PROGRAM STUDI INFORMATIKA
FAKULTAS TEKNIK DAN INFORMATIKA
UNIVERSITAS MULTIMEDIA NUSANTARA
SEMESTER GENAP TAHUN AJARAN 2021/2022

IF420 – ANALISIS NUMERIK

Pertemuan ke 4 – Linear Algebra and Systems of Linear Equations

Seng Hansun, S.Si., M.Cs.

Capaian Pembelajaran Mingguan Mata Kuliah (Sub-CPMK):

Sub-CPMK 4: Mahasiswa mampu memahami dan menerapkan Aljabar Linear serta Sistem Persamaan Linear – C3

Reviews

- Reading and Writing Data
- Visualization and Plotting
- Parallel Your Python

Outlines

- Basics of Linear Algebra
- Linear Transformations
- Systems of Linear Equations
- Solutions to Systems of Linear Equations
- Solve Systems of Linear Equations in Python
- Matrix Inversion

Motivation

- Numerous problems in engineering and science can be described or approximated by **linear relationships**.
- For example, if you combine resistors in a complicated circuit, you will obtain a system of linear relationships.
- Moreover, the concept of linear algebra has been massively applied in the **machine learning** area.
- The study of **linear relationship** is contained in the field of **linear algebra**, and this chapter provides a basic overview of some basic linear algebraic **vocabulary** and **concepts** that are important for later chapters.
- However, the information in this chapter is in no way comprehensive and should not be considered a substitute for a full linear algebra course.

Basics of Linear Algebra - Sets

- In mathematics, a **set** is a **collection** of **objects**.
- Sets are usually denoted by **braces** {}. For example, S={orange,apple,banana} means "S is the set containing 'orange', 'apple', and 'banana'".
- The **empty set** is the set containing **no objects** and is typically denoted by **empty braces** such as {} or by Ø.
- Given two sets, A and B, the **union** of A and B is denoted by $A \cup B$ and equal to the set containing all the elements of A and B. The **intersect** of A and B is denoted by $A \cap B$ and equal to the set containing all the elements that belong to both A and B. In set notation, a **colon (:)** is used to mean "**such that**". The usage of these terms will become apparent shortly. The symbol ∈ is used to denote that an object is **contained** in a set. A backslash, ****, in set notation means **set minus**. So if $a \in A$ then $A \setminus a$ means "A minus the element, a."

Standard Number Sets

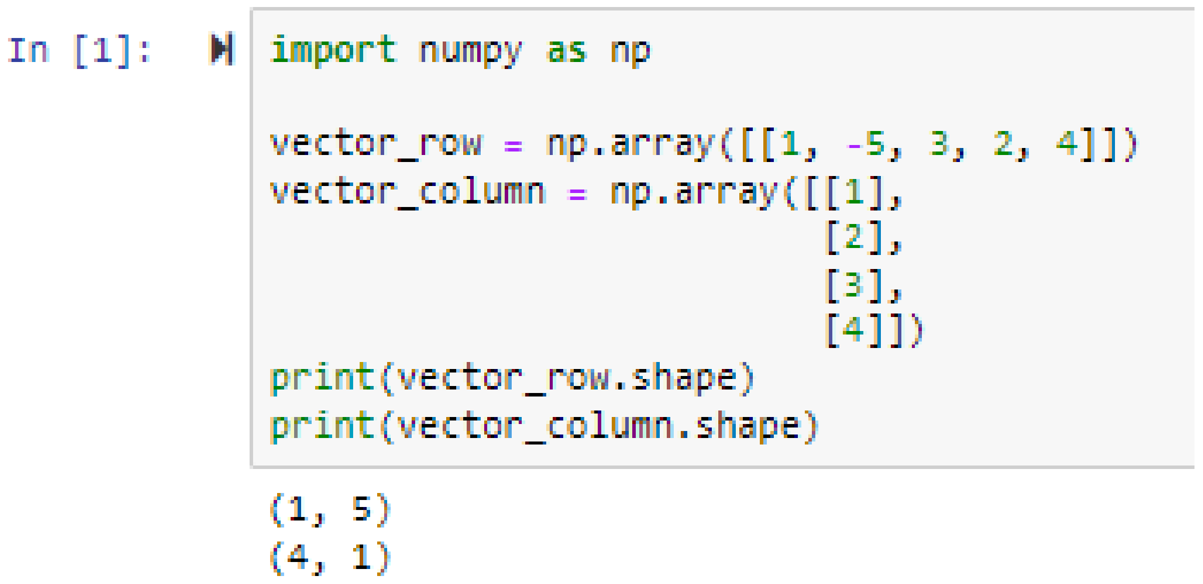
Set Name	Symbol	Description
Naturals	$\mathbb{N}$	$\mathbb{N} = \{1, 2, 3, 4, \cdots\}$
Wholes	$\mathbb{W}$	$\mathbb{W} = \mathbb{N} \cup \{0\}$
Integers	$\mathbb{Z}$	$\mathbb{Z} = \mathbb{W} \cup \{-1, -2, -3, \cdots\}$
Rationals	$\mathbb{Q}$	$\mathbb{Q} = \{\frac{p}{q} : p \in \mathbb{Z}, q \in \mathbb{Z}\backslash\{0\}\}$
Irrationals	$\mathbb{I}$	$\mathbb{I}$ is the set of real numbers not expressible as a fraction of integers.
Reals	$\mathbb{R}$	$\mathbb{R} = \mathbb{Q} \cup \mathbb{I}$
Complex Numbers	$\mathbb{C}$	$\mathbb{C} = \{a + bi : a, b \in \mathbb{R}, i = \sqrt{-1}\}$

Basics of Linear Algebra - Vectors

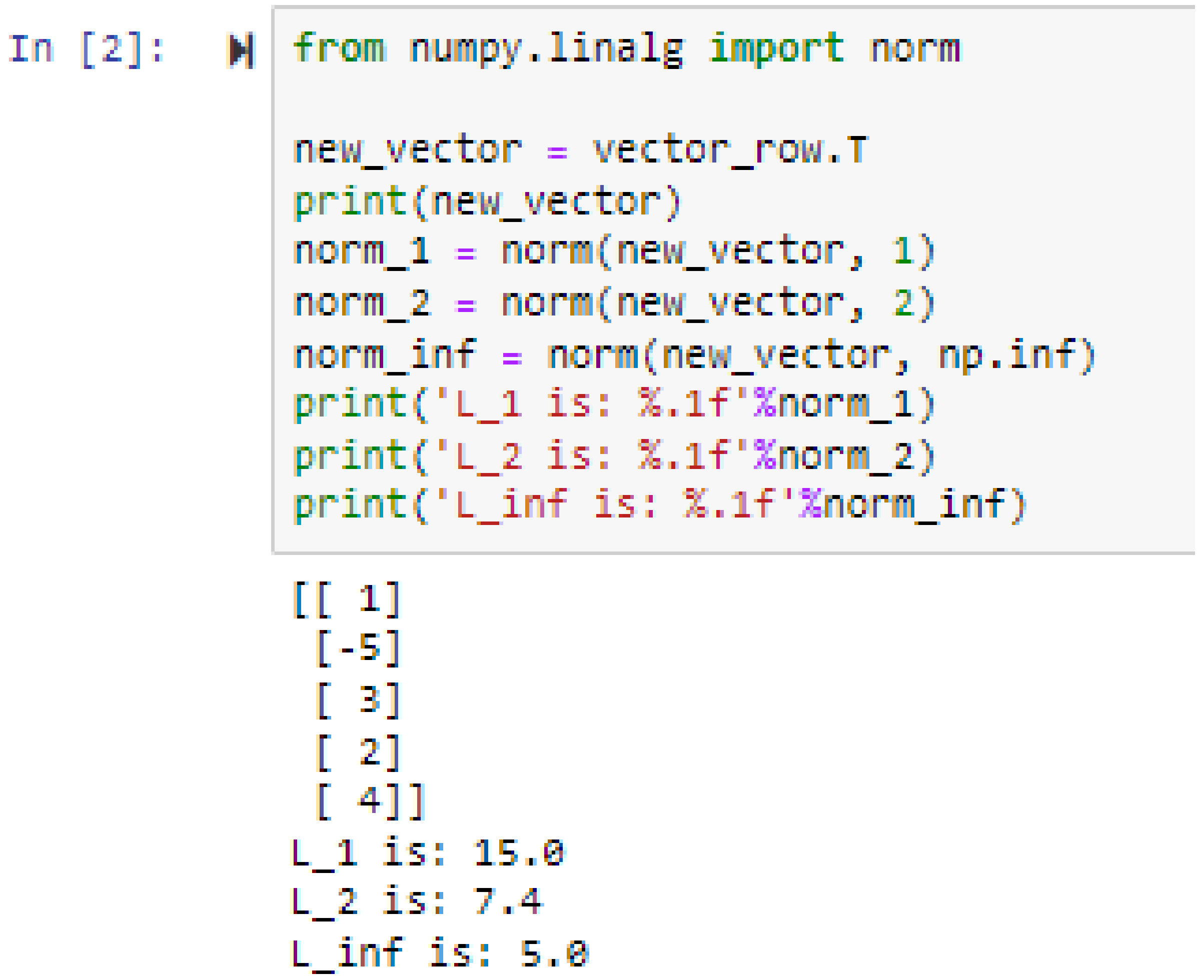
- The set $\mathbb{R}^n$ is the **set** of all n-tuples of **real** numbers. In set notation this is $\mathbb{R}^n = \{(x_1, x_2, \dots, x_n): x_1, x_2, \dots, x_n \in \mathbb{R}\}$.
- A **vector** in $\mathbb{R}^n$ is an n-tuple, or **point**, in $\mathbb{R}^n$. Vectors can be written **horizontally** (i.e., with the elements of the vector next to each other) in a **row vector**, or **vertically** (i.e., with the elements of the vector on top of each other) in a **column vector**. If the context of a vector is ambiguous, it usually means the vector is a **column** vector.
- The i-th element of a vector, v, is denoted by v_i. The **transpose** of a column vector is a row vector of the same length, and the **transpose** of a row vector is a column vector. In mathematics, the transpose is denoted by a superscript T, or v^T.
- The **zero vector** is the vector in $\mathbb{R}^n$ containing all **zeros**.

- The **norm** of a vector is a measure of its **length**. There are many ways of defining the length of a vector depending on the metric used (i.e., the **distance** formula chosen).
- The most common is called the $\boldsymbol{L_2}$ **norm**, which is computed according to the distance formula you are probably familiar with from grade school. The L_2 norm of a vector v is denoted by $\|v\|_2$ and $\|v\|_2 = \sqrt{\sum_i v_i^2}$. This is sometimes also called **Euclidian length** and refers to the "physical" length of a vector in one-, two-, or three-dimensional space.
- The $\boldsymbol{L_1}$ **norm**, or "**Manhattan Distance**," is computed as $\|v\|_1 = \sum_i |v_i|$, and is named after the grid-like road structure in **New York City**.
- In general, the **p-norm,** $\boldsymbol{L_p}$, of a vector is $\|v\|_p = \sqrt[p]{\left(\sum_i v_i^p\right)}$.
- The $\boldsymbol{L_\infty}$ **norm** is the p-norm, where $p = \infty$. The L_∞ norm is written as $\|v\|_\infty$and it is equal to the **maximum absolute** value in v.

Basics of Linear Algebra - Vectors



```
In [1]:  import numpy as np

         vector_row = np.array([[1, -5, 3, 2, 4]])
         vector_column = np.array([[1],
                                   [2],
                                   [3],
                                   [4]])
         print(vector_row.shape)
         print(vector_column.shape)

         (1, 5)
         (4, 1)
```



```
In [2]:  from numpy.linalg import norm

         new_vector = vector_row.T
         print(new_vector)
         norm_1 = norm(new_vector, 1)
         norm_2 = norm(new_vector, 2)
         norm_inf = norm(new_vector, np.inf)
         print('L_1 is: %.1f'%norm_1)
         print('L_2 is: %.1f'%norm_2)
         print('L_inf is: %.1f'%norm_inf)

         [[ 1]
          [-5]
          [ 3]
          [ 2]
          [ 4]]
         L_1 is: 15.0
         L_2 is: 7.4
         L_inf is: 5.0
```

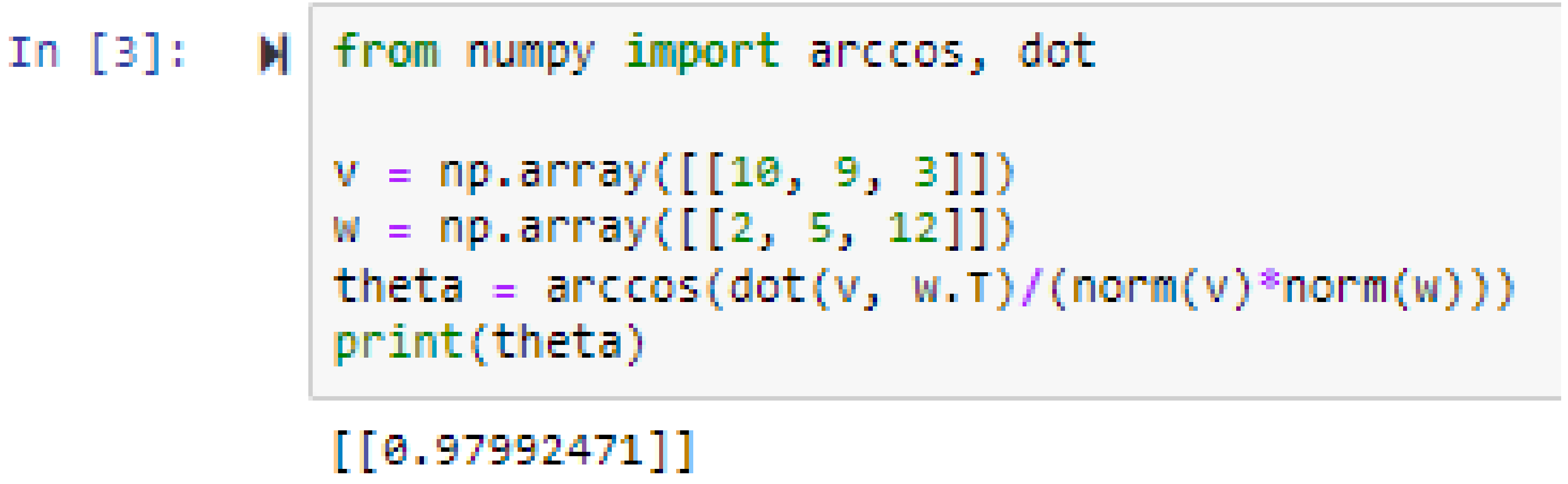
Basics of Linear Algebra - Vectors

- Vector **addition** is defined as the **pairwise addition** of each of the elements of the added vectors.
- Vector **multiplication** can be defined in several ways depending on the context. **Scalar multiplication** of a vector is the product of a **vector** and a **scalar** (i.e., a number in $\mathbb{R}$). Scalar multiplication is defined as the **product** of each element of the vector by the scalar.
- The **dot product** of two vectors is the **sum** of the **product** of the respective elements in each vector and is denoted by $\cdot$. Therefore for v and $w \in \mathbb{R}^n$, $d = v \cdot w$ is defined as $d = \sum_{i=1}^{n} v_i w_i$.
- The **angle** between two vectors, θ, is defined by the formula:

$$v \cdot w = \|v\|_2 \|w\|_2 \cos\theta$$

Basics of Linear Algebra - Vectors

- The **dot product** is a measure of how **similarly directed** the two vectors are. For example, the vectors (1,1) and (2,2) are **parallel**. If you compute the angle between them using the dot product, you will find that $\theta = 0$. If the angle between the vectors, $\theta = \pi/2$, then the vectors are said to be **perpendicular** or **orthogonal**, and the dot product is 0.



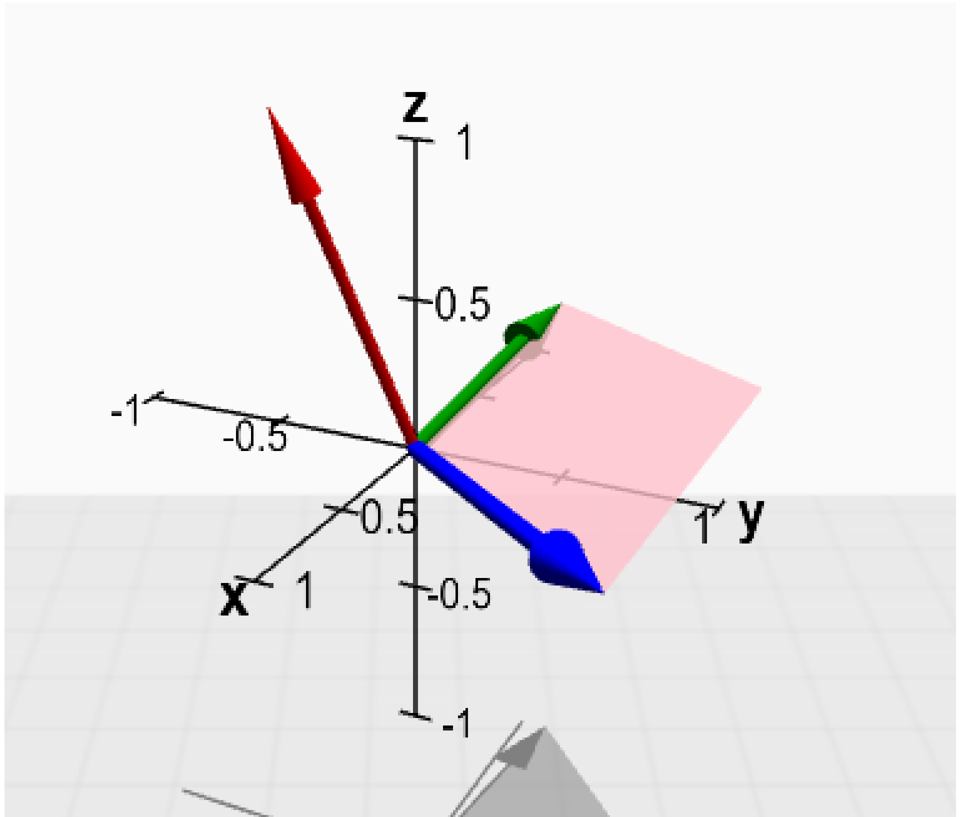
```
In [3]:  from numpy import arccos, dot

         v = np.array([[10, 9, 3]])
         w = np.array([[2, 5, 12]])
         theta = arccos(dot(v, w.T)/(norm(v)*norm(w)))
         print(theta)

         [[0.97992471]]
```

Basics of Linear Algebra - Vectors

- Finally, the **cross product** between two vectors, v and w, is written $v \times w$. It is defined by $v \times w = \|v\|_2 \|w\|_2 \sin\theta\, n$, where θ is the **angle** between the v and w (which can be computed from the dot product) and n is a vector perpendicular to both v and w with **unit** length (i.e., the length is one).

- The geometric interpretation of the cross product is a **vector** perpendicular to both v and w with length equal to the area enclosed by the parallelogram created by the two vectors.

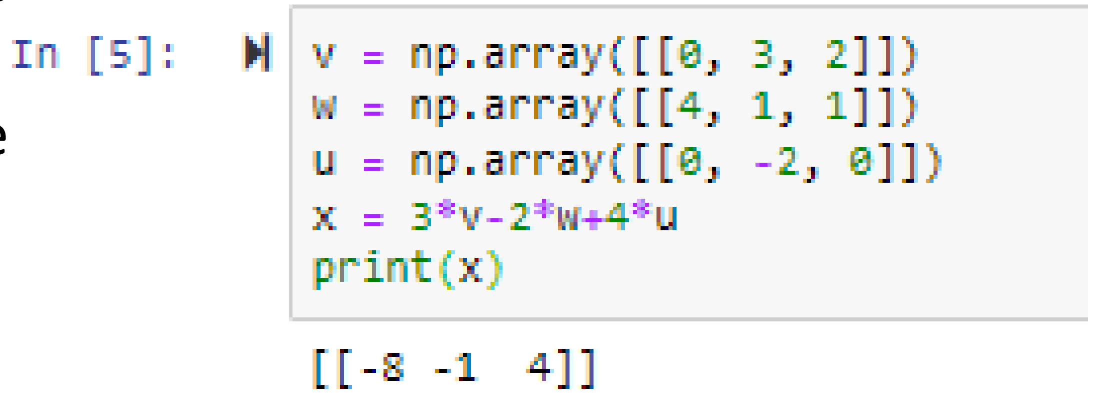
```
In [4]:  v = np.array([[0, 2, 0]])
         w = np.array([[3, 0, 0]])
         print(np.cross(v, w))

         [[ 0  0 -6]]
```



https://mathinsight.org/cross_product

Basics of Linear Algebra - Vectors

- Assuming that S is a set in which **addition** and **scalar multiplication** are defined, a **linear combination** of S is defined as $\sum \alpha_i s_i$, where α_i is any real number and s_i is the i-th object in S. Sometimes the α_i values are called the **coefficients** of s_i.
- **Linear combinations** can be used to describe numerous things. For example, a grocery bill can be written $\sum c_i n_i$, where c_i is the **cost** of item i and n_i is the **number** of item i purchased. Thus, the **total cost** is a linear combination of the items purchased.
- A set is called **linearly independent** if no object in the set can be written as a linear combination of the other objects in the set. We will only consider the linear independence of a set of vectors. A set of vectors that is not linearly independent is **linearly dependent**.



```
In [5]:  v = np.array([[0, 3, 2]])
         w = np.array([[4, 1, 1]])
         u = np.array([[0, -2, 0]])
         x = 3*v-2*w+4*u
         print(x)

         [[-8 -1  4]]
```

Basics of Linear Algebra - Matrices

- An $m \times n$ **matrix** is a rectangular table of numbers consisting of m rows and n columns. The **norm** of a matrix can be considered as a particular kind of **vector norm**, if we treat the $m \times n$ elements of M are the elements of an mn dimensional vector, then the p-norm of this vector can be write as:

$$\|M\|_p = \sqrt[p]{\left(\sum_i^m \sum_j^n |a_{ij}|^p\right)}$$

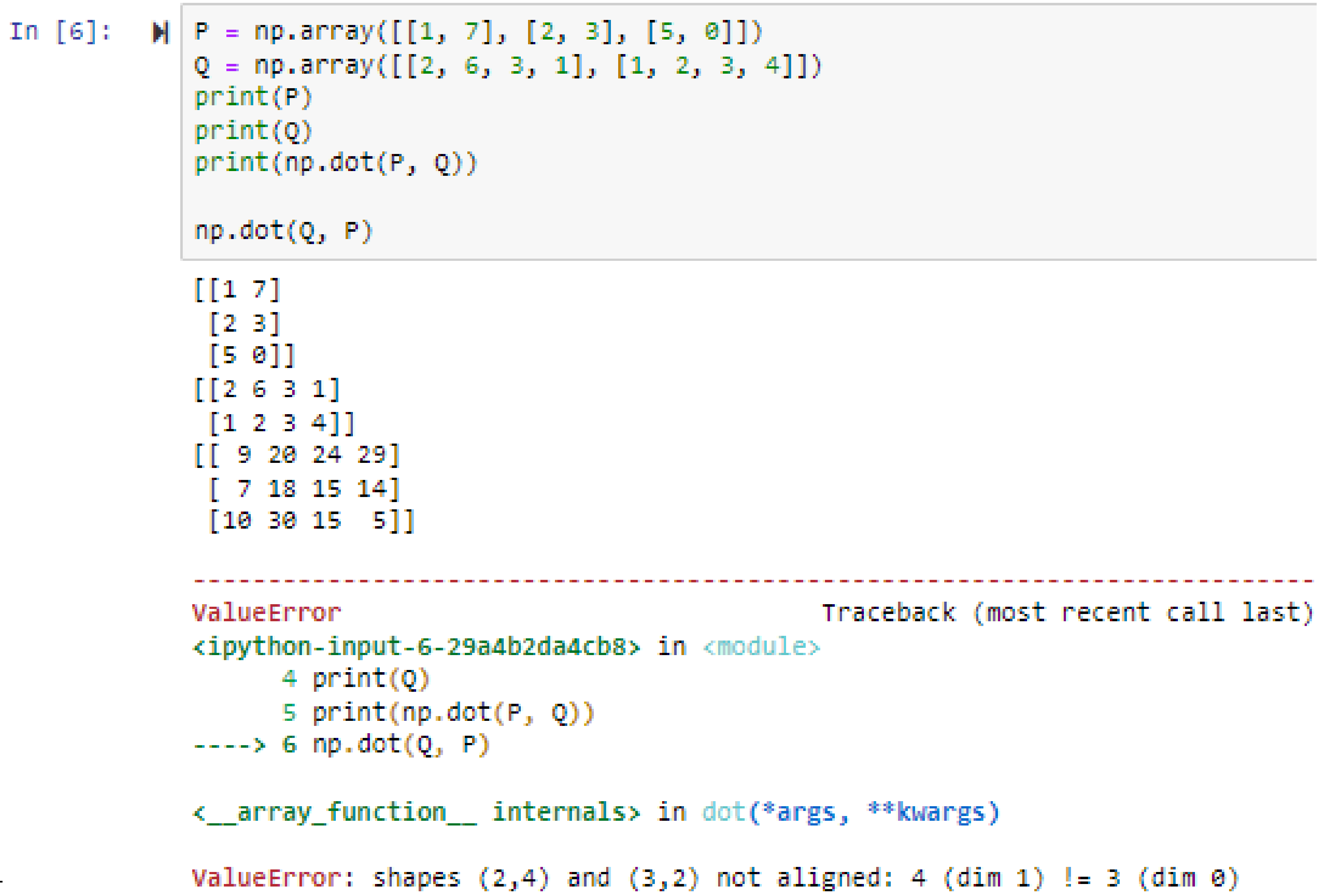
- You can calculate the matrix norm using the same **norm** function in Numpy as that for vector.

Basics of Linear Algebra - Matrices

- **Matrix multiplication** between two matrices, P and Q, is defined when P is an $m \times p$ matrix and Q is a $p \times n$ matrix. The result of $M = PQ$ is a matrix M that is $m \times n$. The dimension with size p is called the **inner matrix dimension**, and the inner matrix dimensions must match for matrix multiplication to be defined. The dimensions m and n are called the **outer matrix dimensions**. Formally, if P is $m \times p$ and Q is $p \times n$, then $M = PQ$ is defined as

$$M_{ij} = \sum_{k=1}^{p} P_{ik} Q_{kj}$$

- The **product** of two matrices P and Q in Python is achieved by using the **dot** method in Numpy.
- The **transpose** of a matrix is a reversal of its rows with its columns. The transpose is denoted by a superscript, T, such as M^T is the **transpose** of matrix M.



```
In [6]:  P = np.array([[1, 7], [2, 3], [5, 0]])
         Q = np.array([[2, 6, 3, 1], [1, 2, 3, 4]])
         print(P)
         print(Q)
         print(np.dot(P, Q))

         np.dot(Q, P)
```

```
[[1 7]
 [2 3]
 [5 0]]
[[2 6 3 1]
 [1 2 3 4]]
[[ 9 20 24 29]
 [ 7 18 15 14]
 [10 30 15  5]]

---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-6-29a4b2da4cb8> in <module>
      4 print(Q)
      5 print(np.dot(P, Q))
----> 6 np.dot(Q, P)

<__array_function__ internals> in dot(*args, **kwargs)

ValueError: shapes (2,4) and (3,2) not aligned: 4 (dim 1) != 3 (dim 0)
```

Basics of Linear Algebra - Matrices

- A **square matrix** is an $n \times n$ matrix; that is, it has the same number of rows as columns.
- The **determinant** is an important property of **square matrices**. The determinant is denoted by $\boldsymbol{det(M)}$, both in Mathematics and in Numpy's **linalg** package, sometimes it is also denoted as $|M|$.
- In the case of a 2×2 matrix, the determinant is: $|M| = \begin{bmatrix} a & b \\ c & d \end{bmatrix} = ad - bc$
- Similarly, in the case of a 3×3 matrix, the determinant is:

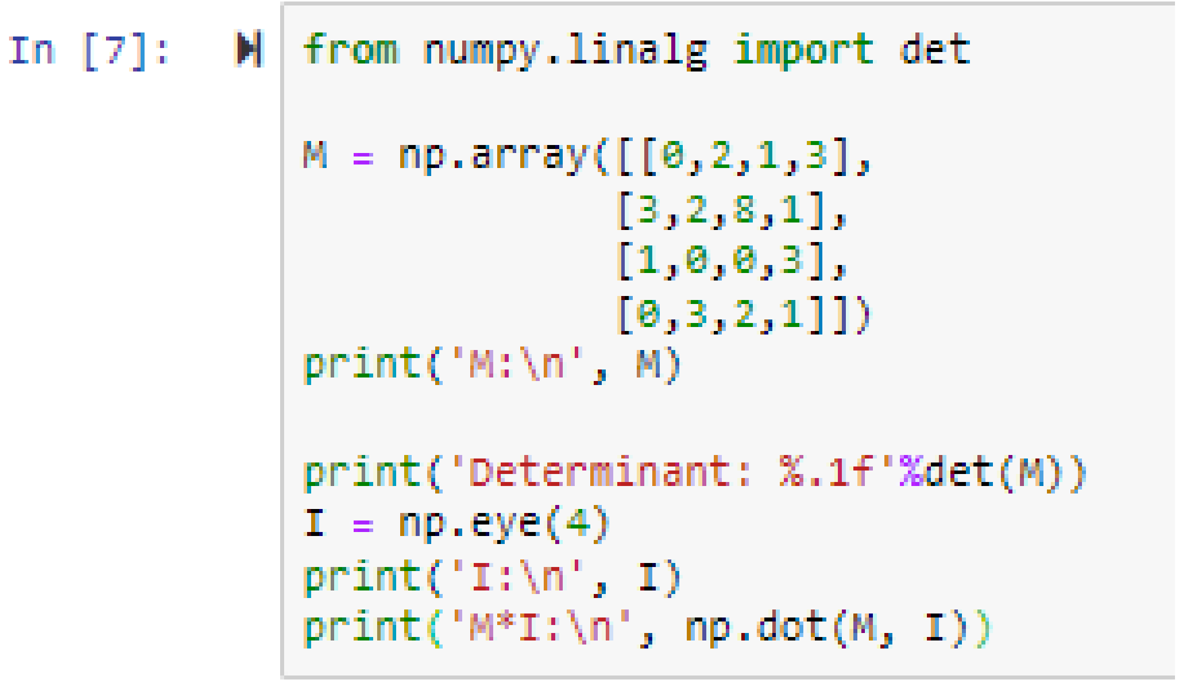
$$|M| = \begin{bmatrix} a & b & c \\ d & e & f \\ g & h & i \end{bmatrix} = a \begin{bmatrix} \square & \square & \square \\ \square & e & f \\ \square & h & i \end{bmatrix} - b \begin{bmatrix} \square & \square & \square \\ d & \square & f \\ g & \square & i \end{bmatrix} + c \begin{bmatrix} \square & \square & \square \\ d & e & \square \\ g & h & \square \end{bmatrix}$$

$$= a \begin{bmatrix} e & f \\ h & i \end{bmatrix} - b \begin{bmatrix} d & f \\ g & i \end{bmatrix} + c \begin{bmatrix} d & e \\ g & h \end{bmatrix}$$

$$= aei + bfg + cdh - ceg - bdi - afh$$

Basics of Linear Algebra - Matrices

- The **identity matrix** is a **square matrix** with **ones** on the diagonal and **zeros** elsewhere.
- The **identity matrix** is usually denoted by I, and is analogous to the real number identity, 1. That is, multiplying any matrix by I (of compatible size) will produce the same matrix.



```
In [7]:   from numpy.linalg import det

          M = np.array([[0,2,1,3],
                        [3,2,8,1],
                        [1,0,0,3],
                        [0,3,2,1]])
          print('M:\n', M)

          print('Determinant: %.1f'%det(M))
          I = np.eye(4)
          print('I:\n', I)
          print('M*I:\n', np.dot(M, I))
```

```
M:
 [[0 2 1 3]
 [3 2 8 1]
 [1 0 0 3]
 [0 3 2 1]]
Determinant: -38.0
I:
 [[1. 0. 0. 0.]
 [0. 1. 0. 0.]
 [0. 0. 1. 0.]
 [0. 0. 0. 1.]]
M*I:
 [[0. 2. 1. 3.]
 [3. 2. 8. 1.]
 [1. 0. 0. 3.]
 [0. 3. 2. 1.]]
```

Basics of Linear Algebra - Matrices

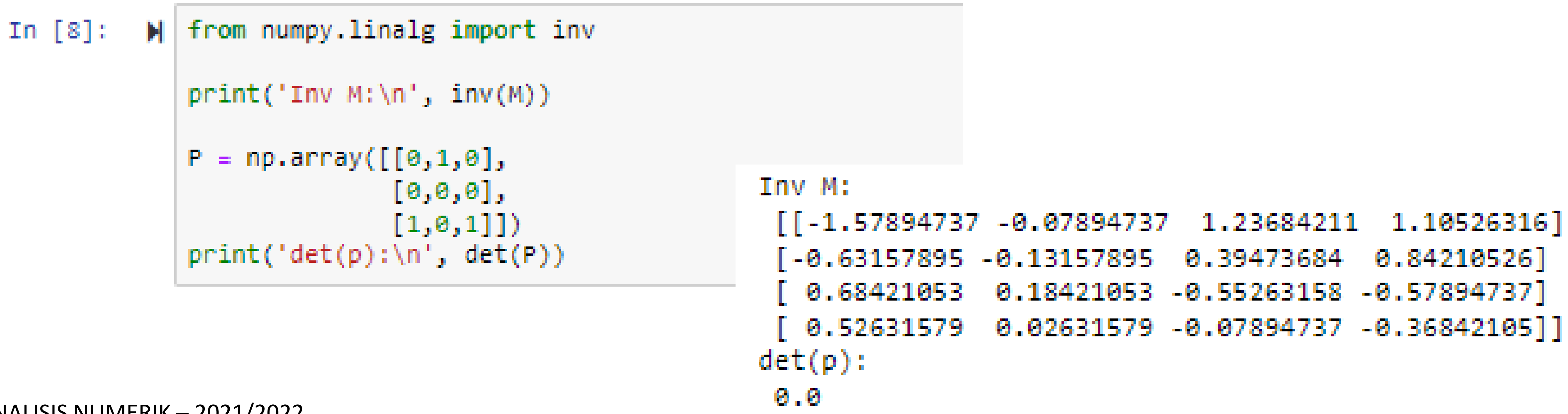
- The **inverse** of a **square matrix** M is a matrix of the same size, N, such that $M \cdot N = I$.
- A matrix is said to be **invertible** if it has an **inverse**. The inverse of a matrix is **unique**; that is, for an invertible matrix, there is only one inverse for that matrix. If M is a square matrix, its inverse is denoted by M^{-1} in Mathematics, and it can be computed in Python using the function **inv** from Numpy's **linalg** package.
- For a 2×2 matrix, the analytic solution of the matrix inverse is:

$$M^{-1} = \begin{bmatrix} a & b \\ c & d \end{bmatrix}^{-1} = \frac{1}{|M|} \begin{bmatrix} d & -b \\ -c & a \end{bmatrix}$$

- The calculation of the matrix inverse for the analytic solution becomes complicated with increasing matrix **dimension**, there are many other methods can make things easier, such as **Gaussian elimination**, **Newton's** method, **Eigen decomposition,** and so on.

Basics of Linear Algebra - Matrices

- Recall that 0 has **no inverse** for multiplication in the real-numbers setting. Similarly, there are matrices that do not have inverses. These matrices are called **singular**. Matrices that do have an inverse are called **nonsingular**.
- One way to determine if a matrix is **singular** is by computing its **determinant**. If the **determinant** is 0, then the matrix is **singular**; if not, the matrix is **nonsingular**.



```
In [8]:  from numpy.linalg import inv

         print('Inv M:\n', inv(M))

         P = np.array([[0,1,0],
                       [0,0,0],
                       [1,0,1]])
         print('det(p):\n', det(P))
```

```
Inv M:
 [[-1.57894737 -0.07894737  1.23684211  1.10526316]
 [-0.63157895 -0.13157895  0.39473684  0.84210526]
 [ 0.68421053  0.18421053 -0.55263158 -0.57894737]
 [ 0.52631579  0.02631579 -0.07894737 -0.36842105]]
det(p):
 0.0
```

Basics of Linear Algebra - Matrices

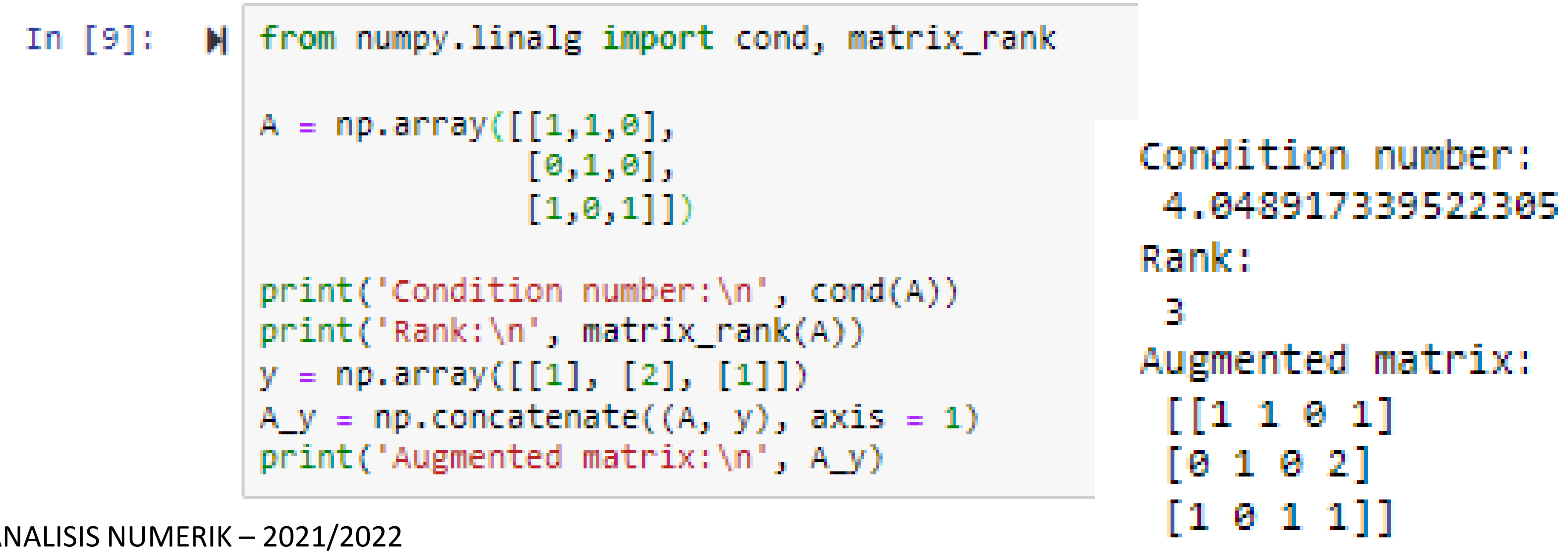
- A matrix that is **close** to being **singular** (i.e., the **determinant** is close to 0) is called **ill-conditioned**. Although **ill-conditioned** matrices have **inverses**, they are problematic numerically in the same way that dividing a number by a very, very small number is problematic. That is, it can result in computations that result in **overflow**, **underflow**, or numbers small enough to result in significant **round-off errors**.

- The **condition number** is a measure of how ill-conditioned a matrix is, and it can be computed using Numpy's function **cond** from **linalg**. The **higher** the condition number, the **closer** the matrix is to being **singular**.

- The **rank** of an $m \times n$ matrix A is the number of **linearly independent columns or rows** of A, and is denoted by $rank(A)$. It can be shown that the number of linearly independent **rows** is always equal to the number of linearly independent **columns** for any matrix. A matrix is called **full rank** if $rank(A) = \min(m, n)$. The matrix, A, is also **full rank** if all of its columns are **linearly independent**.

Basics of Linear Algebra - Matrices

- An **augmented matrix** is a matrix, A, concatenated with a vector, y, and is written $[A, y]$. This is commonly read "A augmented with y." You can use **np.concatenate** to **concatenate** them.
- If $rank([A, y]) = rank(A) + 1$, then the vector, y, is "**new**" information. That is, it cannot be created as a linear combination of the columns in A.



```
In [9]:  from numpy.linalg import cond, matrix_rank

         A = np.array([[1,1,0],
                       [0,1,0],
                       [1,0,1]])

         print('Condition number:\n', cond(A))
         print('Rank:\n', matrix_rank(A))
         y = np.array([[1], [2], [1]])
         A_y = np.concatenate((A, y), axis = 1)
         print('Augmented matrix:\n', A_y)
```

```
Condition number:
 4.048917339522305
Rank:
 3
Augmented matrix:
 [[1 1 0 1]
 [0 1 0 2]
 [1 0 1 1]]
```

Linear Transformations

- For vectors x and y, and scalars a and b, it is sufficient to say that a function, F, is a **linear transformation** if

$$F(ax + by) = aF(x) + bF(y)$$

- It can be shown that multiplying an $m \times n$ matrix, A, and an $n \times 1$ vector, v, of compatible size is a **linear transformation** of v.
- Therefore from this point forward, a matrix will be synonymous with a **linear transformation function**.

Linear Transformations

- If A is an $m \times n$ matrix, then there are two important **subspaces** associated with A, one is $\mathbb{R}^n$, the other is $\mathbb{R}^m$.
- The **domain** of A is a subspace of $\mathbb{R}^n$. It is the set of all vectors that can be multiplied by A on the right.
- The **range** of A is a subspace of $\mathbb{R}^m$. It is the set of all vectors y such that $y = Ax$. It can be denoted as $\mathcal{R}(A)$, where $\mathcal{R}(A) = \{y \in \mathbb{R}^m : Ax = y\}$. Another way to think about the **range** of A is the set of all linear combinations of the columns in A, where x_i is the coefficient of the i-th column in A.
- The **null space** of A, defined as $N(A) = \{x \in \mathbb{R}^n : Ax = 0_m\}$, is the subset of vectors in the domain of A, x, such that $Ax = 0_m$, where 0_m is the **zero vector** (i.e., a vector in $\mathbb{R}^m$ with all zeros).

Systems of Linear Equations

- A **linear equation** is an equality of the form $\sum_{i=1}^{n}(a_i x_i) = y$, where a_i are scalars, x_i are unknown variables in $\mathbb{R}$, and y is a scalar.
- **Examples**: Determine which of the following equations is linear and which is not. For the ones that are not linear, can you manipulate them so that they are?

1. $3x_1 + 4x_2 - 3 = -5x_3$

2. $\frac{-x_1+x_2}{x_3} = 2$

3. $x_1 x_2 + x_3 = 5$

- Equation 1 can be rearranged to be $3x_1 + 4x_2 + 5x_3 = 3$, which clearly has the form of a linear equation. Equation 2 is not linear but can be rearranged to be $-x_1 + x_2 - 2x_3 = 0$, which is linear. Equation 3 is not linear.

Systems of Linear Equations

- A **system of linear equations** is a set of **linear equations** that share the same variables.
- Consider the following **system of linear equations** where $a_{i,j}$ and y_i are real numbers:

$$\begin{array}{ccccccccccc}
a_{1,1}x_1 & + & a_{1,2}x_2 & + & \dots & + & a_{1,n-1}x_{n-1} & + & a_{1,n}x_n & = & y_1, \\
a_{2,1}x_1 & + & a_{2,2}x_2 & + & \dots & + & a_{2,n-1}x_{n-1} & + & a_{2,n}x_n & = & y_2, \\
 & & & & \dots & & \dots & & & & \\
a_{m-1,1}x_1 & + & a_{m-1,2}x_2 & + & \dots & + & a_{m-1,n-1}x_{n-1} & + & a_{m-1,n}x_n & = & y_{m-1}, \\
a_{m,1}x_1 & + & a_{m,2}x_2 & + & \dots & + & a_{m,n-1}x_{n-1} & + & a_{m,n}x_n & = & y_m.
\end{array}$$

- The **matrix form** of a system of linear equations is $Ax = y$, where A is a $m \times n$ matrix, $A(i,j) = a_{i,j}$, y is a vector in $\mathbb{R}^m$, and x is an **unknown** vector in $\mathbb{R}^n$.

$$\begin{bmatrix} a_{1,1} & a_{1,2} & \dots & a_{1,n} \\ a_{2,1} & a_{2,2} & \dots & a_{2,n} \\ \dots & \dots & \dots & \dots \\ a_{m,1} & a_{m,2} & \dots & a_{m,n} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ \dots \\ x_n \end{bmatrix} = \begin{bmatrix} y_1 \\ y_2 \\ \dots \\ y_m \end{bmatrix}$$

Solutions to Systems of Linear Equations

- Consider a system of linear equations in matrix form, $Ax = y$, where A is an $m \times n$ matrix. Recall that this means there are m equations and n unknowns in our system. A **solution** to a system of linear equations is an x in $\mathbb{R}^n$ that satisfies the matrix form equation. Depending on the values that populate A and y, there are **three** distinct solution possibilities for x.
- **Case 1**: There is **no solution** for x. If $rank([A, y]) = rank(A) + 1$, then y is **linearly independent** from the columns of A. Therefore y is not in the range of A and by definition, there cannot be an x that satisfies the equation.
- **Case 2**: There is a **unique solution** for x. If $rank([A, y]) = rank(A)$, then y can be written as a **linear combination** of the columns of A and there is at least **one solution** for the matrix equation. For there to be only one solution, $rank(A) = n$ must also be true. In other words, the number of equations must be exactly equal to the number of unknowns.

Solutions to Systems of Linear Equations

- **Case 3**: There is an **infinite number of solutions** for x. If $rank([A, y]) = rank(A)$, then y is in the range of A, and there is at least one solution for the matrix equation. However, if $rank(A) < n$, then there is an **infinite number of solutions**.
- We will only discuss how we solve a systems of equations when it has **unique solution**.
- Let's say we have n equations with n variables, $Ax = y$, as shown in the following:

$$\begin{bmatrix} a_{1,1} & a_{1,2} & \dots & a_{1,n} \\ a_{2,1} & a_{2,2} & \dots & a_{2,n} \\ \dots & \dots & \dots & \dots \\ a_{n,1} & a_{n,2} & \dots & a_{n,n} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ \dots \\ x_n \end{bmatrix} = \begin{bmatrix} y_1 \\ y_2 \\ \dots \\ y_n \end{bmatrix}$$

Gauss Elimination Method

- The **Gauss Elimination** method is a procedure to turn matrix A into an **upper triangular** form to solve the system of equations. Let's use a system of 4 equations and 4 variables to illustrate the idea. The Gauss Elimination essentially turning the system of equations to:

$$\begin{bmatrix} a_{1,1} & a_{1,2} & a_{1,3} & a_{1,4} \\ 0 & a'_{2,2} & a'_{2,3} & a'_{2,4} \\ 0 & 0 & a'_{3,3} & a'_{3,4} \\ 0 & 0 & 0 & a'_{4,4} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix} = \begin{bmatrix} y_1 \\ y'_2 \\ y'_3 \\ y'_4 \end{bmatrix}$$

- By turning the matrix form into this, we can see the equations turn into:

$$\begin{array}{rcrcrcrcl} a_{1,1}x_1 & + & a_{1,2}x_2 & + & a_{1,3}x_3 & + & a_{1,4}x_4 & = & y_1, \\ & & a'_{2,2}x_2 & + & a'_{2,3}x_3 & + & a'_{2,4}x_4 & = & y'_2 \\ & & & & a'_{3,3}x_3 & + & a'_{3,4}x_4 & = & y'_3, \\ & & & & & & a'_{4,4}x_4 & = & y'_4. \end{array}$$

Gauss Elimination Method

- We can see by turning into this form, x_4 can be easily solved by dividing both sides with $a'_{4,4}$, then we can back substitute it into the 3rd equation to solve x_3. With x_3 and x_4, we can substitute them to the 2nd equation to solve x_2. Finally, we can get all the solution for x.
- We solve the system of equations from **bottom-up**, this is called **backward substitution**. Note that, if A is a **lower triangular** matrix, we would solve the system from **top-down** by **forward substitution**.
- **Example**: Use Gauss Elimination to solve the following equations.

$$4x_1 + 3x_2 - 5x_3 = 2$$
$$-2x_1 - 4x_2 + 5x_3 = 5$$
$$8x_1 + 8x_2 = -3$$

Gauss Elimination Method

- Step 1: Turn these equations to matrix form $Ax = y$.

$$\begin{bmatrix} 4 & 3 & -5 \\ -2 & -4 & 5 \\ 8 & 8 & 0 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 2 \\ 5 \\ -3 \end{bmatrix}$$

- Step 2: Get the augmented matrix $[A, y]$

$$[A, y] = \begin{bmatrix} 4 & 3 & -5 & 2 \\ -2 & -4 & 5 & 5 \\ 8 & 8 & 0 & -3 \end{bmatrix}$$

- Step 3: Now we start to eliminate the elements in the matrix, we do this by choose a **pivot equation**, which is used to eliminate the elements in other equations. Let's choose the first equation as the pivot equation and turn the 2nd row first element to 0. To do this, we can multiply -0.5 for the 1st row (pivot equation) and subtract it from the 2nd row. The multiplier is $m_{2,1} = -0.5$. We will get

$$\begin{bmatrix} 4 & 3 & -5 & 2 \\ 0 & -2.5 & 2.5 & 6 \\ 8 & 8 & 0 & -3 \end{bmatrix}$$

- Step 4: Turn the 3rd row first element to 0. We can do something similar, multiply 2 to the 1st row and subtract it from the 3rd row. The multiplier is $m_{3,1} = 2$. We will get

$$\begin{bmatrix} 4 & 3 & -5 & 2 \\ 0 & -2.5 & 2.5 & 6 \\ 0 & 2 & 10 & -7 \end{bmatrix}$$

- Step 5: Turn the 3rd row 2nd element to 0. We can multiple -4/5 for the 2nd row, and add subtract it from the 3rd row. The multiplier is $m_{3.2} = -0.8$. We will get

$$\begin{bmatrix} 4 & 3 & -5 & 2 \\ 0 & -2.5 & 2.5 & 6 \\ 0 & 0 & 12 & -2.2 \end{bmatrix}$$

- Step 6: Therefore, we can get $\boldsymbol{x_3} = -\frac{\mathbf{2.2}}{\mathbf{12}} = \mathbf{-0.183}$.
- Step 7: Insert x_3 to the 2nd equation, we get $\boldsymbol{x_2} = \mathbf{-2.583}$.
- Step 8: Insert x_2 and x_3 to the first equation, we have $\boldsymbol{x_1} = \mathbf{2.208}$.
- **Note!** Sometimes you will have the first element in the 1st row is 0, just switch the first row with a non-zero first element row, then you can do the same procedure as above.

Gauss-Jordan Elimination Method

- **Gauss-Jordan Elimination** solves the systems of equations using a procedure to turn A into a **diagonal** form, such that the matrix form of the equations becomes

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix} = \begin{bmatrix} y_1' \\ y_2' \\ y_3' \\ y_4' \end{bmatrix}$$

- Essentially, the equations become:

$$\begin{array}{ccccccccc} x_1 & + & 0 & + & 0 & + & 0 & = & y_1', \\ 0 & + & x_2 & + & 0 & + & 0 & = & y_2' \\ 0 & + & 0 & + & x_3 & + & 0 & = & y_3', \\ 0 & + & 0 & + & 0 & + & x_4 & = & y_4'. \end{array}$$

- **Example**: Use Gauss-Jordan Elimination to solve the following equations.

$$4x_1 + 3x_2 - 5x_3 = 2$$
$$-2x_1 - 4x_2 + 5x_3 = 5$$
$$8x_1 + 8x_2 = -3$$

- Step 1: Get the augmented matrix $[A, y]$

$$[A, y] = \begin{bmatrix} 4 & 3 & -5 & 2 \\ -2 & -4 & 5 & 5 \\ 8 & 8 & 0 & -3 \end{bmatrix}$$

- Step 2: Get the first element in 1st row to 1, we divide 4 to the row:

$$\begin{bmatrix} 1 & 3/4 & -5/4 & 1/2 \\ -2 & -4 & 5 & 5 \\ 8 & 8 & 0 & -3 \end{bmatrix}$$

- Step 3: Eliminate the first element in 2nd and 3rd rows, we multiply -2 and 8 to the 1st row and subtract it from the 2nd and 3rd rows.

$$\begin{bmatrix} 1 & 3/4 & -5/4 & 1/2 \\ 0 & -5/2 & 5/2 & 6 \\ 0 & 2 & 10 & -7 \end{bmatrix}$$

- Step 4: Normalize the 2nd element in 2nd row to 1, we divide -5/2 to achieve this.

$$\begin{bmatrix} 1 & 3/4 & -5/4 & 1/2 \\ 0 & 1 & -1 & -12/5 \\ 0 & 2 & 10 & -7 \end{bmatrix}$$

- Step 5: Eliminate the 2nd element of the 3rd row, we multiply 2 to the 2nd row and subtract it from the 3rd row.

$$\begin{bmatrix} 1 & 3/4 & -5/4 & 1/2 \\ 0 & 1 & -1 & -12/5 \\ 0 & 0 & 12 & -11/5 \end{bmatrix}$$

- Step 6: Normalize the last row by divide 12.

$$\begin{bmatrix} 1 & 3/4 & -5/4 & 1/2 \\ 0 & 1 & -1 & -12/5 \\ 0 & 0 & 1 & -11/60 \end{bmatrix}$$

- Step 7: Eliminate the 3rd element in 2nd row by multiply -1 to the 3rd row and subtract it from the 2nd row.

$$\begin{bmatrix} 1 & 3/4 & -5/4 & 1/2 \\ 0 & 1 & 0 & -155/60 \\ 0 & 0 & 1 & -11/60 \end{bmatrix}$$

- Step 8: Eliminate the 3rd element in 1st row by multiply -5/4 to the 3rd row and subtract it from the 1st row.

$$\begin{bmatrix} 1 & 3/4 & 0 & 13/48 \\ 0 & 1 & 0 & -2.583 \\ 0 & 0 & 1 & -0.183 \end{bmatrix}$$

- Step 9: Eliminate the 2nd element in 1st row by multiply 3/4 to the 2nd row and subtract it from the 1st row.

$$\begin{bmatrix} 1 & 0 & 0 & 2.208 \\ 0 & 1 & 0 & -2.583 \\ 0 & 0 & 1 & -0.183 \end{bmatrix}$$

LU Decomposition Method

- We see the previous two methods that involves of changing both A and y at the same time when trying to turn A to an **upper triangular** or **diagonal** matrix form. It involves **many operations**.
- But sometimes, we may have **same set of equations** but different sets of y for **different experiments**. This is actually quite common in the real-world, that we have different experiment observations $y_a, y_b, y_c, \dots$. Therefore, we have to solve $Ax = y_a$, $Ax = y_b$, … many times, since every time the $[A, y]$ will change. This is really inefficient, is there a method that we only change the left side of A but not the right hand y?
- The **LU decomposition** method is one of the solution that we only change the matrix A instead of y. It has the **advantages** for solving the systems that have the **same coefficient matrices** A but **different constant vectors** y.

- The LU decomposition method **aims** to turn A into the **multiply** of **two matrices** L and U, where L is a **lower triangular** matrix, while U is an **upper triangular** matrix. With this decomposition, we convert the system of equations to the following form:

$$LUx = y \rightarrow \begin{bmatrix} l_{1,1} & 0 & 0 & 0 \\ l_{2,1} & l_{2,2} & 0 & 0 \\ l_{3,1} & l_{3,2} & l_{3,3} & 0 \\ l_{4,1} & l_{4,2} & l_{4,3} & l_{4,4} \end{bmatrix} \begin{bmatrix} u_{1,1} & u_{1,2} & u_{1,3} & u_{1,4} \\ 0 & u_{2,2} & u_{2,3} & u_{2,4} \\ 0 & 0 & u_{3,3} & u_{3,4} \\ 0 & 0 & 0 & u_{4,4} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix} = \begin{bmatrix} y_1 \\ y_2 \\ y_3 \\ y_4 \end{bmatrix}$$

- If we define $Ux = M$, then the above equations become:

$$\begin{bmatrix} l_{1,1} & 0 & 0 & 0 \\ l_{2,1} & l_{2,2} & 0 & 0 \\ l_{3,1} & l_{3,2} & l_{3,3} & 0 \\ l_{4,1} & l_{4,2} & l_{4,3} & l_{4,4} \end{bmatrix} M = \begin{bmatrix} y_1 \\ y_2 \\ y_3 \\ y_4 \end{bmatrix}$$

- We can easily solve the above problem by **forward substitution**.

- After we solve M, we can easily solve the rest of the problem using **backward substitution**:

$$\begin{bmatrix} u_{1,1} & u_{1,2} & u_{1,3} & u_{1,4} \\ 0 & u_{2,2} & u_{2,3} & u_{2,4} \\ 0 & 0 & u_{3,3} & u_{3,4} \\ 0 & 0 & 0 & u_{4,4} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix} = \begin{bmatrix} m_1 \\ m_2 \\ m_3 \\ m_4 \end{bmatrix}$$

LU Decomposition Method

- But how can we calculate and get the L and U matrices? There are different ways to get the LU decomposition, let's just look one way using the **Gauss Elimination** method.
- From the above, we know that we get an **upper triangular** matrix after we conduct the Gauss Elimination. But at the same time, we actually also get the **lower triangular** matrix, it is just we never explicitly write it out. During the Gauss Elimination procedure, the matrix A actually turns into the **multiplication** of **two matrices** as shown below. With the right upper triangular form is the one we get before, but the lower triangular matrix has the diagonal are 1, and the **multipliers** that multiply the **pivot equation** to eliminate the elements during the procedure as the elements below the diagonal.

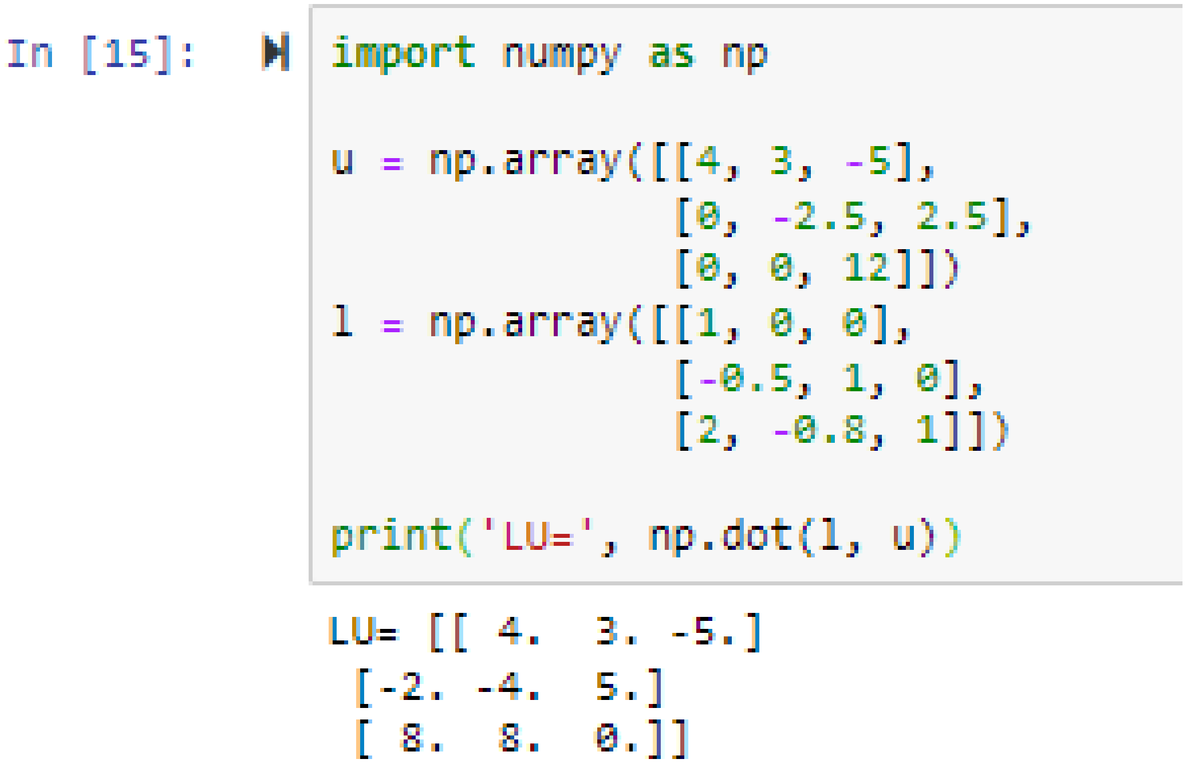
$$A = \begin{bmatrix} 1 & 0 & 0 & 0 \\ m_{2,1} & 1 & 0 & 0 \\ m_{3,1} & m_{3,2} & 1 & 0 \\ m_{4,1} & m_{4,2} & m_{4,3} & 1 \end{bmatrix} \begin{bmatrix} u_{1,1} & u_{1,2} & u_{1,3} & u_{1,4} \\ 0 & u_{2,2} & u_{2,3} & u_{2,4} \\ 0 & 0 & u_{3,3} & u_{3,4} \\ 0 & 0 & 0 & u_{4,4} \end{bmatrix}$$

LU Decomposition Method

- We can see that, we actually can get both L and U at the same time when we do **Gauss Elimination**. Let’s see the previous example, where U is the one we used before to solve the equations, and L is composed of the **multipliers** (you can check the examples in the Gauss Elimination section).

$$L = \begin{bmatrix} 1 & 0 & 0 \\ -0.5 & 1 & 0 \\ 2 & -0.8 & 1 \end{bmatrix}$$

$$U = \begin{bmatrix} 4 & 3 & -5 \\ 0 & -2.5 & 2.5 \\ 0 & 0 & 12 \end{bmatrix}$$



```
In [15]:  import numpy as np

          u = np.array([[4, 3, -5],
                        [0, -2.5, 2.5],
                        [0, 0, 12]])
          l = np.array([[1, 0, 0],
                        [-0.5, 1, 0],
                        [2, -0.8, 1]])

          print('LU=', np.dot(l, u))

          LU= [[ 4.   3.  -5.]
           [-2.  -4.   5.]
           [ 8.   8.   0.]]
```

Iterative Method

- The above methods we introduced are all **direct** methods, in which we compute the solution with a **finite number of operations**.
- In this section, we will introduce a different class of methods, the **iterative methods**, or **indirect methods**. It starts with an **initial guess** of the solution and then **repeatedly improve the solution** until the change of the solution is below a **threshold**. In order to use this iterative process, we need first write the **explicit form** of a system of equations. If we have a system of linear equations:

$$\begin{bmatrix} a_{1,1} & a_{1,2} & \cdots & a_{1,n} \\ a_{2,1} & a_{2,2} & \cdots & a_{2,n} \\ \cdots & \cdots & \cdots & \cdots \\ a_{m,1} & a_{m,2} & \cdots & a_{m,n} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ \cdots \\ x_n \end{bmatrix} = \begin{bmatrix} y_1 \\ y_2 \\ \cdots \\ y_m \end{bmatrix}$$

- We can write its **explicit form** as:

$$x_i = \frac{1}{a_{i,i}} \left[y_i - \sum_{j=1, j \neq i}^{j=n} a_{i,j} x_j \right]$$

Iterative Method

- This is the basics of the iterative methods, we can assume **initial values** for all the x, and use it as $x^{(0)}$. In the first iteration, we can substitute $x^{(0)}$ into the right-hand side of the explicit equation above, and get the first iteration solution $x^{(1)}$. Thus, we can substitute $x^{(1)}$ into the equation and get substitute $x^{(2)}$. The iterations continue until the difference between $x^{(k)}$ and $x^{(k-1)}$ is smaller than some pre-defined value.
- In order to have the iterative methods work, we do need **specific condition** for the solution to **converge**. A sufficient but not necessary condition of the convergence is the **coefficient matrix** a is a **diagonally dominant**. This means that in each row of the matrix of coefficients a, the **absolute value** of the **diagonal element** is **greater** than the sum of the absolute values of the **off-diagonal elements**. If the coefficient matrix satisfy the condition, the iteration will converge to the solution. The solution might still converge even when this condition is not satisfied.

Iterative Method – Gauss-Seidel

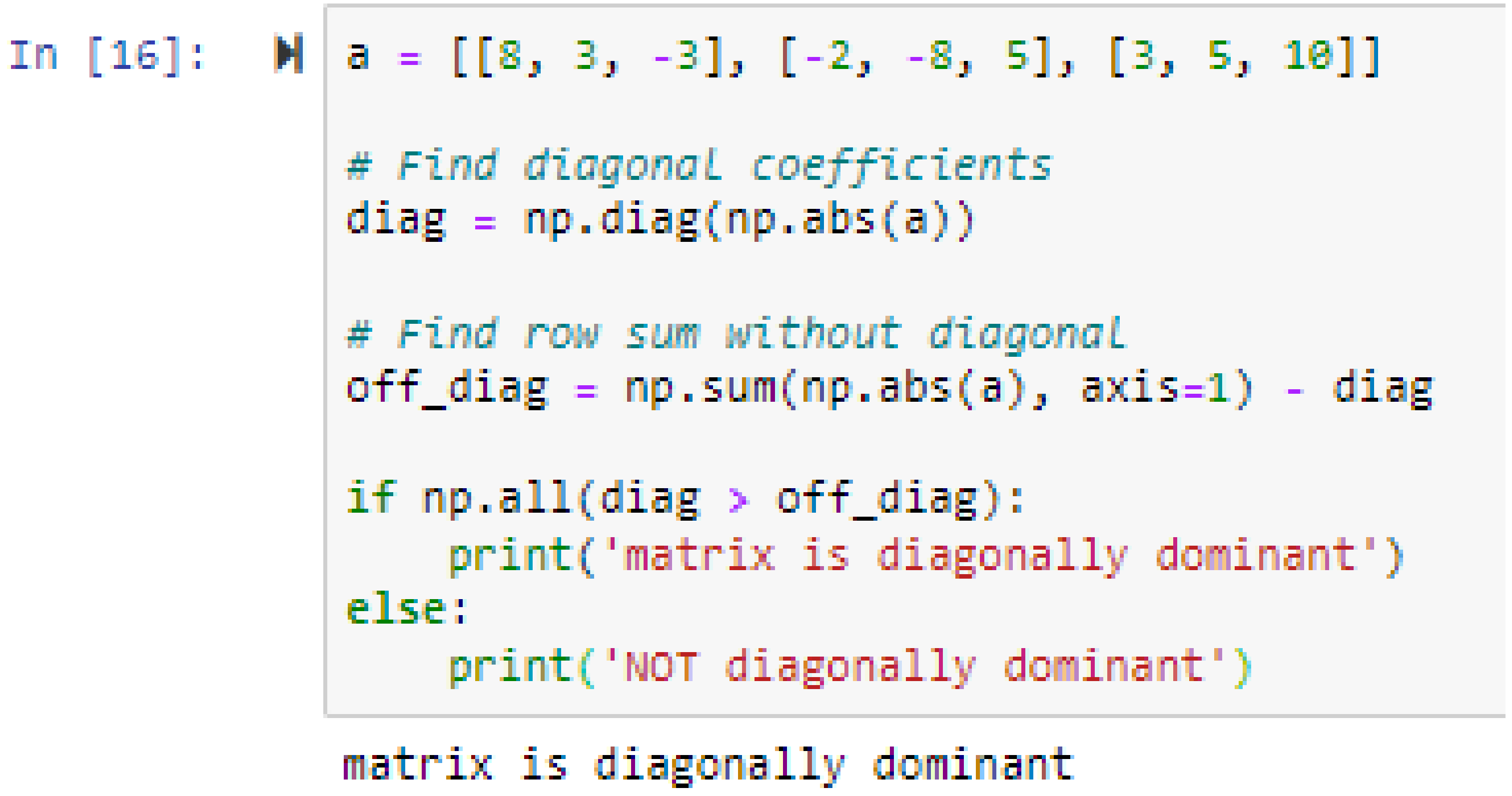
- The **Gauss-Seidel Method** is a specific **iterative** method, that is always using the **latest estimated value** for each elements in x. For example, we first assume the initial values for $x_2, x_3, \dots, x_n$ (except for x_1), and then we can calculate x_1. Using the calculated x_1 and the rest of the x (except for x_2), we can calculate x_2. We can continue in the same manner and calculate all the elements in x. This will conclude the first iteration. We can see the **unique part** of Gauss-Seidel method is that we are always using the **latest** value for calculate the next value in x. We can then continue with the iterations until the value converges.

- **EXAMPLE**: Solve the following system of linear equations using Gauss-Seidel method, use a pre-defined threshold $\epsilon = 0.01$. Do remember to check if the converge condition is satisfied or not.

$$8x_1 + 3x_2 - 3x_3 = 14$$
$$-2x_1 - 8x_2 + 5x_3 = 5$$
$$3x_1 + 5x_2 + 10x_3 = -8$$

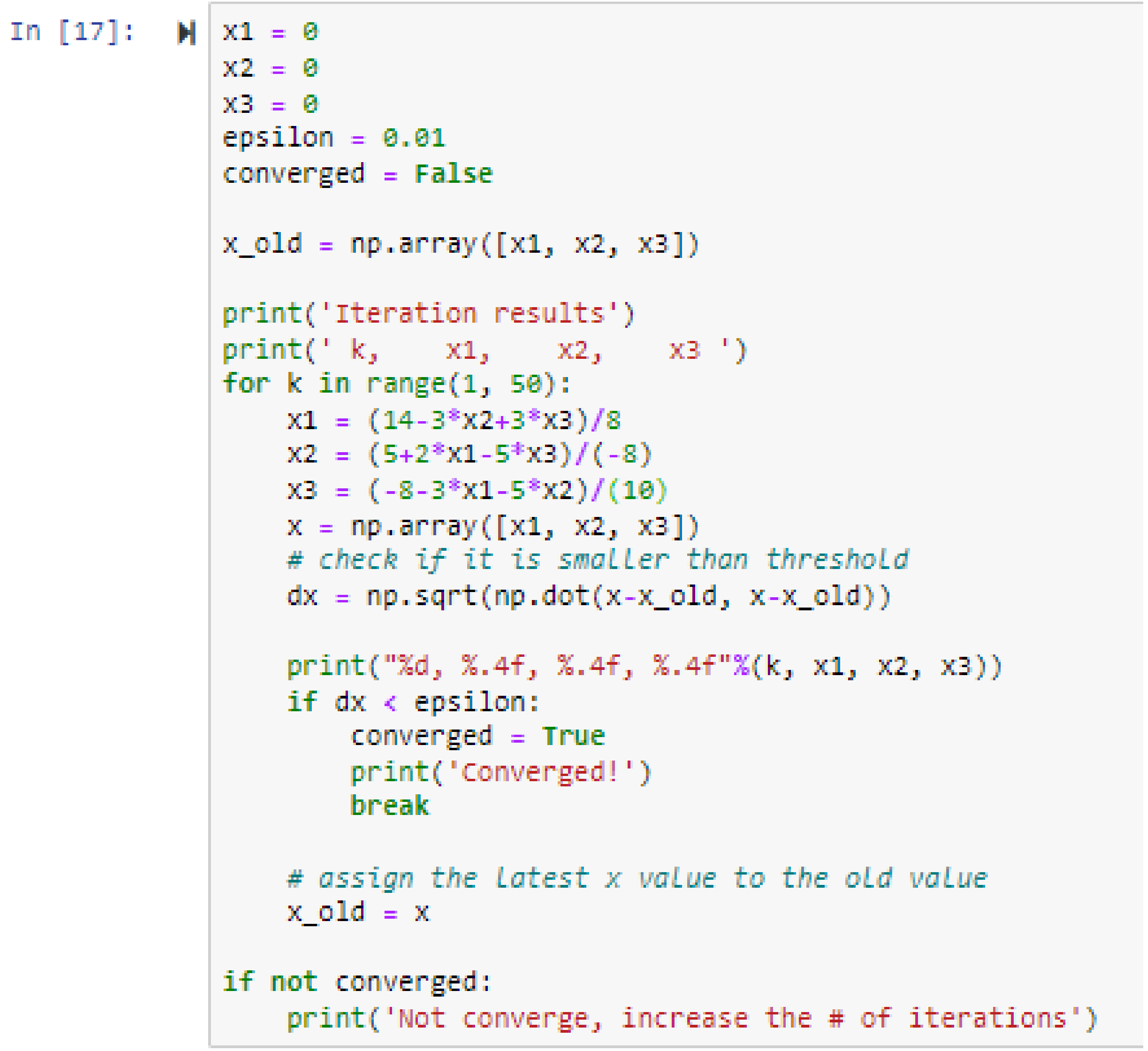
Iterative Method – Gauss-Seidel

- Let us first check if the coefficient matrix is **diagonally dominant** or not.

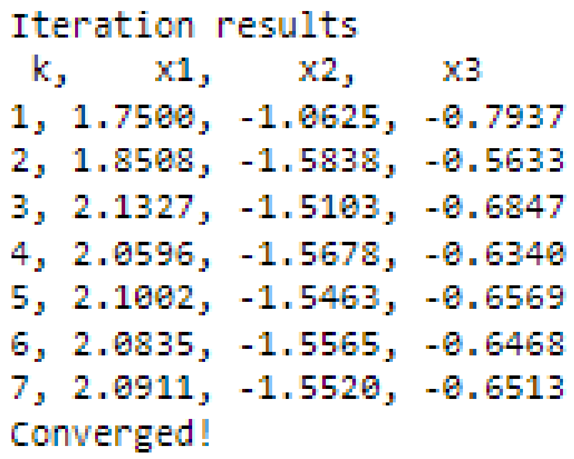


```
In [16]:   a = [[8, 3, -3], [-2, -8, 5], [3, 5, 10]]

           # Find diagonal coefficients
           diag = np.diag(np.abs(a))

           # Find row sum without diagonal
           off_diag = np.sum(np.abs(a), axis=1) - diag

           if np.all(diag > off_diag):
               print('matrix is diagonally dominant')
           else:
               print('NOT diagonally dominant')

           matrix is diagonally dominant
```

- Since it is guaranteed to converge, we can use **Gauss-Seidel** method to solve it.



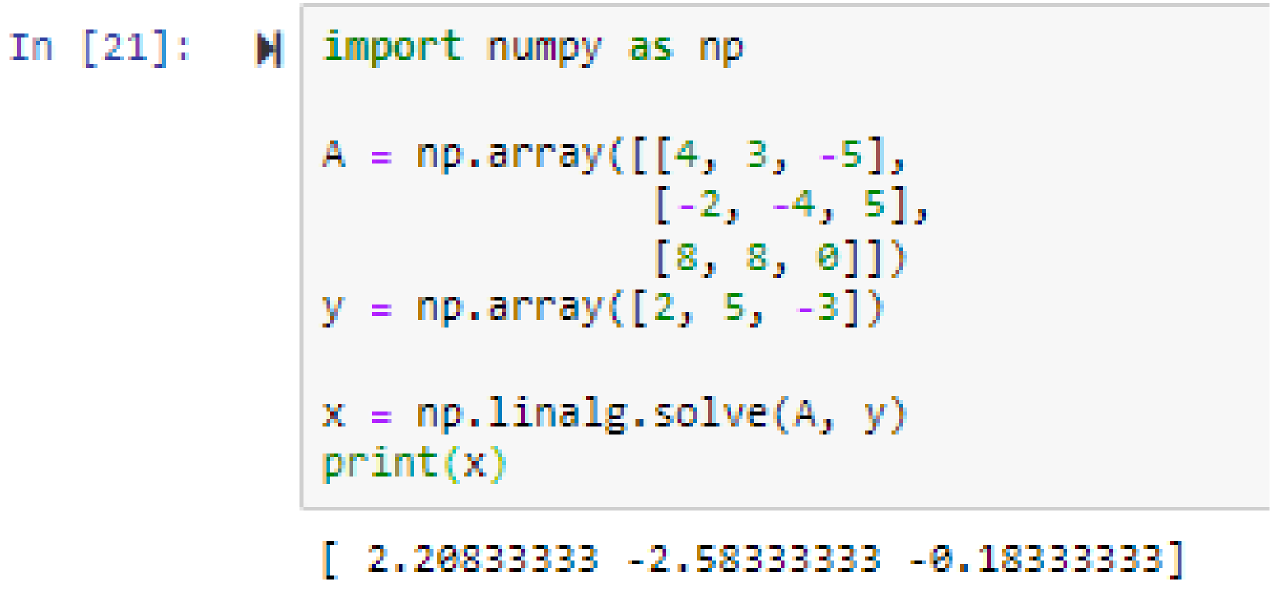
```
In [17]:  x1 = 0
          x2 = 0
          x3 = 0
          epsilon = 0.01
          converged = False

          x_old = np.array([x1, x2, x3])

          print('Iteration results')
          print(' k,    x1,    x2,    x3 ')
          for k in range(1, 50):
              x1 = (14-3*x2+3*x3)/8
              x2 = (5+2*x1-5*x3)/(-8)
              x3 = (-8-3*x1-5*x2)/(10)
              x = np.array([x1, x2, x3])
              # check if it is smaller than threshold
              dx = np.sqrt(np.dot(x-x_old, x-x_old))

              print("%d, %.4f, %.4f, %.4f"%(k, x1, x2, x3))
              if dx < epsilon:
                  converged = True
                  print('Converged!')
                  break

              # assign the latest x value to the old value
              x_old = x

          if not converged:
              print('Not converge, increase the # of iterations')
```



```
Iteration results
 k,    x1,    x2,    x3
1, 1.7500, -1.0625, -0.7937
2, 1.8508, -1.5838, -0.5633
3, 2.1327, -1.5103, -0.6847
4, 2.0596, -1.5678, -0.6340
5, 2.1002, -1.5463, -0.6569
6, 2.0835, -1.5565, -0.6468
7, 2.0911, -1.5520, -0.6513
Converged!
```

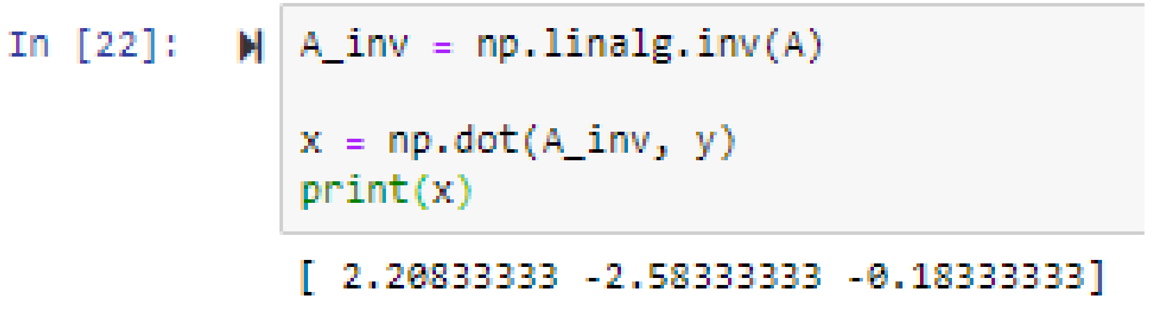
Solve the Systems in Python

- Though we discussed various methods to solve the systems of linear equations, it is actually very easy to do it in **Python**.
- In this section, we will use Python to solve the systems of equations. The easiest way to get a solution is via the **solve** function in **Numpy**.



```
In [21]:  import numpy as np

          A = np.array([[4, 3, -5],
                        [-2, -4, 5],
                        [8, 8, 0]])
          y = np.array([2, 5, -3])

          x = np.linalg.solve(A, y)
          print(x)

          [ 2.20833333 -2.58333333 -0.18333333]
```

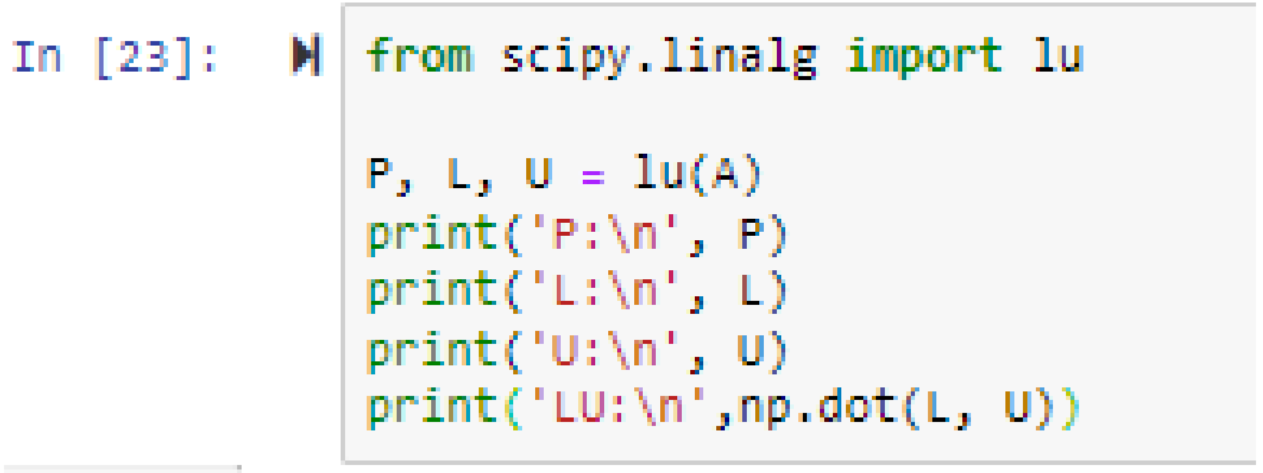
- Try to solve the equations using the **matrix inversion** approach.



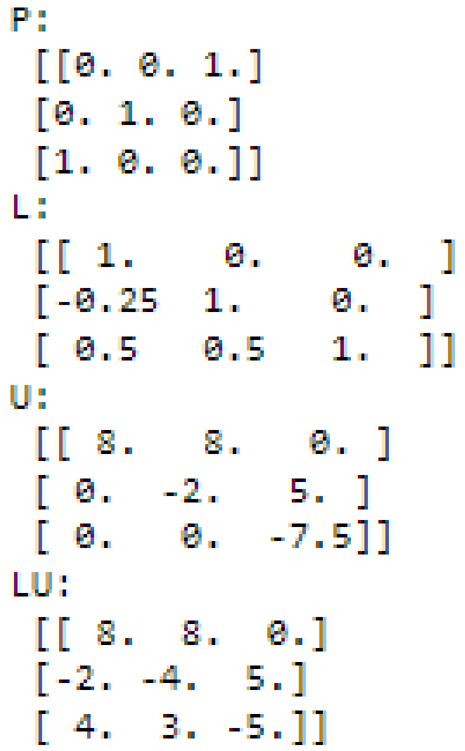
```
In [22]:  A_inv = np.linalg.inv(A)

          x = np.dot(A_inv, y)
          print(x)

          [ 2.20833333 -2.58333333 -0.18333333]
```

Solve the Systems in Python

- We can see we get the same results as that in the previous section when we calculated by hand. Under the hood, the solver is actually doing a **LU decomposition** to get the results. You can check the **help** of the **function**, it needs the input matrix to be **square** and of **full-rank**, i.e., all rows (or, equivalently, columns) must be **linearly independent**.
- We can also get the L and U matrices used in the LU decomposition using the **scipy** package.



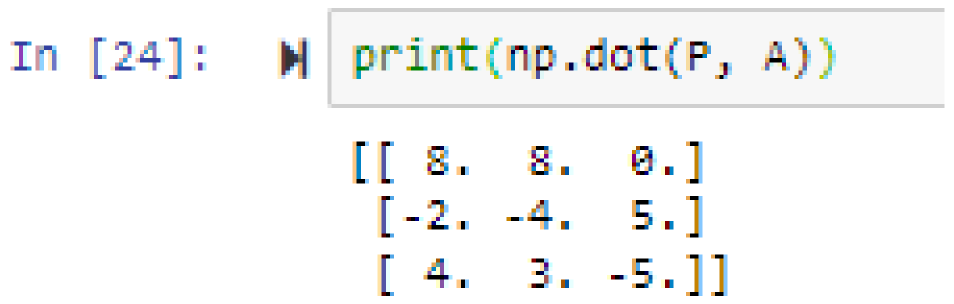
```
In [23]:  from scipy.linalg import lu

          P, L, U = lu(A)
          print('P:\n', P)
          print('L:\n', L)
          print('U:\n', U)
          print('LU:\n',np.dot(L, U))
```



```
P:
 [[0. 0. 1.]
 [0. 1. 0.]
 [1. 0. 0.]]
L:
 [[ 1.    0.    0.  ]
 [-0.25  1.    0.  ]
 [ 0.5   0.5   1.  ]]
U:
 [[ 8.   8.   0. ]
 [ 0.  -2.   5. ]
 [ 0.   0.  -7.5]]
LU:
 [[ 8.  8.  0.]
 [-2. -4.  5.]
 [ 4.  3. -5.]]
```

Solve the Systems in Python

- We can see the L and U we get are different from the ones we got in the last section by hand. You will also see there is a **permutation matrix** P that returned by the **lu** function.
- This **permutation matrix** record how do we change the order of the equations for easier calculation purposes (for example, if first element in first row is zero, it can not be the pivot equation, since you can not turn the first elements in other rows to zero. Therefore, we need to switch the order of the equations to get a new pivot equation).
- If you multiply P with A, you will see that this permutation matrix **reverse** the order of the equations for this case.



```
In [24]:    print(np.dot(P, A))

            [[ 8.  8.  0.]
             [-2. -4.  5.]
             [ 4.  3. -5.]]
```

Matrix Inversion

- We defined the **inverse** of a **square matrix** M is a matrix of the same size, M^{-1}, such that $M \cdot M^{-1} = M^{-1} \cdot M = I$. If the **dimension** of the matrix is **high**, the analytic solution for the matrix inversion will be **complicated**. Therefore, we need some other efficient ways to get the inverse of the matrix.

- Let us use a 4×4 matrix for illustration. If we have

$$M = \begin{bmatrix} m_{1,1} & m_{1,2} & m_{1,3} & m_{1,4} \\ m_{2,1} & m_{2,2} & m_{2,3} & m_{2,4} \\ m_{3,1} & m_{3,2} & m_{3,3} & m_{3,4} \\ m_{4,1} & m_{4,2} & m_{4,3} & m_{4,4} \end{bmatrix}$$

and the inverse of M is

$$X = \begin{bmatrix} x_{1,1} & x_{1,2} & x_{1,3} & x_{1,4} \\ x_{2,1} & x_{2,2} & x_{2,3} & x_{2,4} \\ x_{3,1} & x_{3,2} & x_{3,3} & x_{3,4} \\ x_{4,1} & x_{4,2} & x_{4,3} & x_{4,4} \end{bmatrix}$$

- Therefore, we will have:

$$M \cdot X = \begin{bmatrix} m_{1,1} & m_{1,2} & m_{1,3} & m_{1,4} \\ m_{2,1} & m_{2,2} & m_{2,3} & m_{2,4} \\ m_{3,1} & m_{3,2} & m_{3,3} & m_{3,4} \\ m_{4,1} & m_{4,2} & m_{4,3} & m_{4,4} \end{bmatrix} \begin{bmatrix} x_{1,1} & x_{1,2} & x_{1,3} & x_{1,4} \\ x_{2,1} & x_{2,2} & x_{2,3} & x_{2,4} \\ x_{3,1} & x_{3,2} & x_{3,3} & x_{3,4} \\ x_{4,1} & x_{4,2} & x_{4,3} & x_{4,4} \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

- We can rewrite the above equation to four separate equations, such as:

$$\begin{bmatrix} m_{1,1} & m_{1,2} & m_{1,3} & m_{1,4} \\ m_{2,1} & m_{2,2} & m_{2,3} & m_{2,4} \\ m_{3,1} & m_{3,2} & m_{3,3} & m_{3,4} \\ m_{4,1} & m_{4,2} & m_{4,3} & m_{4,4} \end{bmatrix} \begin{bmatrix} x_{1,1} \\ x_{2,1} \\ x_{3,1} \\ x_{4,1} \end{bmatrix} = \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \end{bmatrix}$$

$$\begin{bmatrix} m_{1,1} & m_{1,2} & m_{1,3} & m_{1,4} \\ m_{2,1} & m_{2,2} & m_{2,3} & m_{2,4} \\ m_{3,1} & m_{3,2} & m_{3,3} & m_{3,4} \\ m_{4,1} & m_{4,2} & m_{4,3} & m_{4,4} \end{bmatrix} \begin{bmatrix} x_{1,3} \\ x_{2,3} \\ x_{3,3} \\ x_{4,3} \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 1 \\ 0 \end{bmatrix}$$

$$\begin{bmatrix} m_{1,1} & m_{1,2} & m_{1,3} & m_{1,4} \\ m_{2,1} & m_{2,2} & m_{2,3} & m_{2,4} \\ m_{3,1} & m_{3,2} & m_{3,3} & m_{3,4} \\ m_{4,1} & m_{4,2} & m_{4,3} & m_{4,4} \end{bmatrix} \begin{bmatrix} x_{1,2} \\ x_{2,2} \\ x_{3,2} \\ x_{4,2} \end{bmatrix} = \begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \end{bmatrix}$$

$$\begin{bmatrix} m_{1,1} & m_{1,2} & m_{1,3} & m_{1,4} \\ m_{2,1} & m_{2,2} & m_{2,3} & m_{2,4} \\ m_{3,1} & m_{3,2} & m_{3,3} & m_{3,4} \\ m_{4,1} & m_{4,2} & m_{4,3} & m_{4,4} \end{bmatrix} \begin{bmatrix} x_{1,4} \\ x_{2,4} \\ x_{3,4} \\ x_{4,4} \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \end{bmatrix}$$

- Therefore, if we solve the above **four system of equations**, we will get the **inverse** of the matrix. We can use any method we introduced previously to solve these equations, such as **Gauss Elimination**, **Gauss-Jordan**, and **LU decomposition**.
- Recall that, in **Gauss-Jordan method**, we convert our problem from

$$\begin{bmatrix} m_{1,1} & m_{1,2} & m_{1,3} & m_{1,4} \\ m_{2,1} & m_{2,2} & m_{2,3} & m_{2,4} \\ m_{3,1} & m_{3,2} & m_{3,3} & m_{3,4} \\ m_{4,1} & m_{4,2} & m_{4,3} & m_{4,4} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix} = \begin{bmatrix} y_1 \\ y_2 \\ y_3 \\ y_4 \end{bmatrix} \Rightarrow \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix} = \begin{bmatrix} y'_1 \\ y'_2 \\ y'_3 \\ y'_4 \end{bmatrix}$$

- Essentially, we are **converting**

$$\begin{bmatrix} m_{1,1} & m_{1,2} & m_{1,3} & m_{1,4} & y_1 \\ m_{2,1} & m_{2,2} & m_{2,3} & m_{2,4} & y_2 \\ m_{3,1} & m_{3,2} & m_{3,3} & m_{3,4} & y_3 \\ m_{4,1} & m_{4,2} & m_{4,3} & m_{4,4} & y_4 \end{bmatrix} \Rightarrow \begin{bmatrix} 1 & 0 & 0 & 0 & y'_1 \\ 0 & 1 & 0 & 0 & y'_2 \\ 0 & 0 & 1 & 0 & y'_3 \\ 0 & 0 & 0 & 1 & y'_4 \end{bmatrix}$$

Matrix Inversion

- Let us generalize it here, all we need to do is to convert

$$\begin{bmatrix} m_{1,1} & m_{1,2} & m_{1,3} & m_{1,4} & 1 & 0 & 0 & 0 \\ m_{2,1} & m_{2,2} & m_{2,3} & m_{2,4} & 0 & 1 & 0 & 0 \\ m_{3,1} & m_{3,2} & m_{3,3} & m_{3,4} & 0 & 0 & 1 & 0 \\ m_{4,1} & m_{4,2} & m_{4,3} & m_{4,4} & 0 & 0 & 0 & 1 \end{bmatrix} \Longrightarrow \begin{bmatrix} 1 & 0 & 0 & 0 & m'_{1,1} & m'_{1,2} & m'_{1,3} & m'_{1,4} \\ 0 & 1 & 0 & 0 & m'_{2,1} & m'_{2,2} & m'_{2,3} & m'_{2,4} \\ 0 & 0 & 1 & 0 & m'_{3,1} & m'_{3,2} & m'_{3,3} & m'_{1,4} \\ 0 & 0 & 0 & 1 & m'_{4,1} & m'_{4,2} & m'_{4,3} & m'_{1,4} \end{bmatrix}$$

- Then the matrix

$$\begin{bmatrix} m'_{1,1} & m'_{1,2} & m'_{1,3} & m'_{1,4} \\ m'_{2,1} & m'_{2,2} & m'_{2,3} & m'_{2,4} \\ m'_{3,1} & m'_{3,2} & m'_{3,3} & m'_{1,4} \\ m'_{4,1} & m'_{4,2} & m'_{4,3} & m'_{1,4} \end{bmatrix}$$

is the **inverse** of M we are looking for.

Practice

1. Use Gauss Elimination to solve the following equations!

$$3x_1 - x_2 + 4x_3 = 2$$
$$17x_1 + 2x_2 + x_3 = 14$$
$$x_1 + 12x_2 - 7x_3 = 54$$

2. Use Gauss-Jordan Elimination to solve the above equations!

3. Determine the lower triangular matrix L and upper triangular matrix U from the equations!

Next Week's Outline

- Eigenvalues and Eigenvectors Problem Statement
- The Power Method
- The QR Method
- Eigenvalues and Eigenvectors in Python

References

- Kong, Qingkai; Siauw, Timmy, and Bayen, Alexandre. 2020. Python Programming and Numerical Methods: A Guide for Engineers and Scientists. Academic Press. https://www.elsevier.com/books/python-programming-and-numerical-methods/kong/978-0-12-819549-9
- Other online and offline references

Visi

Menjadi Program Studi Strata Satu Informatika **unggulan** yang menghasilkan lulusan **berwawasan internasional** yang **kompeten** di bidang Ilmu Komputer (*Computer Science*), **berjiwa wirausaha** dan **berbudi pekerti luhur**.

INFORMATIKA
UMN
UNIVERSITAS MULTIMEDIA NUSANTARA

Misi

1. Menyelenggarakan pembelajaran dengan teknologi dan kurikulum terbaik serta didukung tenaga pengajar profesional.
2. Melaksanakan kegiatan penelitian di bidang Informatika untuk memajukan ilmu dan teknologi Informatika.
3. Melaksanakan kegiatan pengabdian kepada masyarakat berbasis ilmu dan teknologi Informatika dalam rangka mengamalkan ilmu dan teknologi Informatika.