# NUMERICAL SOLUTION AND SIMULATION IN ENGINEERING WITH PYTHON

By:

Ayoob Salari

Chapter 8

# Table of Contents

SECTION 1

# Systems of Nonlinear Equations

This chapter presents the advanced numerical topic of solving non-linear systems of equations. The application of Python's `fsolve` function to solve one-dimensional non-linear equations is described. The use of the Newton-Raphson method to solve a system of nonlinear equations is also described. An application to power flow and and an example of chaotic fractal are covered.

## 1.1 Root Finding and Non-Linear Systems of Equations

We start with some simple examples of non-linear equations and their solution.
Examples of non-linear equations are

$$xe^{\frac{-x}{10}} + x^2 sin(x) = 1$$

and

$$5x^3 - 2x^2 + 4x = 3.$$

Solving single non-linear equations such as these is referred to as *finding the roots* of the equation.
In contrast, an example of a *system* of non-linear system equations is shown below:

$$sin(x) + y^2 + ln(z) = 7 \quad (1)$$

$$3x + 2^y - z^3 = -1 \quad (2)$$

$$x + y + z = 5 \quad (3)$$

This system of equations can be written in vector form as:

$$\boldsymbol{f}(x, y, z) = \mathbf{0}$$

or even

$$\boldsymbol{f}(\boldsymbol{x}) = \mathbf{0}$$

where $\boldsymbol{f}$ is the vector function:

$$\boldsymbol{f} = \begin{cases} f_1(x,y,z) = sin(x) + y^2 + ln(z) - 7 \\ f_2(x,y,z) = 3x + 2^y - z^3 + 1 \\ f_3(x,y,z) = x + y + z - 5 \end{cases}$$

Finding the roots of a single non-linear equation is much easier and qualitatively different than solving a system of non-linear equations. The reason for this difference is that in one dimension it is possible to systematically *bracket* a root between two values. One can then squeeze the brackets closer and closer together, trapping the root. For instance, if a one-dimensional function is continuous and has a positive value at $x_1$ and a negative value at $x_2$, then it has to cross the $x$-axis somewhere in between these values and there must be root between $x_1$ and $x_2$. With a multi-dimensional set of non-linear equations, on the other hand, one is never sure that there is a root until it is found.

## 1.2 One-Dimensional Non-Linear Equations

### 1.2.1 Polynomial Equations

As we have seen before, finding the zeros of a polynomial equation in MATLAB is performed using the `nroots` function. For example, the roots of the equation:

$$10x^{10} + 9x^9 + 8x^8 + 7x^7 + 6x^6 + 5x^5 + 4x^4 + 3x^3 + 2x^2 + x + 1 = 0$$

```
import sympy as sym
x = sym.symbols("x")
p = 10*x**10  + 9*x**9 + 8*x**8 + 7*x**7 + 6*x**6 +5*x**5
   + 4*x**4 + 3*x**3 + 2*x**2 + x +1
roots_value = sym.nroots(p)
print("Roots=", roots_value)
```

```
Roots =  [-0.7931313 - 0.2432934*I, -0.7931313 + 0.2432934*I,
-0.5136180 - 0.6356577*I, -0.5136180 + 0.6356577*I,
-0.0761635 - 0.7835333*I, -0.0761635 + 0.7835333*I,
0.3079765 - 0.6796694*I, 0.3079765 + 0.6796694*I,
0.6249363 - 0.4898256*I, 0.6249363 + 0.4898256*I]
```

Numpy `roots` module is another approach of solve polynomial problems as is shown below

```
from numpy import roots
coeff = [10, 9, 8, 7, 6, 5, 4, 3, 2, 1, 1]
print("Roots=",roots(coeff))
```

```
Roots= [-0.79313137+0.24329342j -0.79313137-0.24329342j -0.51361802+0.63565773j
 -0.51361802-0.63565773j -0.0761635 +0.78353334j -0.0761635 -0.78353334j
  0.62493632+0.48982563j  0.62493632-0.48982563j  0.30797657+0.67966949j
  0.30797657-0.67966949j]
```

**Example 1.1** Roots of polynomials

Find the roots of:

(i) $x^5 - 5x^4 + 10x^3 - 10x^2 + 5x - 1 = 0$

(ii) $x^3 - 1 - (\sqrt{3}/2)i = 0$

Solution:

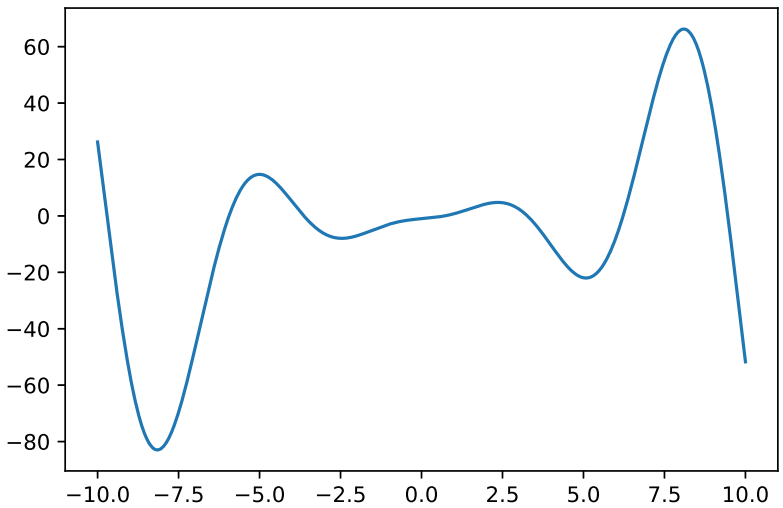
### 1.2.2 Transcendental Equations

Finding the roots of non-linear equations involving transcendental functions is challenging because the number of roots may not be known, and there may even be an infinite number of roots. In solving transcendental equations we encounter a property that is common to almost all algorithms for solving non-linear systems of equations: they involve making an initial guess an then iterating until some convergence criterion is met. Consider the solution of

$$xe^{\frac{-x}{10}} + x^2 sin(x) = 1.$$

A first step in solving such an equation could be to plot it:

```
import sympy as sym
x = sym.Symbol('x')
p1 = sym.plot(x*sym.exp(-x/10) + x**2 *sym.sin(x) -1, show
   =False)
```



The figure shows that there are a number of zeros. Suppose we wish to find the zeros between 0 an 5. we can use the figure's zoom function to find the zeros. `fsolve` is one of the most flexible approaches. This function needs an initial estimate, and it has the benefit of being able to determine the zeros of a general multi-variable function.

```
import sympy as sym
from scipy.optimize import fsolve
def f(x):
    out = x[0]*sym.exp(-x[0]/10) + x[0]**2 *sym.sin(x[0])
       -1
    return out
x = fsolve(f, [3])
print(x)
```

```
[3.26896511]
```

**Example 1.2** Roots of Transcendental Equations

Find the roots of

(i) $\sin(\frac{1}{x}) = 0$ (the root near 0.18)

(ii) $x - \tanh(3x) = 0$ (the root near 0.9)

(iii) $\cos\left[(x^2+5)(x^4+1)^{-1}\right] = 0$ (all roots)

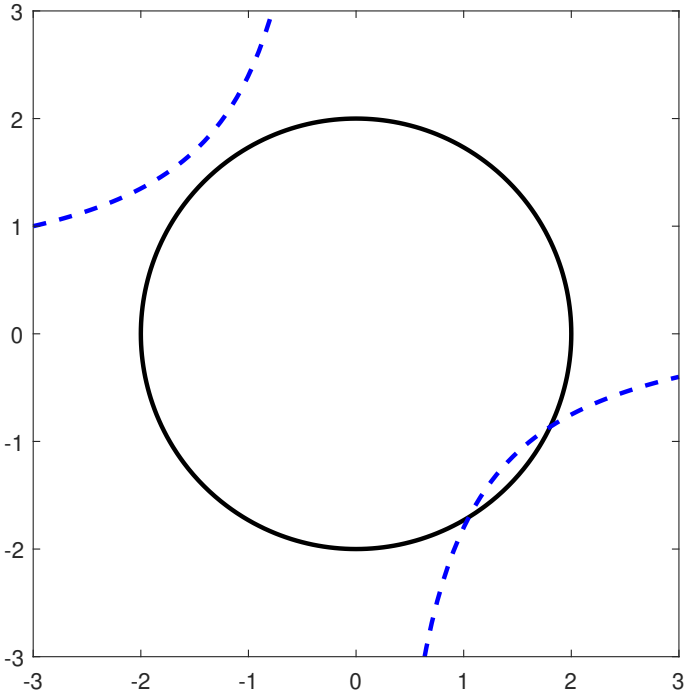
Solution:

## 1.3 Multi-Dimensional Systems of Non-Linear Equations

The solution of a system of non-linear functions such as:

$$\boldsymbol{f}(x,y) = \mathbf{0} \leftrightarrow \begin{cases} x^2 + y^2 - 4 = 0 \\ y + \frac{2.1}{x} - 0.3 = 0 \end{cases}$$

corresponds to the crossing of the zero contour boundaries of the individual functions. A plot of the zero contour boundaries for this system of equations is shown below.

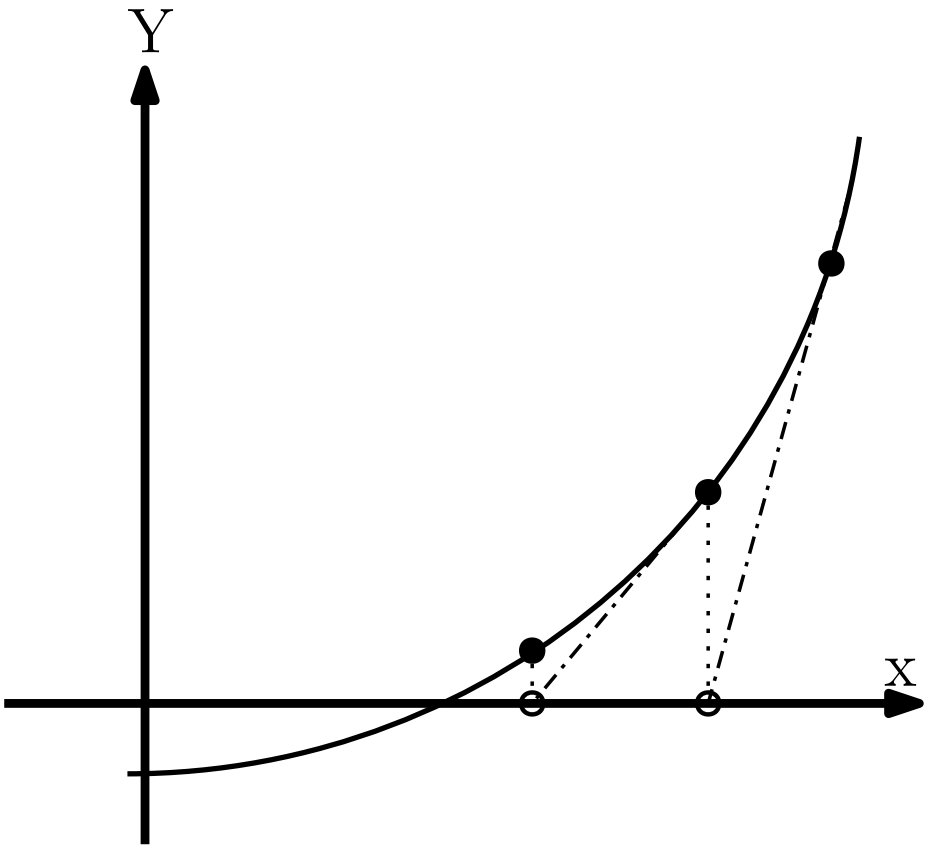



The figure shows that there are two roots to this system of equations in the lower right-hand corner. In the upper left corner there is "almost a zero-crossing," and it is regions such as these that can cause problems for numerical algorithms. Indeed, it has been said (Press, 1992) that "there are no good, general methods for solving systems of more than one nonlinear equation. Furthermore it is not hard to see why (very likely) there never will be any good, general methods." This is related to the "no free lunch" theorems for optimisation and machine learning of Wolpert and Macready:

> We have dubbed the associated results "No Free Lunch theorems" because they demonstrate that if an algorithm performs well on a certain class of problems then it necessarily pays for that with degraded performance on the set of all remaining problems.

### 1.3.1 Newton-Raphson method

The simplest an most straight-forward method for solving systems of non-linear equations is the Newton-Raphson method. However, its success is tied to a good first guess for the solution, and it can fail spectacularly. This method is usually describe for a single nonlinear equation in first year calculus books as *Newton's* method. The Newton-Raphson formula consists of geometrically extending the tangent line at a current point of the curve until is crosses the $x$-axis an then setting the next guess to the $x$-value of the zero-crossing as shown in the figure below.




In one-dimension, the Newton-Raphson method consists of using an initial estimate to calculate improve estimates $x_{(1)}$, $x_{(2)}$,... (where the iteration number is in braces) according to the following iteration formula:

$$x_{(n+1)} = x_{(n)} - f(x_{(n)}) \left( \frac{df(x_{(n)})}{dx} \right)^{-1}$$

The Newton-Raphson procedure for a single function can be extended to multi-dimensional vector functions. Let the vector function $\boldsymbol{f}$ be given as:

$$\boldsymbol{f}(\boldsymbol{x}) = \boldsymbol{0},$$

where $\boldsymbol{x}$ is a $n \times 1$ vector of variables. The vector function notation is a concise way to write the following set of simultaneous equations:

$$f_1(x_1, x_2, \ldots, x_n) = 0 \tag{4}$$

$$f_2(x_1, x_2, \ldots, x_n) = 0 \tag{5}$$

$$\vdots \tag{6}$$

$$f_n(x_1, x_2, \ldots, x_n) = 0 \tag{7}$$

The iteration $\boldsymbol{x}_{(n+1)}$ is given by:

$$\boldsymbol{x}_{(n+1)} = \boldsymbol{x}_{(n)} - \frac{\boldsymbol{f}(\boldsymbol{x}_{(n)})}{\boldsymbol{J}(\boldsymbol{x}_{(n)})}$$

where $\boldsymbol{J}$ is the $n \times n$ Jacobian matrix of partial derivatives:

$$\boldsymbol{J} = \frac{d\boldsymbol{f}}{d\boldsymbol{x}} = \begin{bmatrix} \frac{df_1}{dx_1} & \frac{df_1}{dx_2} & \cdots & \frac{df_1}{dx_n} \\ \frac{df_2}{dx_1} & \frac{df_2}{dx_2} & \cdots & \frac{df_2}{dx_n} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{df_n}{dx_1} & \frac{df_n}{dx_2} & \cdots & \frac{df_n}{dx_n} \end{bmatrix}$$

Example 1.3 **Newton-Raphson**

Write a Python function which uses the Newton-Raphson method to solve arbitrary systems of symbolic equations. The input to the function should be:

1. the symbolic vector function,
2. the symbolic independent variables,
3. an initial estimate for $\boldsymbol{x}$ (a column vector), and
4. the required accuracy (the process should terminate when the absolute values of are all less than the specified accuracy).

The function should return:

1. the final values for $\boldsymbol{x}$,
2. the value of the vector corresponding to the final value of $\boldsymbol{f}(\boldsymbol{x})$
3. the number of iterations required, and
4. the final Jacobian matrix.

Use your Newton-Raphson function to solve the following system of nonlinear equations:

$$x^2 + y^2 - 4 = 0$$
$$y + \frac{2.1}{x} - 0.3 = 0$$

from an initial point of [3, -1.5].

Solution:

**Exercise 1.1**

Use `fsolve` and calculate the solution to the following system of nonlinear equations:

$$\begin{aligned} \sin x + y^2 + \ln z - 7 &= 0 \\ 3x + 2^y - z^3 + 1 &= 0 \\ x + y + z - 5 &= 0 \end{aligned}$$

from an initial point of [0, 2, 2].

Solution:

```
import sympy as sym
from scipy.optimize import fsolve
def f(x):
    a = sym.sin(x[0]) + x[1] **2 + sym.ln(x[2]) - 7
    b = 3*x[0] + 2**x[1] -x[2] **3 + 1
    c = x[0] + x[1]  + x[2]  - 5
    return [a, b, c]
x = fsolve(f, [0,2,2])
print(x)
```

Example 1.4 Power flow studies

The analysis of normal steady-state operation of a power system is called a *power-flow study* (or sometimes a load-flow study). The objective is to determine the voltages, currents, and real and reactive power flows in a system under a given load conditions, which is important in operation an planning of electrical power systems. Real life problems involve the solution of hundreds, sometimes thousands or tens-of-thousands of such equations.
In the solution for the bus voltages in a three bus power system, the following equations might arise:

$$f_1(\boldsymbol{x}) = -0.7 + 8.4 \sin x_1 + 7.35 x_3 \sin(x_1 - x_2) \tag{8}$$

$$f_2(\boldsymbol{x}) = 3 + 10 x_3 \sin x_2 + 7.35 x_3 \sin(x_2 - x_1) \tag{9}$$

$$f_3(\boldsymbol{x}) = 1.25 - 10 x_3 \cos x_2 - 7.35 x_3 \cos(x_2 - x_1) + 17 x_3^2 \tag{10}$$

Test your Newton-Raphson function with an initial estimate of `[0, 0, 1]`$^{\mathrm{T}}$ for an an accuracy of $10^{-6}$. You should find that 4 iterations are required.
*Answer*: $\boldsymbol{x}$ = [ -0.0466 -0.2084 0.9225 ]$^{\mathrm{T}}$.

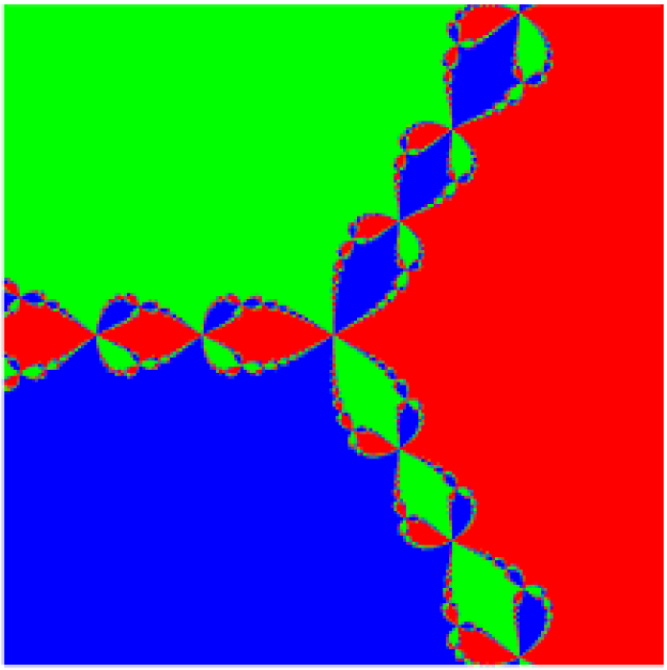
Solution:

## 1.4 Newton-Raphson Fractal

The Newton-Raphson algorithm is valid in the complex plane and can be use to generate *fractals*, which are images that illustrate chaotic behaviour. For example, consider the equation:

$$z^3 - 1 = 0$$

which has a single real root at $z = 1$, but which also has complex roots at the other two cube roots of unity, $e^{\pm\frac{2\pi j}{3}}$. The Newton-Raphson iteration formula for this equation can be applied to complex numbers, and will converge to one of the three roots. We can therefore map out the complex plane into regions from which a starting value does or does not iterate to the real root 1. If we paint each point in the complex plane that iterates to 1 red and the complex roots green or blue, we will generate a fractal (shown below).



**Example 1.5** Newton-Raphson Fractal

To what solution does the Newton-Raphson algorithm converge on for the following points: $0.5$, $0.5i$, $-0.5i$?

Solution: