # calling the function y(x1)
y2 = y(x2)                         # calling the function y(x2)

if y1*y2 > 0:                      # test if signs are similar
    print('No roots exist within given interval')
    exit

for i in range(100):
    xh = (x1+x2)/2
    yh = y(xh)              # calling the function y(xh)
    y1 = y(x1)              # calling the function y(x1)
    if abs(y1) < 1.0e-6:
        break
    elif y1*yh < 0:
        x2 = xh
    else:
        x1 = xh
print('The root: %.5f' % x1)
print('Number of bisections: %d' % (i+1))
```

For example, if x_1 and x_2 are input as 1 and 1.1, respectively, the run will be as following

```
Enter the value of x1: 1
Enter the value of x2: 1.1
x1 is one of the roots
```

Root finding functions in SciPy

In **SciPy** (Scientific Python), there are many root finding functions in the *optimizing and root finding module*: scipy.optimize to solve different types of equations by means of advanced numerical methods. In this section, let's solve the example given above by using the functions: `newton()`, `bisect()`, `fsolve()` and `root()` in the following Python console lines.

```
>>>  from scipy.optimize import newton, bisect, fsolve, root
>>> f = lambda x: 2*x**2-5*x + 3
```

```
>>> print(newton(f, 0))   # given the function and trial value
0.9999999999999998
>>> print(newton(f, 3))
1.4999999999999993

>>> print(bisect(f,0,1.2)) # given the function and initial interval
0.9999999999996361
>>> print(bisect(f,1.2,4))
1.4999999999994542

>>> print(fsolve(f,0))     # given the function and trial value
[ 1.]
>>> print(fsolve(f, 2))
[ 1.5]
>>> print(fsolve(f,[0, 1, 2]))
[ 1.   1.   1.5]
```

In the last statement, the trial values (initial guesses) are given as a list and the function returned an array containing the roots that can be obtained by each guess value.

```
>>> print(root(f, 0))      # given the function and trial value
    fjac: array([[-1.]])
     fun: array([ 0.])
 message: 'The solution converged.'
    nfev: 11
     qtf: array([ -1.03264508e-10])
       r: array([ 1.00000103])
  status: 1
 success: True
       x: array([ 1.])
```

The root() function returns a number of attributes containing information about the solution process. Since the sought root(s) are in the array x, the function can be called as following

```
>>> print(root(f, 0).x)
[ 1.]

>>> print(root(f, 2).x)
[ 1.5]

>>> print(root(f, [0,1,2]).x)
[ 1.   1.   1.5]
```

The returned values from the functions can be assigned to a variable.

```
>>> x = fsolve(f, 3)
>>> print(x)
[ 1.5]         # array of one element
>>> print(x[0])
1.5            # value of element 0
```

```
>>> x = root(f, [0,1,2]).x
>>> print(x[0])
0.999999999999
```

For more information about the function in the optimizing and root finding module of SciPy:
https://docs.scipy.org/doc/scipy/reference/optimize.html