NUMERICAL SOLUTION AND SIMULATION IN ENGINEERING WITH PYTHON

By:

Ayoob Salari

Chapter 4

Table of Contents

SECTION 1

Numerical Methods

The first part of this tutorial continues the basics of programming in Python started in previous section. It is designed to give you more practice in Python programming. Mainly, the exercises go beyond those in the previous tutorial in that you are now being asked to devise a solution and then translate this solution into Python code.
The second part of this section presents an introduction to numerical methods. Matrix decompositions and methods for solving systems of linear equations are introduced. The application of QR factorisation to find the least-square solution to a set of over-determined linear equations is demonstrated.

1.1 Turning problems into code

Get into some good habits:

- Try to devise a solution on paper first, and then translate this solution into Python code.
- Use comments to guide you as you develop your answers.
- Try out unit testing on individual code modules.
- Keep a log of your bugs and mistakes, and learn from them

Example 1.1 GPA Calculation

Honours are awarded on the basis of a "Grand WAM", an average mark over all units of study, weighted according to credit point value and according to the nominal year of each unit of study. The formula is:

$$\text{GWAM} = \sum M_i C_i Y_i / \sum C_i Y_i$$

Where M is the mark, C the credit point value and Y the year (values of 1, 2, 3 or 4). Honours are awarded as follows:

- H1 for GWAM ≥ 75,
- H2(1) for $75 >$ GWAM ≥ 70,
- H2(2) for $70 >$ GWAM ≥ 65.

Write a function which will accept a 3-column matrix containing a mark, credit point value and year in each row and will return a value for the GWAM (it could consist of just one line).
Joe obtained the following results :

1^{st} Year		2^{nd} Year		3^{rd} Year		4^{th} Year	
Mark	CP	Mark	CP	Mark	CP	Mark	CP
66	6	68	4	76	4	74	12
54	6	77	4	70	4		
		70	4	69	4		

What grade of Honours will he be awarded?

Solution:

Example 1.2 Legendre polynomials

Legendre polynomials, $P_n(x)$, $n = 0, 1, \cdots, K$, can be computed using *Bonnet's recursion formula*, which is defined as follows:

$$nP_n(x) = (2n-1)xP_{n-1}(x) - (n-1)P_{n-2}(x), \quad n = 2, 3, \cdots, K,$$

with $P_0(x) = 1$ and $P_1(x) = x$.
Write a function that takes an integer n and returns the coefficients of $P_n(x)$ in descending order of powers. Use only 1 `for` loop. Then use the function `polyval` to evaluate the Legendre polynomial of degree 8 at $x = 0.6$.

Solution:

1.2 Numerical approximation

Series expansions are used to calculate the value of many trigonometric functions. For example, the value of cos(x) can be expressed as a Maclaurin series:

$$\cos(x) = 1 - \frac{x^2}{2!} + \frac{x^4}{4!} - \frac{x^6}{6!} + \cdots = \sum_{k=1}^{\infty} (-1)^{k-1} \frac{x^{2(k-1)}}{(2(k-1))!}$$

Likewise, the value of $\sin(x)$ is given by:

$$\sin(x) = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{1!} + \cdots = \sum_{k=1}^{\infty} (-1)^{k-1} \frac{x^{2k-1}}{(2k-1)!}$$

In practice, these functions' values are approximated by truncating the corresponding series at an appropriate level of accuracy. This approximation methods is the workhorse of many numerical techniques, as demonstrated in the exercise below.

Example 1.3 Elliptic Integral

The elliptic integral,

$$K(k^2) = \int_0^{\pi/2} \frac{dt}{\sqrt{1 - k^2 \sin^2(t)}}, \quad 0 < k < 1,$$

cannot be evaluated in terms of elementary functions. Carl Friedrick Gauss (1777-1855) devised an algorithm to solve this integral which uses a sequence of arithmetic means $\{a_n\}$ and geometric means $\{b_n\}$, where

$$a_0 = 1, \; b_0 = \sqrt{1 - k^2},$$
$$a_n = \frac{a_{n-1} + b_{n-1}}{2}, \; b_n = \sqrt{a_{n-1} b_{n-1}}, \; n = 1, 2, \cdots, K,$$

Both sequences have the same limit g, and $K(k^2) = \frac{\pi}{2g}$. Also, $a_n > b_n$ for all n.
Write a Python function that computes the elliptic integral $K(k^2)$. Use a `while` loop to generate the sequences and continue looping until $a_n - b_n < \epsilon$, where ϵ is the relative floating point accuracy value.
Write your code so that the function takes either a single number or an array of numbers as k^2-values (i.e. make it compatible with vector inputs). Your code should use only 1 loop, the `while` loop. Return the answer in the same format (row or column vector) as the input. What is the value of the elliptic integral for $k^2 = 0.3, 0.4, 0.5, 0.6$?

Solution:

1.3 Circuit analysis applications

We will now move on to some circuit analysis applications. It may be useful to work through Example 10.5 in Moore – Solving Simultaneous Equations: An Electrical Circuit – before moving on to Exercise 7.

Example 1.4 Circuit analysis

Electrical resistors are said to be connected "in series" if the same current passes through each and "in parallel" if the same voltage is applied across each.
If in series, they are equivalent to a single resistor with resistance given by:

$$R = R_1 + R_2 + \cdots + R_n,$$

and if in parallel, their equivalent resistance is given by:

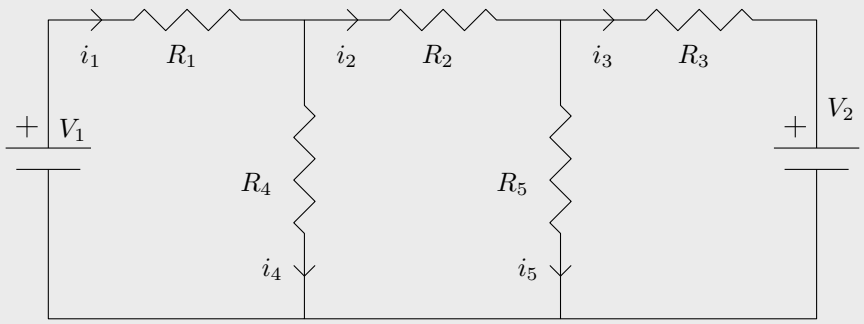
$$\frac{1}{R} = \frac{1}{R_1} + \frac{1}{R_2} + \cdots + \frac{1}{R_n}.$$

Write an Python script that prompts the user to input the individual resistors resistances, and the type of connection (series or parallel) and then computes the equivalent resistance.

Solution:

Example 1.5 Circuit analysis

Consider the following circuit



a. Derive the equations describing the circuit above, and encode them in a script for values $V_4 = 100$v, $R_1 = 5$ $k\Omega$, $R_2 = 100$ $k\Omega$, $R_3 = 200$ $k\Omega$, $R_4 = 150$ $k\Omega$ and $R_5 = 250$ $k\Omega$. Store the equations in a matrix format i.e. $Ax = b$. A should be a matrix consisting of coefficients of the current variables, x should be your column vector comprising of current variables and b should be a column vector containing the values of the constants/voltages.

   Hint: Use KCL and KVL to come up with 5 equations. Arrange them in the order $Ax = b$ and see what values you need to store for each matrix.

b. Suppose the resistors are rated to carry a current of no more than 1 mA. What is the allowable range of positive values for the voltage V2?

   Hint: You can vary V_2 from 0 to 100v and test to see what values are allowable.

c. Investigate how the resistance of R_3 limits the allowable range (minimum

and maximum values) for V_2, by generating a plot of the allowable limit on V_2 as a function of R_3 for the range $150 \leq R_3 \leq 250k\Omega$.

Solution:

1.4 Optimisation

Engineers work to improve designs and operations. One way of doing this is called optimization, which uses a mathematical description of a design to select the best values of certain variables. Many sophisticated mathematical tools for large complicated optimization problems are possible via various optimisation libraries of Python (ex. `scipy`). However, some problems can be solved in a more direct way, using exhaustive search. The next problem is an example of distribution centre location optimization, which you might see is related to planning problems that arise in power and telecommunications.

Example 1.6 Optimisation

A company wants to locate a distribution call centre that will serve six of its major customers in a 30×30km area. The customers' locations are given in the following table, listing their (x, y) coordinates, alongside the volume that must be delivered each week:

Costumer	x location(km)	y location(km)	Volume(tonnes/week)
1	1	28	3
2	6	18	8
3	8	16	7
4	19	2	5
5	25	10	2
6	27	8	3

a. Plot the locations of each customer, and include a legend. The weekly delivery cost c_i for customer i depends on both the volume v_i and distance d_i from the distribution centre (assume the distances are straight-line distances) and is given by $c_i = 0.5 d_i v_i$.

b. Write a script that finds the location of the distribution centre that minimizes the total weekly cost to service all six customers, to the nearest km, and returns the weekly cost.

c. Generate a `contour` plot of the costs for all locations.

Solution:

```
import numpy as np
```

```
import matplotlib.pyplot as plt
x = np.array([1,6,8,19,25,27])
y = np.array([28,18,16,2,10,8])
# Part a
n = ['C1 - 3 Tonnes', 'C2 - 8 Tonnes','C3 - 7 Tonnes','C4 - 5
Tonnes','C5 - 2 Tonnes','C6 - 2 Tonnes',]
plt.scatter(x, y)
for i, txt in enumerate(n):
    plt.annotate(txt, (x[i], y[i]))
plt.grid()
# Part b
volume=np.array([3,8,7,5,2,5])
min_cost=10**5
cost=np.zeros((30,30))
for i in range(0,30):
    for j in range(0,30):
        distan = np.sqrt( (i - y)**2  + (j - x)**2   )
        cost[i,j]  = 0.5 * np.dot(distan,volume)
optimal = np.where(cost ==np.amin(cost))
listOfCordinates =list(zip(optimal[1], optimal[0]))
print("The X-Y Co-ordinate is:", listOfCordinates)
# Part c
X=range(0,30)
Y=range(0,30)
CS= plt.contour(X,Y,cost)
plt.clabel(CS, fontsize=10, inline=1,fmt ='%1.0f ')
```

1.5 Efficient methods for solving systems of linear equations

We now consider efficient numerical methods for solving systems of linear equations and least squares problems, using the LU and QR matrix factorisations.

Numerical Efficiency Numerical methods are evaluated according to their computational efficiency. To measure computation time, one can use `time.process_time()` in Python.

Example 1.7 Numerical Efficiency

Determine the time required to multiply two 1000×1000 random matrices.

Solution:

```
import numpy as np
from time import process_time_ns
```

```
A = np.random.random((1000,1000))
B = np.random.random((1000,1000))
t_start = process_time_ns()
C = np.dot(A,B)
t_stop = process_time_ns()
print("Elapsed time in nanoseconds:", t_stop-t_start)
```

1.5.1 Matrix Decompositions and Linear Systems of Equations

Linear systems of equations are a set of equations such as:

$$x+5y+8z=5$$
$$2x-4y+10z=1$$
$$6x+2y-5z=3$$

You have solved several such equations in your Linear Algebra course using Gaussian elimination. Although Gaussian elimination is useful for solving a matrix equation by hand, it is not an efficient computational algorithm and is not used in numerical computations. The best way to solve a system of linear equations such as $\mathbf{M}x = b$ in Python is to just type $x = M\backslash b$. This is much more efficient than computing `inv(M)*b`. The backslash operator solves the equation using an LU matrix factorisation, which is the most common numerical method for solving linear equations. Matrix decompositions refer to a factorization of matrices and are extremely useful. We examine two matrix decompositions: (1) LU decomposition; and (2) QR decomposition.

LU Decomposition A matrix, **M**, can be factored so that **PM** = **LU**, where **L** is a lower triangular matrix, **U** is an upper triangular matrix, and **P** is a permutation matrix that permutes the rows of **M** (the permutation matrix results from pivoting to avoid round-off errors). For example,

```
import numpy as np
from scipy.linalg import lu_factor
M = np.array([[1,5,8],[2,-4,10],[6,2,-5]])
print("M=", M)
lu_M, piv = lu_factor(M)
print("LU=", lu_M)
print("P=", piv)
```

```
M= [[ 1  5  8]
    [ 2 -4 10]
    [ 6  2 -5]]
LU=  [[ 6.          2.         -5.        ]
      [ 0.33333333 -4.66666667 11.66666667]
      [ 0.16666667 -1.         20.5       ]]
```

```
P=  [2 1 2]
```

In the above example, matrix *LU* contains U in its upper triangle, and L in its lower triangle. The unit diagonal elements of L are not stored. P consists of Pivot indices representing the permutation matrix P: row i of matrix was interchanged with row P[i]. It is worth mentioning that there is another way to calculate the LU decomposition as is shown below

```
import numpy as np
from scipy.linalg import lu
M = np.array([[1,5,8],[2,-4,10],[6,2,-5]])
print("M=", M)
P, L, U = lu(M)
print("np.dot(P.T,M)=", np.dot(P.T, M))
print("np.dot(L,U)=", np.dot(L, U))
print("P=", P)
print("L=", L)
print("U=", U)
```

```
M= [[ 1  5  8]
    [ 2 -4 10]
    [ 6  2 -5]]
np.dot(P.T,M)=  [[ 6.  2. -5.]
                 [ 2. -4. 10.]
                 [ 1.  5.  8.]]
np.dot(L,U)=  [[ 6.  2. -5.]
               [ 2. -4. 10.]
               [ 1.  5.  8.]]
P= [[0. 0. 1.]
    [0. 1. 0.]
    [1. 0. 0.]]
L= [[ 1.          0.          0.        ]
    [ 0.33333333  1.          0.        ]
    [ 0.16666667 -1.          1.        ]]
U= [[ 6.          2.         -5.        ]
    [ 0.         -4.66666667 11.66666667]
    [ 0.          0.         20.5       ]]
```

However, it should be noted that this second method of LU decomposition is not very efficient in solving system of linear equations.
Using an LU decomposition, the matrix equation $\mathbf{M}x = b$ can be re-written as $\mathbf{LU}x = \mathbf{P}b$ and one can use `lu_solve` to solve the system of equations. This method solves system in two steps:

1. solve $\mathbf{L}y = \mathbf{P}b$

2. solve $\mathbf{U}x = y$.

For example, if $\mathbf{M}$ and b are

```
import numpy as np
M = np.array([[1,5,8],[2,-4,10],[6,2,-5]])
b = np.array([[5],[1],[3]])
```

then the solution to $\mathbf{M}x = b$ can be determined as follows:

```
from scipy.linalg import lu_factor, lu_solve
lu_M = lu_factor(M)
x = lu_solve(lu_M, b)
```

The reason for using an LU decomposition is that equations (1) and (2) involve triangular matrices which are easily solved by a sequence of substitutions. This is very efficient numerically. Importantly, $\mathbf{L}$ and $\mathbf{U}$ are reusable. Suppose we want to solve $\mathbf{M}x = a$, $\mathbf{M}x = b$ and $\mathbf{M}x = c$. In other words, suppose we have a number of linear systems with the same matrix. Using the `np.linalg.solve(M, b)` three times is not the best method of solution. Instead, it is better to use a single LU decomposition to find the matrix $\mathbf{LU}$ then solve the three equations using $\mathbf{LU}$ as above.

Example 1.8 Linear System of Equations

Compare the computation time required for solving $\mathbf{M}x = a$, $\mathbf{M}x = b$, and $\mathbf{M}x = c$ using the `np.linalg.solve` and LU decomposition, where
M = np.random.rand(5000, 5000)
a = np.random.rand(5000, 1)
b = np.random.rand(5000, 1)
c = np.random.rand(5000, 1)

Solution:

```
import numpy as np
from time import process_time_ns
from scipy.linalg import lu_factor, lu_solve
# Using np.linalg.solve :
M, a = np.random.rand(5000, 5000), np.random.rand(5000, 1)
b, c = np.random.rand(5000, 5000), np.random.rand(5000, 1)
t1_start = process_time_ns()
x1_a = np.linalg.solve(M, a)
x1_b = np.linalg.solve(M, b)
x1_c = np.linalg.solve(M, c)
t1_stop = process_time_ns()
print("Elapsed time using solve in nanosec:", t1_stop-t1_start)
# Using LU decomposition
t2_start = process_time_ns()
```

```
lu_M = lu_factor(M)
x2_a = lu_solve(lu_M, a)
x2_b = lu_solve(lu_M, b)
x2_c = lu_solve(lu_M, c)
t2_stop = process_time_ns()
print("Elapsed time using LU in nanosec:", t2_stop-t2_start)
```

QR Decomposition A matrix **M** can be factored so that **M** = **QR** where **Q** is an orthogonal matrix and **R** is an upper triangular matrix. The fact that **Q** is an orthogonal matrix means that the columns of **Q** are unit basis vectors. For example, $\mathbf{Q}(:,n)^T * \mathbf{Q}(:,n) = 1$ and $\mathbf{Q}(:,n)^T * \mathbf{Q}(:,m) = 0$ $(m \neq n)$. Furthermore, $\mathbf{Q}^T * \mathbf{Q} = \mathbf{I}$. For example,

```
import numpy as np
from numpy.linalg import qr
M = np.array([[1, 2, 3],[4, 5, 6], [7, 8, 10]])
q, r = qr(M)
print('Q:', q)
print('R:', r)
QR = np.dot(q, r)
print('QR:', b)
```

```
Q: [[-0.12309149  0.90453403  0.40824829]
    [-0.49236596  0.30151134 -0.81649658]
    [-0.86164044 -0.30151134  0.40824829]]
R: [[ -8.1240384   -9.6011363  -11.93987462]
    [  0.           0.90453403   1.50755672]
    [  0.           0.           0.40824829]]
QR: [[ 1.  2.  3.]
     [ 4.  5.  6.]
     [ 7.  8. 10.]]
```

The QR decomposition is commonly used to find least-square solutions to linear problems such as curve fitting. Suppose we would like to fit the curve 2/3 2

$$C_1 x + C_2 x^{2/3} + C_3 x e^{-2x} + C_4 \cos(\ln 5x) + C_5,$$

to a set of data. In this case, we cannot directly use curve fitting utility because it is not a polynomial function. Nonetheless, the least-square approach can still solve this problem because we have a set of linear equations for the coefficients C_i . For example, suppose that the data is given in a set of column vectors x and y and that we want to use the above function to map the data in x to the data in y. One first forms the matrix **M**. Then, the equation $y = \mathbf{M}C$ represents the curve fitted to the data, where C is a vector of length 5 (the 5 coefficients). This is the important point: by forming the matrix **M** using

the values for x, we obtain a set of linear equations for the coefficients C describing the curve.

To perform the curve fitting, one finds the best least-square error solution to the equation $y = \mathbf{M}C$. In other words, we find C that minimizes:

$$||y - \mathbf{M}C|| = \left((y - \mathbf{M}C)^T (y - \mathbf{M}C)\right)^{1/2}$$

The expression with the double brackets is referred to as the norm of the vector. The norm is equal to the square root of the dot product of the vector with itself. For example,

$$||v|| = \left(v^T v\right)^{1/2} = \sqrt{\sum_i v_i^2}.$$

In this case, the vector is $y = \mathbf{M}C$, which is the error vector between the true y and the curve approximation $\mathbf{M}C$. Thus, the C that minimizes $||y - \mathbf{M}C||$ gives the coefficients for the fitted curve.

The solution to the minimization problem using the QR decomposition method is based on the fact that the norm of a vector is preserved when multiplied by an orthogonal matrix. That is to say, if $\mathbf{Q}$ is an orthogonal matrix, then we have:

$$||\mathbf{Q}x|| = \left[(\mathbf{Q}x)^T (\mathbf{Q}x)\right]^{1/2} = \left[x^T \mathbf{Q}^T \mathbf{Q}x\right]^{1/2} = \left[x^T x\right]^{1/2} = ||x||.$$

Using this property and the QR decomposition of $\mathbf{M}$, we obtain:

$$\begin{aligned} ||y - \mathbf{M}C|| &= ||\mathbf{Q}^T (y - \mathbf{M}C)|| \\ &= ||\mathbf{Q}^T y - \mathbf{Q}^T \mathbf{M}C|| \\ &= ||\mathbf{Q}^T y - \mathbf{Q}^T \mathbf{Q}\mathbf{R}C|| \\ &= ||\mathbf{Q}^T y - \mathbf{R}C||. \end{aligned}$$

In order to understand the advantage of the last form of the equation, we must consider the QR decomposition of an over-determined system of equations (as occurs during curve fitting). If the matrix $\mathbf{M}$ represents an over-determined set of equations then it is not a square matrix; it has more rows than columns. For instance, the QR decomposition of a random 5×3 matrix gives:

```
import numpy as np
from numpy.linalg import qr
M = np.random.rand(5, 3)
q, r = qr(M)
print('M=', M)
print('Q=', q)
print('R=', r)
```

```
M= [[0.5771165  0.88787116 0.02005312]
    [0.11522494 0.86712121 0.78307694]
    [0.9798506  0.03243016 0.59140814]
```

```
    [0.84387828 0.65033309 0.73617805]
    [0.35613615 0.3282058  0.84822425]]
Q= [[-0.39401091 -0.47478613  0.67848668]
    [-0.07866675 -0.70638792 -0.42486487]
    [-0.66896688  0.50159084 -0.0854651 ]
    [-0.5761354  -0.11981062 -0.05761516]
    [-0.24314246 -0.09821551 -0.59035741]]
R= [[-1.46472214 -0.89421992 -1.0955133 ]
    [ 0.         -1.1279579  -0.43754286]
    [ 0.          0.         -0.91281136]]
```

In order to find C that minimizes $||y - \mathbf{M}C||$ in a least-square sense, one uses the following Python commands to solve $\mathbf{Q}^T y - \mathbf{R}C = 0$

```
q, r = qr(M)
z = np.dot(q.T,y)
C= np.linalg.solve(r,z)
```

Example 1.9 Linear System of Equations

In this exercise, we try to fit the amplitude-modulated wave

$$y = \cos(x)\cos(10x + \pi/3),$$

by the sum of six sinusoidal functions:

$$C_1\cos(9x) + C_2\sin(9x) + C_3\cos(10x) + C_4\sin(10x) + C_5\cos(11x) + C_6\sin(11x).$$

Use the following commands to generate the sample data:
x = np.arange(-4*np.pi, 4*np.pi, 0.01)
y = np.cos(x)*np.cos(10*x + np.pi/3)

Solution:

```
import numpy as np
from numpy.linalg import qr
x = np.arange(-4*np.pi,4*np.pi,0.01)
y =  np.cos(x)*np.cos(10*x + np.pi/3)
M = np.transpose (np.array([np.cos(9*x), np.sin(9*x), np.cos(10*
x), np.sin(10*x), np.cos(11*x), np.sin(11*x)]))
q, r = qr(M)
z = np.dot(q.T,y)
C= np.linalg.solve(r,z)
print("C = ", C)
```