VI. Systems of Linear Equations

Many problems in science and engineering involve systems of linear equations that contain specific number of equations equal to the number of unknowns. The systems are always represented in matrix and vector forms. Thus, all methods of solution actually consists of matrix and vector operations.

$$\begin{aligned}
a_{1,1}x_1 + a_{1,2}x_2 + \cdots + a_{1,n}x_n &= b_1 \\
a_{2,1}x_1 + a_{2,2}x_2 + \cdots + a_{2,n}x_n &= b_2 \\
\vdots \qquad \vdots \qquad\qquad \vdots \quad &\quad \vdots \\
a_{n,1}x_1 + a_{n,2}x_2 + \cdots + a_{n,n}x_n &= b_n
\end{aligned}$$

The matrix form:

$$\begin{bmatrix} a_{1,1} & a_{1,2} & \dots & a_{1,n} \\ a_{2,1} & a_{2,2} & \dots & a_{2,n} \\ \vdots & \vdots & & \vdots \\ a_{n,1} & a_{n,2} & \dots & a_{n,n} \end{bmatrix} \begin{Bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{Bmatrix} = \begin{Bmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{Bmatrix}$$

There are two main types of methods of solution applied in numerical analysis to determine the unknowns of linear systems:

- **Elimination Methods**: which eliminates parts of the coefficients matrix by means of row and column operations in order to reach a form that yields the solution by simple calculations. They are known as direct method also.
- **Iterative Methods**: where the equations are rearranged in a way that enables recursive calculations of values of the unknown until convergence is reached. Similar to simple iteration method, an initial guess for all unknowns is required.

Gauss Elimination Method

It is the most known method in solving linear equation systems and considered the basis of the other elimination methods. It consists two stages; the first is the elimination of the elements under the main diagonal of the coefficients matrix and the second stage is the back substitution of the solved unknowns until all system is solved.

Stage 1: the elimination

This algorithm requires three nested loops:

1- Loop of k from 1 to n-1 to index the fixed rows and eliminated columns
2- Loop of i from k+1 to n to index the subtracted rows
3- Loop of j from k to n to index the columns for element subtraction
   the elimination statement: $a_{i,j} = a_{k,j} - \frac{a_{k,k}}{a_{i,k}} a_{i,j}$

The new constant terms are calculated in the i-loop: $b_i = b_k - \frac{a_{k,k}}{a_{i,k}} b_i$ a

At the end of this stage, the whole elements under the main diagonal should equal to zero.

Stage 2: the back substitution

1- Starting with last row, let $x_n = b_n / a_{n,n}$
2- For rows from $i = n$-1 to 1: substitute values of obtained for x_{i+1} and compute the x_i values of the current row. This can be expressed in the formula:

$$x_i = \frac{b_i - \sum_{j=i+1}^{n} a_{i,j} x_j}{a_{i,i}}$$

At the end, the all values of vector x will be determined and, thus, the system is solved.

Example 1: Solve the system:

$$\begin{bmatrix} 2 & 7 & -1 & 3 & 1 \\ 2 & 3 & 4 & 1 & 7 \\ 6 & 2 & -3 & 2 & -1 \\ 2 & 1 & 2 & -1 & 2 \\ 3 & 4 & 1 & -2 & 1 \end{bmatrix} \begin{Bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \end{Bmatrix} = \begin{Bmatrix} 5 \\ 7 \\ 2 \\ 3 \\ 4 \end{Bmatrix}$$

answer: x = {0.44444, 0.55556, 0.66667, 0.22222, 0.22222}

Solution: Let's carry out programming in the two stages mentioned above. But first of all the matrix and vectors should be defined as following

```
from numpy import array, zeros      # array module
a = array([[2, 7, -1, 3, 1],        # matrix of coefficients
           [2, 3, 4, 1, 7],
           [6, 2, -3, 2, -1],
           [2, 1, 2, -1, 2],
           [3, 4, 1, -2, 1]],float)
b = array([5, 7, 2, 3, 4], float)   # vector of constant terms
n = len(b);               # the size of the system
x = zeros(n,float);       # creating solution vector
```

Stage 1: The elimination consists three nested loops:

1- The loop from row 1 to row n-1 that is fixed during multiplication and subtraction.

```
for k in range(n-1):
```

2- The loop for all rows below the fixed row. The factor to be multiplied to the elements of the row is calculated here. Multiplication and subtraction of right-hand vector (*b*) is done here.

```
for i in range(k+1, n):
        fctr = a[k, k] / a[i, k]       # multiplier of each row
        b[i] = b[k] - fctr*b[i]        # elimination of {b}
```

3- The loop of columns of the coefficient matrix from that equal to the number of current fixed row to the end of columns. Multiplication by the factor and subtraction of coefficients matrix is done here.

```
for j in range(k, n):
      a[i, j] = a[k, j] - fctr*a[i, j]    # eliminiation of [a]
```

Stage 2: The back substitution starts with calculation `x(n)` then substitution to upper rows is performed by two nested loops:

1- The loop of all rows above the last one, where known values of x from lower rows are substituted in the equation to obtain the unknown x of the current row.

```
x[n-1] = b[n-1] / a[n-1, n-1]     # computation of the nth x value
for i in range(n-2,-1,-1):        # loop from (n-1)th to first row
```

2- The loop for all columns of known values on *x*. This loop includes summation of coefficients multiplied by known *x* values of the current row. Just after the end of this loop the current unknown *x* is calculated.

```
terms = 0
  for j in range(i+1, n):
      terms += a[i, j] * x[j]
  x[i] = (b[i] - terms) / a[i, i]
```

Finally, necessary display statements are added to print the required data and the solution:

```
from numpy import array, zeros
a = array([[2, 7, -1, 3, 1],
           [2, 3, 4, 1, 7],
           [6, 2, -3, 2, -1],
           [2, 1, 2, -1, 2],
           [3, 4, 1, -2, 1]],float)
b = array([5, 7, 2, 3, 4], float)
n = len(b);
x = zeros(n,float);
# Elimination
for k in range(n-1):
    for i in range(k+1, n):
        fctr = a[k, k] / a[i, k]
        b[i] = b[k] - fctr*b[i]
        for j in range(k, n):
            a[i, j] = a[k, j] - fctr*a[i, j]
# Back-substitution
x[n-1] = b[n-1] / a[n-1, n-1]
for i in range(n-2,-1,-1):
    terms = 0
    for j in range(i+1, n):
        terms += a[i, j] * x[j]
    x[i] = (b[i] - terms) / a[i, i]
```

```
print('The solution of the system:')
print(x)
```

The run of the code gives:

```
The solution of the system:

[ 0.44444444  0.55555556  0.66666667  0.22222222  0.22222222]
```

Example 2: Solve the system:

$$\begin{bmatrix} 0 & 7 & -1 & 3 & 1 \\ 2 & 3 & 4 & 1 & 7 \\ 6 & 2 & 0 & 2 & -1 \\ 2 & 1 & 2 & 0 & 2 \\ 3 & 4 & 1 & -2 & 1 \end{bmatrix} \begin{Bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \end{Bmatrix} = \begin{Bmatrix} 5 \\ 7 \\ 2 \\ 3 \\ 4 \end{Bmatrix}$$

answer: x = { 0.021705, 0.792248, 1.051163, 0.158140, 0.031008}

Solution: If the given system is defined and the code is run, the following output will appear:

```
The solution of the system:
[ nan  nan  nan  nan  nan]
```

Nan (not a number) appears when some values approach to infinity because of division by values approaching to zero then they used in other calculations. For example,

```
>>> from numpy import Inf
>>> Inf -Inf
nan
>>> Inf / Inf
nan
```

In Gauss elimination method, this problem occurs when an element used in computing the multiplication factor of the corresponding row is zero. If it is in the main diagonal, it can be displaced by interchanging the rows. The entire row (including its constant term value) is replaced with the next row. Since interchanging rows of a system does not affect its solution, this technique can be applied just before elimination as following:

Step 1: Check the current row for zero elements in the main diagonal.

Step 2: If a zero found interchange the corresponding row with the next row

Furthermore, to avoid division by zero, an if-statement can be added just before the calculation of the factor. If the element to be the divisor is zero, which is what is intended by the operation, pass all statements of the current row and **continue** to the next row.

Thus, the final code will be as following

```
from numpy import array, zeros
a = array([[0, 7, -1, 3, 1],
           [2, 3, 4, 1, 7],
           [6, 2, 0, 2, -1],
           [2, 1, 2, 0, 2],
           [3, 4, 1, -2, 1]],float)
b = array([5, 7, 2, 3, 4], float)
n = len(b);
x = zeros(n,float);
# Elimination
for k in range(n-1):
    if a[k,k] == 0:     # if equal to zero interchange rows
        for j in range(n):
            a[k,j], a[k+1,j] = a[k+1,j], a[k,j] # swap by tuples
        b[k], b[k+1] = b[k+1], b[k]             # swap by tuples
    for i in range(k+1, n):         # Eliminination of next rows
        if a[i, k] == 0: continue   # if it is zero, jump to next row
        fctr = a[k, k] / a[i, k]
        b[i] = b[k] - fctr*b[i]
        for j in range(k, n):
            a[i, j] = a[k, j] - fctr*a[i, j]
print(a)
# Back-substitution
x[n-1] = b[n-1] / a[n-1, n-1]
for i in range(n-2,-1,-1):
    terms = 0
    for j in range(i+1, n):
        terms += a[i, j] * x[j]
    x[i] = (b[i] - terms) / a[i, i]

print('The solution of the system:')
print(x)
```

The run of the code displays:

```
The solution of the system:
[ 0.02170543  0.79224806  1.05116279  0.15813953  0.03100775]
```

Jacobi’s Method:

It is the most basic iterative method to solve the linear systems. It has the same concept of simple iterations method where the equations of the linear set are rearranged by taking a different variable from each one to the left hand side then, starting with an initial guess for all variables, the resulting values form the equations are substituted again at each iteration until convergence condition is satisfied for each variable.

Example 3: solve the system:

$$4x_1 + x_2 + 2x_3 - x_4 = 2$$

$$3x_1 + 6x_2 - x_3 + 2x_4 = -1$$

$$2x_1 - x_2 + 5x_3 - 3x_4 = 3$$

$$4x_1 + x_2 - 3x_3 - 8x_4 = 2$$

answer: *x* = { 0.36501,-0.23379, 0.28507, -0.20362}

Solution: To explain the concept, let's rearrange the system manually;

$$x_1 = -\frac{1}{4}(x_2 + 2x_3 - x_4 - 2)$$

$$x_2 = -\frac{1}{6}(3x_1 - x_3 + 2x_4 + 1)$$

$$x_3 = -\frac{1}{5}(2x_1 - x_2 - 3x_4 - 3)$$

$$x_4 = \frac{1}{8}(4x_1 + x_2 - 3x_3 - 2)$$

This rearrangement can simply be expressed in the following formula:

$$x_i^* = -\frac{1}{a_{i,i}}\left(\sum_{\substack{j=1 \\ j\neq i}}^{n} a_{i,j}x_j - b_i\right)$$

The superscript (*) in x_i indicates that this is a *new* value of the current iteration. So, as a first step this formula can be programmed by using two for-loops:

The *i*-loop to solve each equation for x_i , and the *j*-loop to perform summation the remaining terms in addition to the right hand value:

```
for i in range(n):
        s = 0
        for j in range(n):
            if j != i:
                s += a[i,j]*x[j]
        xnew[i] = -1/a[i,i] * (s - b[i])
```

Now, this part should be but inside a conditional iteration loop that ends at convergence of **all** variables. The loop should be given a limit of iterations so it stops at when convergence condition is not satisfied.

```
for iteration in range(iterlimit):
    for i in range(n):
        s = 0
        for j in range(n):
            if j != i:
                s += a[i,j]*x[j]
        xnew[i] = -1/a[i,i] * (s - b[i])
    if (abs(xnew - x) < tolerance).all(): # convergence condition
        break           # exit the loop
    else:
        x = copy(xnew)  # assign a copy of all xnew elements to x
```

After adding definitions of the matrix and vector of the problem, definition of the initial guess value to x vector and setting values of the iterations limit and the tolerance, the final code will as following:

```
from numpy import *            # import all contents of the module numpy
a = array([[4, 1, 2, -1],
           [3, 6, -1, 2],
           [2, -1, 5, -3],
           [4, 1, -3, -8]],float)
b = array([2, -1, 3, 2], float)
(n,) = shape(b)                # numpy array size function
x = full(n, 1, float)          # initial guess = 1.0
xnew = empty(n,float)          # construct xnew array
iterlimit = 100                # max number of iterations
tolerance = 1.0e-6             # degree of accuracy
# Iterations
for iteration in range(iterlimit):
    for i in range(n):
        s = 0
        for j in range(n):
            if j != i:
                s += a[i,j]*x[j]
        xnew[i] = -1/a[i,i] * (s - b[i])
    if (abs(xnew - x) < tolerance).all():# convergence condition
        break       # exit the loop
    else:
        x = copy(xnew)    # assign a copy of all xnew elements to x

print('Number of iterations: %d' % (iteration+1))
print('The solution of the system:')
print(x)
```

The output:

```
Number of iterations: 31
The solution of the system:
[ 0.3650067  -0.23378439  0.28506845 -0.20362079]
```

Gauss-Seidel Method

The Gauss-Seidel method is an iterative method and has an algorithm very similar to Jacobi's method. The only difference is that the obtained new values of x are applied into the coming equations within the same iteration while in Jacobi's method all new values are applied into the equations at the next iteration. So, its general form can be written as

$$x_i^* = -\frac{1}{a_{i,i}}\left(\sum_{\substack{j=1 \\ j\neq i}}^{n} a_{i,j}x_j^* - b_i\right)$$

where x_j^* is the new value of the variable from the previous equations.

This is achieved by changing the position of the new values assignment statement and consequently modify the convergence condition method. The program will be as following

```
from numpy import *
a = array([[4, 1, 2, -1],
           [3, 6, -1, 2],
           [2, -1, 5, -3],
           [4, 1, -3, -8]],float)
b = array([2, -1, 3, 2], float)
(n,) = shape(b)
x = full(n, 1, float)
xdiff = empty(n, float)     # x difference between every two iterations
iterlimit = 100
tolerance = 1.0e-6
# Iterations
for iteration in range(iterlimit):
    for i in range(n):
        s = 0
        for j in range(n):
            if j != i:
                s += a[i,j]*x[j]
        xnew = -1/a[i,i] * (s - b[i]) # xnew is scalar
        xdiff[i] = abs(xnew - x[i]) # compute the absolute difference
        x[i] = xnew                 # assign the new value to x[i]
    if (xdiff < tolerance).all(): # check convergence of all equations
        break
```

```
print('Number of iterations: %d' % (iteration+1))
print('The solution of the system:')
print(x)
```

the output becomes:

```
Number of iterations: 13
The solution of the system:
[ 0.36500739 -0.23378566  0.28506799 -0.20362001]
```

This simple change has effectively reduced the number of iterations in the same problem.

Diagonal dominance

Before applying either Jacobi's or Gauss-Seidel method, an important condition for convergence should be satisfied. It is known as *diagonal dominance* where the absolute value of the elements in the main diagonal should be the largest coefficient in each equation of the system. Thus, the equations of the system should be reordered to the diagonally dominant form.

Example 4: solve the system:

$$2x_1 - x_2 + 5x_3 - 3x_4 = 3$$

$$4x_1 + x_2 + 2x_3 - x_4 = 2$$

$$4x_1 + x_2 - 3x_3 - 8x_4 = 2$$

$$3x_1 + 6x_2 - x_3 + 2x_4 = -1$$

answer: *x* = { 0.36501,-0.23379, 0.28507, -0.20362}

Solution: This is the same system of the previous example. The only difference is in the order of the equations. For better visualization, let's but the system in matrix form:

$$a = \begin{bmatrix} 2 & -1 & \mathbf{5} & -3 \\ \mathbf{4} & 1 & 2 & -1 \\ 4 & 1 & -3 & \mathbf{-8} \\ 3 & \mathbf{6} & -1 & 2 \end{bmatrix}, \qquad b = \begin{Bmatrix} 3 \\ 2 \\ 2 \\ -1 \end{Bmatrix}$$

It is obvious that the largest absolute coefficient of each equation is not on the main diagonal of the coefficients matrix. To see the effect of this change on the solution by using Gauss-Seidel's method, the definitions of the arrays `a` and `b` are changed to:

```
a = array([[2, -1, 5, -3],
           [4, 1, 2, -1],
           [4, 1, -3, -8],
           [3, 6, -1, 2]],float)
b = array([3, 2, 2, -1], float)
```

Then the output becomes

```
Number of iterations: 100
The solution of the system:
```

```
[  8.43572172e+109  -2.71345274e+110  -1.09039090e+110
6.32980450e+110]
```

This shows that the solution has diverged, in other words, the limit of iterations has been reached with extremely far values.

Linear systems solution in NumPy and SciPy

The linear algebra modules *numpy.linalg* and *scipy.linalg* contain a large number of functions to manipulate matrices and solve different types of linear systems. In this section, two methods are used to solve the following examples

```
>>> from scipy.linalg import solve, inv
>>> from numpy import array, dot
>>> a = array([[0, 7, -1, 3, 1],    # Example 2
...            [2, 3, 4, 1, 7],
...            [6, 2, 0, 2, -1],
...            [2, 1, 2, 0, 2],
...            [3, 4, 1, -2, 1]],float)
>>> b = array([5, 7, 2, 3, 4], float)
>>> a1= array([[2, -1, 5, -3],      # Example 4
...            [4, 1, 2, -1],
...            [4, 1, -3, -8],
...            [3, 6, -1, 2]],float)
>>> b1 = array([3, 2, 2, -1], float)
```

1) The direct method by using the function solve():

```
>>> x = solve(a,b)
>>> print(x)
[ 0.02170543  0.79224806  1.05116279  0.15813953  0.03100775]
>>> x1 = solve(a1, b1)
>>> print(x1)
[ 0.36500754 -0.23378582  0.28506787 -0.20361991]
```

2) The inverse matrix multiplication method by using `inv()` and `dot()` functions

```
>>> x = dot(inv(a),b)
>>> print(x)
[ 0.02170543  0.79224806  1.05116279  0.15813953  0.03100775]
>>> x1 = dot(inv(a1), b1)
>>> print(x1)
```

```
[ 0.36500754 -0.23378582  0.28506787 -0.20361991]
```

Note that the matrix multiplication function `dot()`belongs to the numpy module.

The direct method is preferred because it returns solution in less number of matrix operations.

For more information about numpy.linalg and scipy.linalg, respectively:

https://docs.scipy.org/doc/numpy/reference/routines.linalg.html

https://docs.scipy.org/doc/scipy-0.14.0/reference/linalg.html