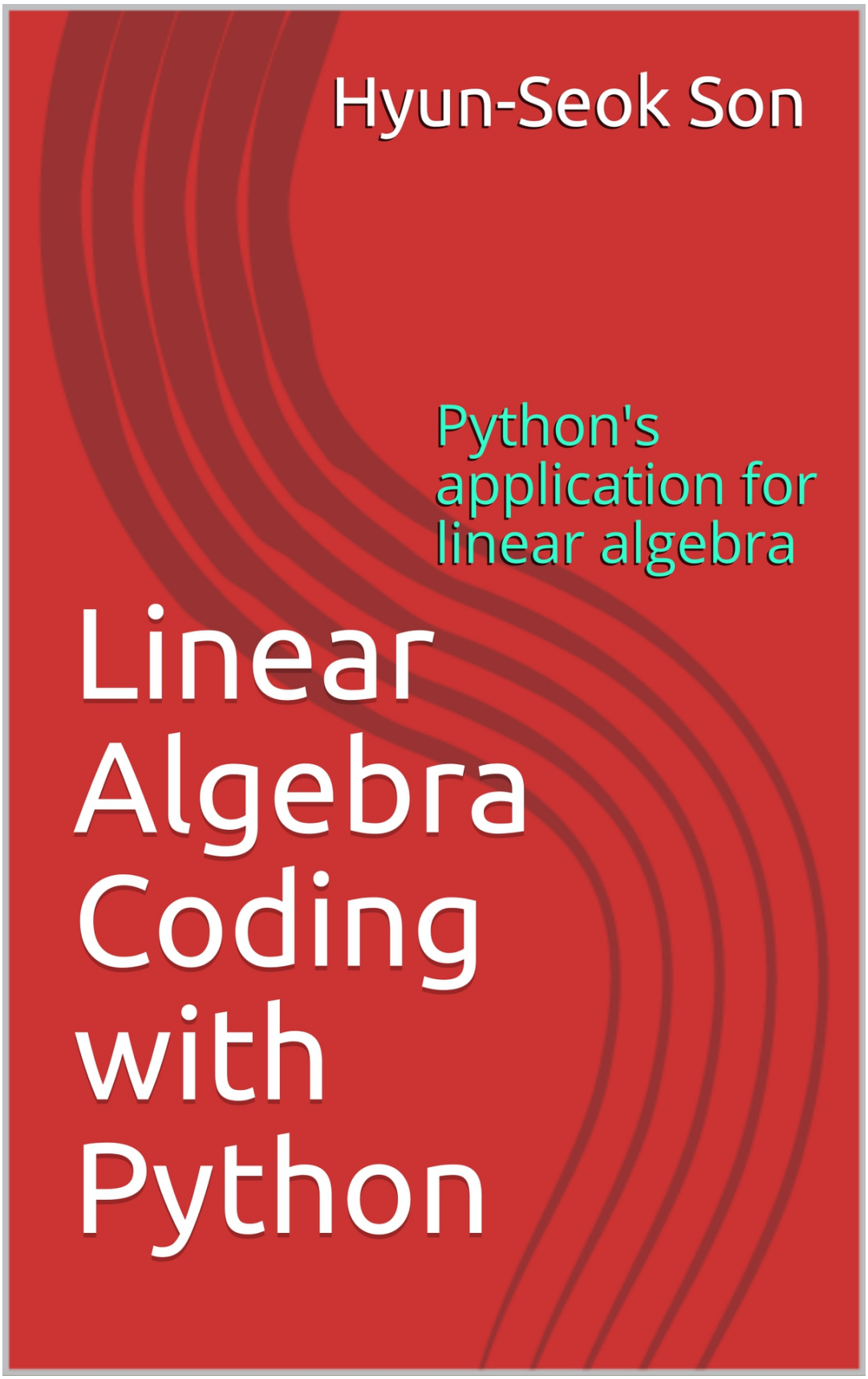
Hyun-Seok Son

Python's application for linear algebra

Linear Algebra Coding with Python


Hyun-Seok Son
Python's application for linear algebra
Linear Algebra Coding with Python

Linear Algebra Coding with Python

Author

The author studied at Seoul National University (Ph.D.) and currently serves as the director of Nnode LTD. He is interested in analyzing data with Pyhton and R.

He published "Create forecast model for stock with regression analysis and R", "python manual", "Trigonometry and Limit Story" and "Calculus Story I with Python", etc.

sonhs67@gmail.com

datastory1.blogspot.com

Preface

Python is one of the most popular languages for data analysis and prediction. What's more, tensorflow and torch, useful tools of recent deep learning, are fully implemented by Python. The basic form of data in these languages is an array, created by Python's important package numpy. In particular, arrays are the basis of data science because they have structures of vectors and matrices that give the meaning of direction and magnitude to each value in the data set. The matrix structure allows transformation to a simple form without losing the basic characteristics of a vast data set. These transformations are useful for efficient processing of data and for finding implicit characteristics.

Linear Algebra, a field that provides a basic theory of vectors and matrices, provides many algorithms to increase the accuracy and speed of computation for analyzing data and to discover the characteristics of a data set. These algorithms are very useful for understanding the computing process of probability, statistics and the learning machine.

This book introduces many basics of linear algebra using Python packages numpy, sympy, and so on. Chapters 1 and 2 introduce the creation and characteristics of vectors and matrices. Chapter 3 describes the linear system(linear combination) through the process finding the solution in a system of simultaneous equations. Vector space, a concept introduced in Chapter 4, is used to infer the collective characteristics and relationships of each vector of a linear system. Chapter 5 introduces the coordinate system to represent the linear system geometrically. Chapter 6 introduces the process of transforming while maintaining basic characteristics such as vectors and matrices. Finally, Chapter 7 describes several ways to decompose the original form into a simple form. In this process, we use a variety of Python functions. To use these functions, you need to import the following Python packages.

```
import numpy as np
from sympy import *
```

In addition, column vectors in the text are indicated by angle brackets (<>), vectors are shown in lowercase letters, and matrices are shown in uppercase letters.

Hyun-Seok Son

Table of Contents

1 Vector

1.1 Vector

1.1.1 Scalar and Vector

What does the Fig. 1.1 represent?

1) The x-axis and y-axis are based on East-West and South-North respectively.

2) It shows the position and distance of 4 points along each axis.

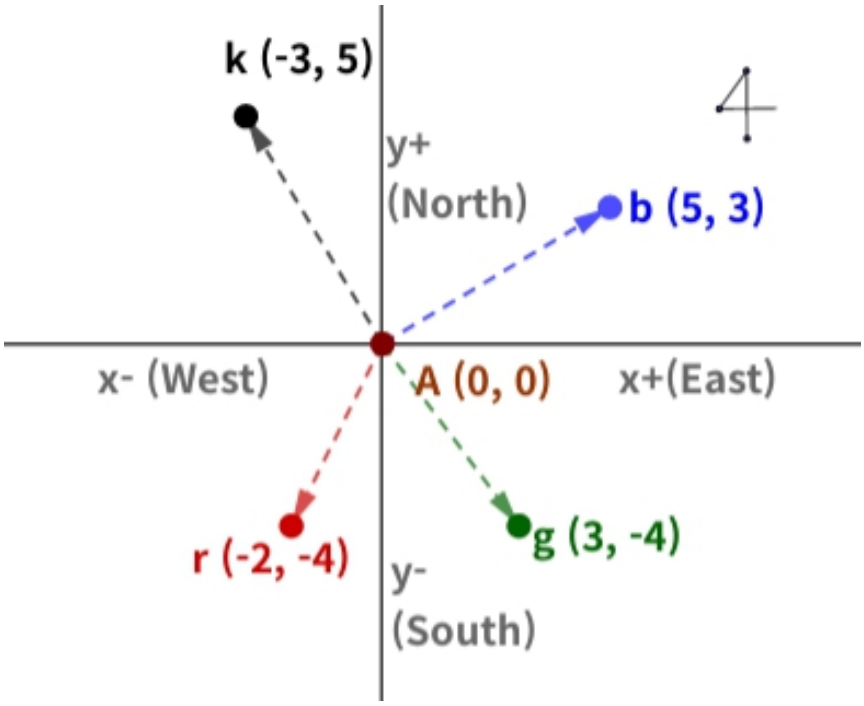
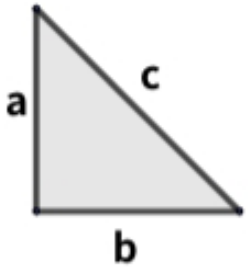


Fig. 1.1 Vector and coordinates.

The data in Fig. 1.1 could give a variety of meanings besides the meaning of directions. Whatever the interpretation, we can extract the two above.

The size and direction of each point can be calculated based on point A. For example, for point b, it is located at (5, 3) and the straight line distance from A is 4. This linear distance can be calculated using the famous Pythagorean theorem of a right triangle.



The length of each side of a right triangle is calculated by the Pythagorean Theorem shown below.

$$c^2 = a^2 + b^2$$
$$\rightarrow c = (a^2 + b^2)^{1/2}$$

In Fig. 1.1, the position of b is 5 on the x-axis and 3 on the y-axis, so it can be regarded as a right triangle with a base(b) of 5 and a height(a) of 3. That is, the straight line distance between the origin(A) and b is as follows.

```
In [1]: ab=(5**2+3**2)**0.5 # √(52+32)
   ...: round(ab,2)
Out[1]: 5.83
```

Each point is calculated based on the origin(A) as above.

At point b, the length of the base is (5-0) and the height is (3-0). Therefore, formulating this relationship gives the following:

```
In [1]: import numpy as np
   ...: point=np.array([[5,3],
   ...:                 [3,-4],
   ...:                 [-2,-4],
   ...:                 [-3,5]])
   ...: print(point)
```

```
[[ 5  3]
 [ 3 -4]
 [-2 -4]
 [-3  5]]

In [2]: import pandas as pd
   ...: d=[(point[I,0]**2+point[I,1]**2)**0.5
   ...:     for I in range(4)]
   ...: re=np.c_[point, d]
   ...: re1=pd.DataFrame(re)
   ...: re1.columns=['x','y','distance']
   ...: re1.index=['b','g','r','k']
   ...: print(re1)
     x      y     distance
b  5.0  3.0  5.830952
g  3.0 -4.0  5.000000
r -2.0 -4.0  4.472136
k -3.0  5.0  5.830952
```

$$\text{Distance} = [(x_1 - x_0)^2 - (y_1 - y_0)^2]^{1/2} \quad \text{(Eq. 1.1)}$$

x_0 , y_0 : start point

x_1 , y_1 : end point

As shown in Eq. 1.1, the coordinates of each point contain the direction and distance from the reference point and are called vectors. On the other hand, the size of each point, such as the distance, is called a scalar. It can be defined as:

Vector : values with magnitude and direction
Scalar : value with size only

In Fig. 1.1, all coordinates are based on the origin(0, 0). However, if these default coordinates are not the origin, you can think that the coordinates themselves are shifted. For example, when A is moved to (1,1), it is changed as in Fig. 1.2.

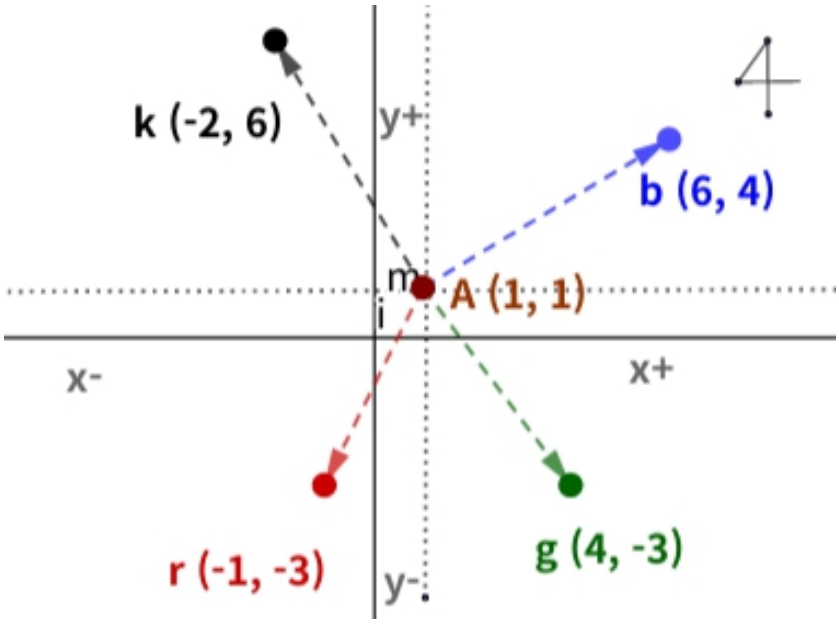


Fig. 1.2 Shift of basic coordinates.

In the case of Fig. 1.2, the reference point A in Fig. 1.1 is moved from (0,0) to (1,1). This means that the reference axis is increased by 1 in the x and y directions, and all positions of each point are increased by the same amount. Therefore, the change of the basic axis induces the change of all other points, so the distance of each point will not change.

1.1.2 Dimension and axis

In Python, vectors and matrices that are a combination of multiple vectors can be represented using numpy's array(array object) . In particular, numpy's array objects are used not only for vectors and matrices, but also for the basic data types of most datasets to represent tables. For example, it is used as a basic type of sympy matrix objects used for vector or matrix operations, and pandas Series and DataFrame structures, which are important libraries for data analysis.

Numpy array objects can be created by the np.array() function. Each value of the array object created by this function has an index. This index

indicates the position of the value within the object.

The argument of this function should be given in the form of a list, which is an ordered data type. One list represents one row .

```
In [1]: a=np.array([1,2]) #row vector
   ...: print(a)
[1 2]

In [2]: a1=np.array([[1],[2]]) #column vector
   ...: print(a1)
[[1]
 [2]]
```

In the code above, object a consists of one row and is called a row vector . In the case of object a_1 , it is an object composed of elements such as a, but it is called a column vector because it is in the form of one column.

Both vectors contain the same elements, but geometrically they have different meanings.

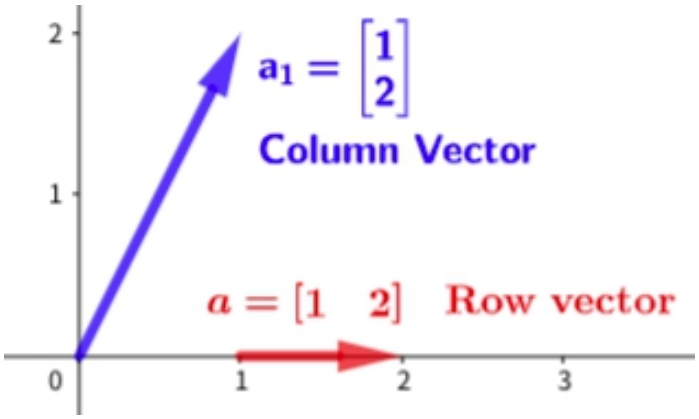


Fig. 1.3 Row vector and column vector.

As in Fig. 1.3, objects a and a_1 both represent a line, but for a , they appear on one axis, but for a_1 , on a plane formed by two axes, x and y. Therefore, a can be represented in one dimension, but a_1 can only be displayed in two dimensions.

You can check the dimension and shape of an object by using the ndim and shape properties of the numpy object.

```
In [3]: a.ndim
Out[3]: 1

In [4]: a.shape
Out[4]: (2,)

In [5]: a1.ndim
Out[5]: 2

In [6]: a1.shape
Out[6]: (2, 1)
```

A vector is any quantity that has a position and size, and a matrix means combining multiple vectors. In this sense, a_1 can be regarded as a matrix by combining two row vectors.

In Python, a single vector is represented in the form of (# of elements,) , and a collection of multiple vectors such as a_1 above, that is, a matrix is expressed as (# of rows, # of columns) . This can be summarized as follows.

> Brackets ([]): Refers to the axis.
> Therefore, the number of square brackets within the object determines the dimension.

Fig. 1.1 consists of two axes, x and y. The four points are represented by the values of the x-axis and the y-axis. In other words, it can be represented as:

	b	g	r	k
x-axis	5.0	3.0	-2.0	-3.0

y-axis	3.0	-4.0	-4.0	5.0

Each of the above points is a vector representing the direction and size based on the origin (0, 0), and the entire table is represented by a group by combining four vectors, called a matrix, and is two-dimensional because the two axes are related.

```
In [1]: arr=np.array([[5.,3.,-2.,-3.],
   ...:               [3.,-4.,-4.,5.]])
   ...: print(arr)
[[ 5.  3. -2. -3.]
 [ 3. -4. -4.  5.]]
```

In the case of the object arr , there are two brackets, which means two axes. The presence of two axes represents a plane, or two-dimensional. In summary, one axis represents a line, or one-dimensional, and three axes represents a space, or three-dimensional. In this way, the concept of dimension represents direction as well as size. Then, the scalar that represents only the size is called 0-dimensional without dimension.

The object arr is a two-dimensional matrix of two rows and four columns. In other words, it is represented as a 2×4 matrix.

```
In [2]: arr.ndim
Out[2]: 2

In [3]: arr.shape
Out[3]: (2, 4)
```

There are cases where the shape or dimension of vectors and matrices needs to be changed in their operation. To convert the structure of an array object, we apply the np.reshape() function. The argument of this function is the number of rows and columns to switch.

In the following code, the number of rows as an argument is -1. This means that the value is automatically assigned based on the value of the

other argument, the column. In the following cases, the number of rows is automatically adjusted based on the number of columns 2.

```
In [4]: print(arr.reshape(-1, 2))
[[ 5.  3.]
 [-2. -3.]
 [ 3. -4.]
 [-4.  5.]]
```

1.1.3 Norm and Unit Vector

The distance calculated using Eq. 1.1 is called norm or Euclidean distance and is expressed as $\|x\|$.

Above we have introduced the case of two axes to visualize the vector, but theoretically these axes can be infinitely extended. For example, the size (norm) of a 3-dimension vector $a = (a_1, a_2, a_3)$ with x, y, and z axes is calculated as Eq. 1.2 based on the $b = (b_1, b_2, b_3)$

$$\|a-b\| = [(a_1 - b_1)^2 + (a_2 - b_2)^2 + (a_3 - b_3)^2]^{1/2} \text{ (Eq. 1.2)}$$

Of course, if the start point is not specified, it is considered as the origin(0, 0, 0). You can use the np.linalg.norm(x) function for norm calculation.

```
In [1]: import numpy.linalg as la
   ...: la.norm([[5.0],[3.0]])
Out[1]: 5.830951894845301
```

The unit vector is a vector whose size (norm) is 1, and can be calculated as Eq. 1.3.

$a = \langle a_1 , a_2 \rangle$

$$\text{unit vector of } a = \left[\frac{a_1}{\|a\|} \quad \frac{a_2}{\|a\|} \right] \quad \text{(Eq. 1.3)}$$

```
In [1]: a=np.array([[2],[7]])
   ...: a_norm=la.norm(a)
   ...: a_unit=a/a_norm
   ...: print(a_unit)
[[0.27472113]
 [0.96152395]]

In [2]: la.norm(a_unit)
Out[2]: 1.0
```

1.2 Vector Operations

1.2.1 Addition and subtraction, and scalar times

Vector operations are performed between elements with the same index. Therefore, there is no operation between different types of vectors.

```
In [1]: a=np.array([10,15])
   ...: b=np.array([8,2])
   ...: c=np.array([1,2,3])
   ...: a+b
Out[1]: array([18, 17])

In [2]: a-b
Out[2]: array([ 2, 13])

In [3]: a-c
ValueError: operands could not be broadcast together with shapes (2,) (3,)
```

In the above code, objects a and c have different shapes, so no operation is performed. The operation between a and b can be represented as in Fig.1.4.

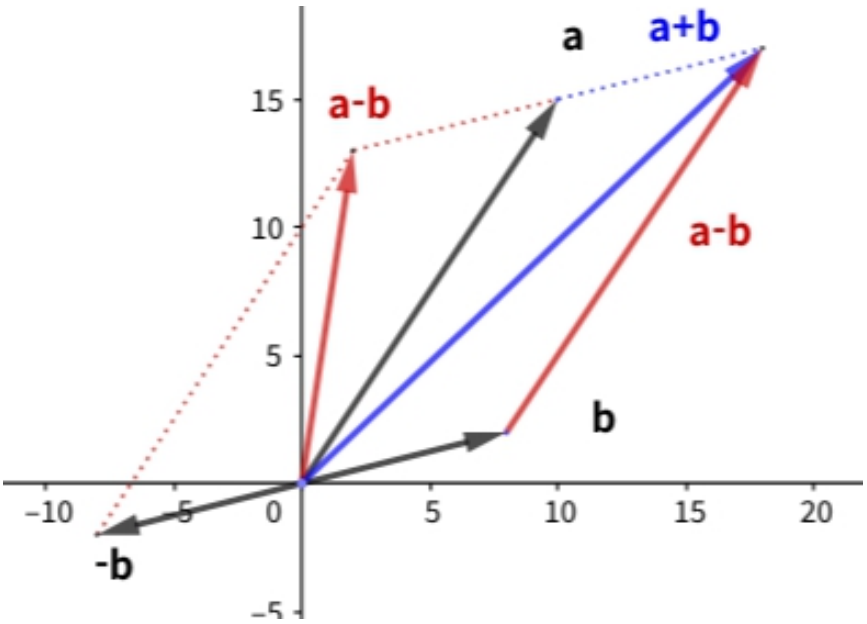


Fig. 1.4 Addition and subtraction of two vectors.

The addition operation of two vectors is the same as the vector corresponding to the diagonal of a parallelogram that both vectors can produce. Subtraction is the same as the addition operation with the result of multiplying b by scalar -1, like a + (-b) . Therefore, as shown in Fig. 1.4, the result of subtraction is the same as the vector representing the diagonal of the parallelogram between vector a and vector -b .

As shown in Eq. 1.4 through 9, the laws of operation can be defined between vectors u , v , w and two scalars a , b .

$$u+v = v+u \quad \text{(Eq. 1.4)}$$
$$u+(v+w) = (v+u)+w \quad \text{(Eq. 1.5)}$$
$$u+0 = u \quad \text{(Eq. 1.6)}$$
$$1\cdot u = u \quad \text{(Eq. 1.7)}$$
$$a\cdot(v+w) = a\cdot v+a\cdot w \quad \text{(Eq. 1.8)}$$
$$(a+b)\cdot v = a\cdot v + b\cdot v \quad \text{(Eq. 1.9)}$$

1.2.2 Inner product

The dot product(inner product) of both vectors, a=<a_1, a_2>, b=<b_1, b_2>, are defined as in Eq. 1.10.

$$a \cdot b = a_1 \times b_1 + a_2 \times b_2 \quad \text{(Eq. 1.10)}$$

The calculation of the dot product can be expressed as a product between vectors executed in the way shown in Fig. 1.5.

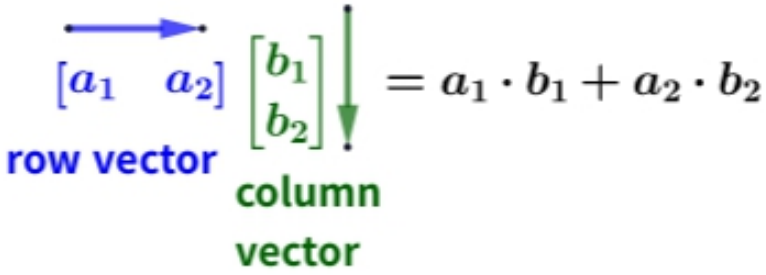


Fig. 1.5 Inner product of two vectors.

Fig. 1.5 shows the operation between the row of the front object and the column of the back object. Therefore, the number of front vector columns and the number of back vector rows must be the same. This method for dot products applies equally between matrices and is called matrix multiplication. The dimensions of each object in the matrix product operation are:

```
(r1×c1)·(r2 ×c2)=(r1×c2)
"c1=r2" must be satisfied.
  r1, r2: # of rows in matrix 1 and 2
  c1, c2: # of columns in matrix 1 and 2
```

The matrix product is calculated using the np.dot (x, y) function.

If the two vectors a and b have the same structure, the shape must be converted by exchanging the rows and columns of the vector. This is said to transpose the vector. (Fig. 1.6)

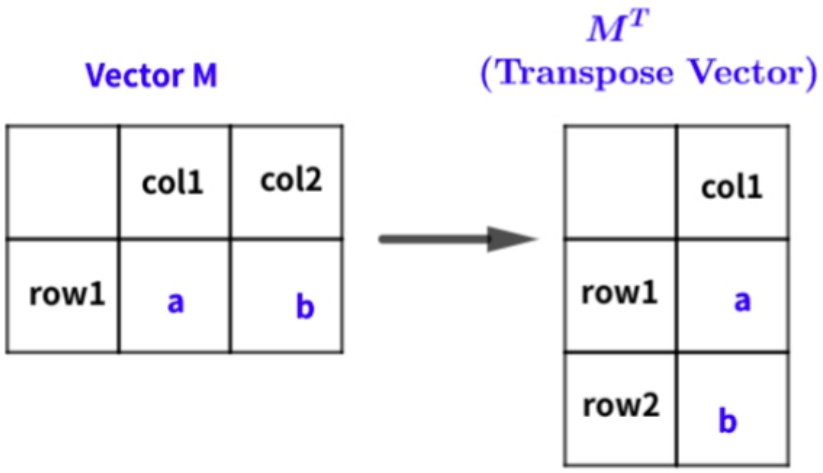


Fig. 1.6. Vector transposition process.

The transposition of an object as shown in Fig. 1.6 uses the obj.T property or np.transpose(obj) function.

For example, calculate the dot product of two vectors: a and b . Both vectors are column vectors.

```
In [1]: a=np.array([[2],[7]])
   ...: b=np.array([[5],[2]])
   ...: print(a)
[[2]
 [7]]

In [2]: print(b)
[[5]
 [2]]
```

Both vectors have the same shape as column vectors.

```
In [3]: a.shape
Out[3]: (2, 1)

In [4]: b.shape
Out[4]: (2, 1))
```

It is necessary to transpose one vector so that the number of columns in the vector is equal to the number of rows in the other vector. Also, because

the dot product is a scalar value that characterizes the vector, the preceding vector must be converted to a row vector.

```
In [5]: at=a.T
   ...: at
Out[5]: array([[2, 7]])

In [6]: at.shape
Out[6]: (1, 2)

In [7]: print(np.transpose(a))
[[2 7]]
```

The dot product of a 's transpose vectors at and b is:

```
In [8]: innerP=np.dot(at, b)
   ...: print(innerP)
[[24]]
```

The dot product can be calculated by applying the angle between two vectors, as in Eq. 1.11. Conversely, the angle between two vectors can be calculated using the lengths and dot products of the vectors.

$$u \cdot v = \|u\| \, \|v\| \cos(\theta)$$

$$\rightarrow \cos(\theta) = \frac{u \cdot v}{\|u\| \, \|v\|} \quad \text{(Eq. 1.11)}$$

Fig. 1.7 shows the angle θ between two vectors a and b .

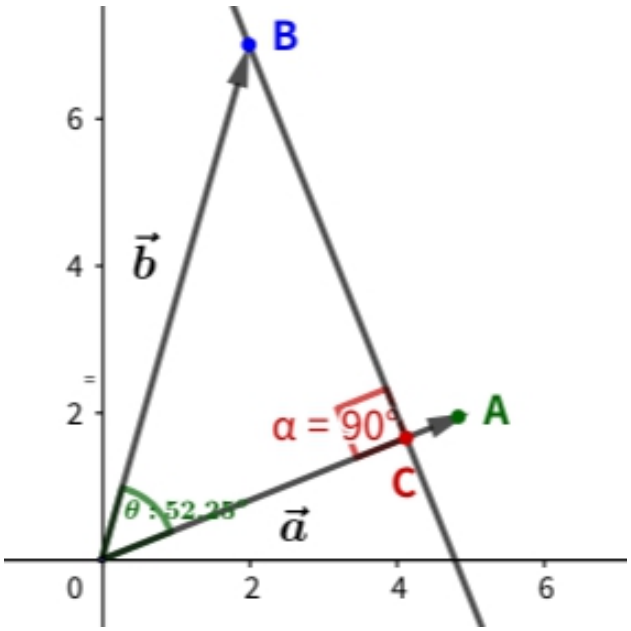


Fig. 1.7 The angle and dot product of two vectors.

The dot product (inner product) is the product of the length of vector a and the length of vector b projected over it, as in the following equation:

$$a=\|b\|\cos(\theta) = \overline{OC}$$
$$b\cdot a=\|b\|\|b\|\cos(\theta) = \|b\|\overline{OC}$$

The above calculation is performed using numpy functions such as np.cos() .

```
In [1]: theta=52.25
   ...: rad=np.radians(theta)
   ...: rad
Out[1]: 0.9119345341670372

In [2]: inner=np.round(
   ...:   la.norm(a)*la.norm(b)*np.cos(rad), 0)
   ...: inner
Out[2]: 24.0
```

The argument passed to numpy's trigonometric functions must be radians. Therefore, the degree was converted to radians by applying np.radians() .

As another example, calculate the angle between vectors a and b .

```
In [1]: a=np.array([[-2],[1]])
   ...: b=np.array([[-3],[1]])
   ...:
   ...: # inner product between a and b
   ...: ab=np.dot(a.T, b)
   ...:
   ...: # norm of a
   ...: a_norm=la.norm(a)
   ...:
   ...: # norm of b
   ...: b_norm=la.norm(b)
   ...:
   ...: cos=ab/(a_norm*b_norm)
   ...: print(cos)
[[0.98994949]]

In [2]: rad=np.arccos(cos)
   ...: print(rad)
[[0.14189705]]

In [3]: deg=np.rad2deg(rad)
   ...: print(deg)
[[8.13010235]]
```

In the above process, np.arccos() and np.rad2deg() functions were used to calculate θ. The value calculated from the above expression is con(θ). To calculate θ from this result, Eq. 1.12 is applied.

$x=\cos(y) \rightarrow y=\cos^{-1}(x)$ (Eq. 1.12)

$\cos^{-1}$ is calculated by the arccos() function of the numpy module. The result is a radian value and should be converted to degrees. This reduction process is performed by the np.rad2deg() function.

1.2.3 Orthogonal vectors

According to Eq. 1.11, if the two vectors are at right angles, the dot product is zero. These vectors are called orthogonal vectors .

In Fig. 1.8, vectors u and v and vectors u and -v both show a right-angle relationship. That is, the vector u perpendicular to the vector v satisfies the following condition:

$$\|u-v\| = \|u-(-v)\|$$

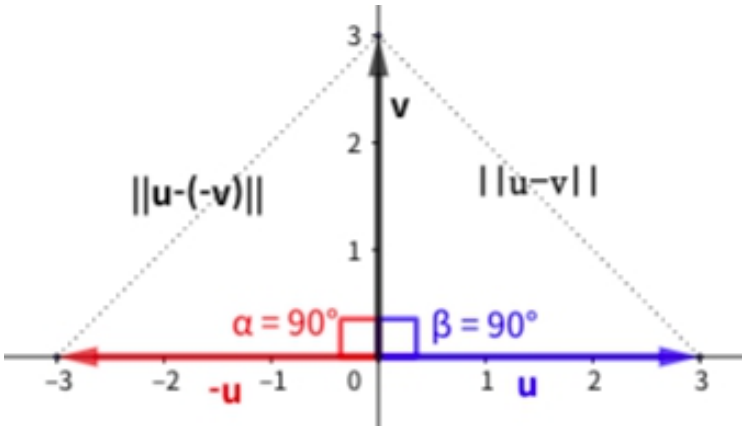


Fig. 1.8 Orthogonal vectors.

As shown in Fig.1.8, if two vectors u and v have a vertical relationship, they are said to represent orthogonality in linear algebra.

Vectors orthogonal to each other have the following characteristics:

- If two vectors $u \cdot v = 0$, the two vectors are orthogonal.
- The 0 vector is orthogonal to all vectors.

1.2.4 Cauchy-Schwarz inequality

You can use the dot product of the vector and the norm to define the relationship of the inequality as in Eq. 1.13. For any two vectors, the dot product is less than or equal to the product of each vector's norm.

$$\|uv\| \leq \|u\|\|v\| \quad \text{(Eq. 1.13)}$$

In Eq. 1.13, the equal sign is established when the two vectors are multiples of each other.

```
In [1]: u=np.array([[-1], [2]])
   ...: v=np.array([[4], [-2]])
   ...: inner=np.dot(u.T, v)
```

```
    ...: print(abs(inner))
[[8]]

In [2]: normProd=la.norm(u)*la.norm(v)
    ...: round(normProd, 3)
Out[2]: 10.000
```

1.2.5 Triangle inequality

An inequality like Eq. 1.14 holds for any two vectors. In other words, the norm of two vector sums is less than or equal to the sum of each vector norm.

$$\|u+v\| \leq \|u\|+\|v\| \quad \text{(Eq. 1.14)}$$

Eq. 1.14 is expressed as follows by squaring both sides:

$$\begin{aligned}\|u+v\|^2 &= \|u\|^2 + \|v\|^2 + 2\|uv\| \\ &\leq \|u\|^2 + \|v\|^2 + 2\|u\|\|v\|\end{aligned}$$

Eventually, Eq. 1.14 is established by Eq. 1.13.

```
In [1]: u_v_norm=la.norm(u+v)
    ...: u_v_norm
Out[1]: 3.0

In [2]: u_v_norm2=la.norm(u)+la.norm(v)
    ...: round(u_v_norm2, 3)
Out[2]: 6.708
```

1.2.6 Projections

When light is irradiated perpendicularly to the vector b in Fig. 1.9, the position vector of the shadow of b generated in the vector a is called

projection. The projected vector is called b_{proj} .

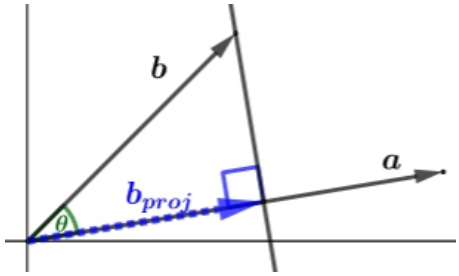


Fig. 1.9 Projection of vector b projected onto vector a.

In Fig. 1.9, if the angle between the two vectors is θ, the dot product can be calculated by Eq. 1.11. Applying the dot product and the law of cosine, the norm of the projected b(b_{proj}) is calculated as in Eq. 1.15.

$$\cos(\theta)=\frac{\|b_{proj}\|}{\|b\|}=\frac{a\cdot b}{\|a\|\,\|b\|}$$
$$\rightarrow \|b_{proj}\|=\frac{a\cdot b}{\|a\|} \quad \text{(Eq. 1.15)}$$

b_{proj} is the projection of b located above the vector a , so the size of the projection can be expressed as a multiple of the unit vector of a . Therefore, as shown in Eq. 1.16, the projection can be calculated by the dot product of two vectors and the norm.

$$a_{unit}=\frac{a}{\|a\|}$$
$$\rightarrow\ b_{proj}=\frac{a\cdot b}{\|a\|}\cdot\frac{a}{\|a\|}$$
$$=\frac{a\cdot b}{\|a\|^2}\cdot a \quad \text{(Eq. 1.16)}$$

The following code computes the orthographic b_{proj} of vector b over vector a .

```
In [1]: a=np.array([[1],[0]])
   ...: b=np.array([[2],[1]])
   ...: ab_d=np.dot(a.T, b)
   ...: a_norm=(la.norm(a))**2
   ...: b_proj=ab_d/a_norm *a
   ...: print(b_proj)
[[2.]
 [0.]]
```

The above result can be expressed as Fig. 1.10.

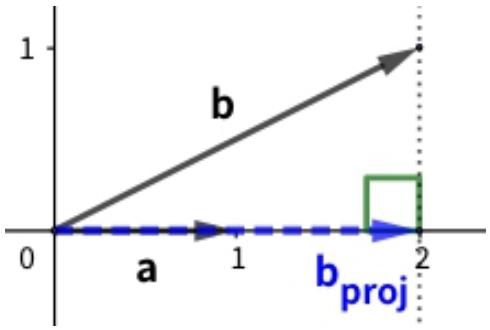


Fig. 1.10 Projection relationship of two vectors.

1.2.7 Outer product

Vectors that are perpendicular to any two vectors are produced by cross product. The cross product of two 3-dimension vectors, $a=<a_1, a_2, a_3>$, $b=<b_1, b_2, b_3>$,is calculated as Eq. 1.17.

$$a \times b = <a_2 b_3 - a_3 b_2, a_3 b_1 - a_1 b_3, a_1 b_2 - a_2 b_1> \text{ (Eq. 1.17)}$$

The above equation can be calculated using the minor of a matrix and a factor(see Minor (linear algebra)). You can do this using the np.cross() function. Calculate the cross product of two vectors a, b .

```
In [1]: a=np.array([[2], [1],[-1]])
   ...: b=np.array([[-3],[4], [1]])
```

```
    ...: print(np.cross(a.T,b.T))
[[ 5  1 11]]
```

The resulting vector above is perpendicular to vectors a and b, respectively, as shown in Fig. 1.11.

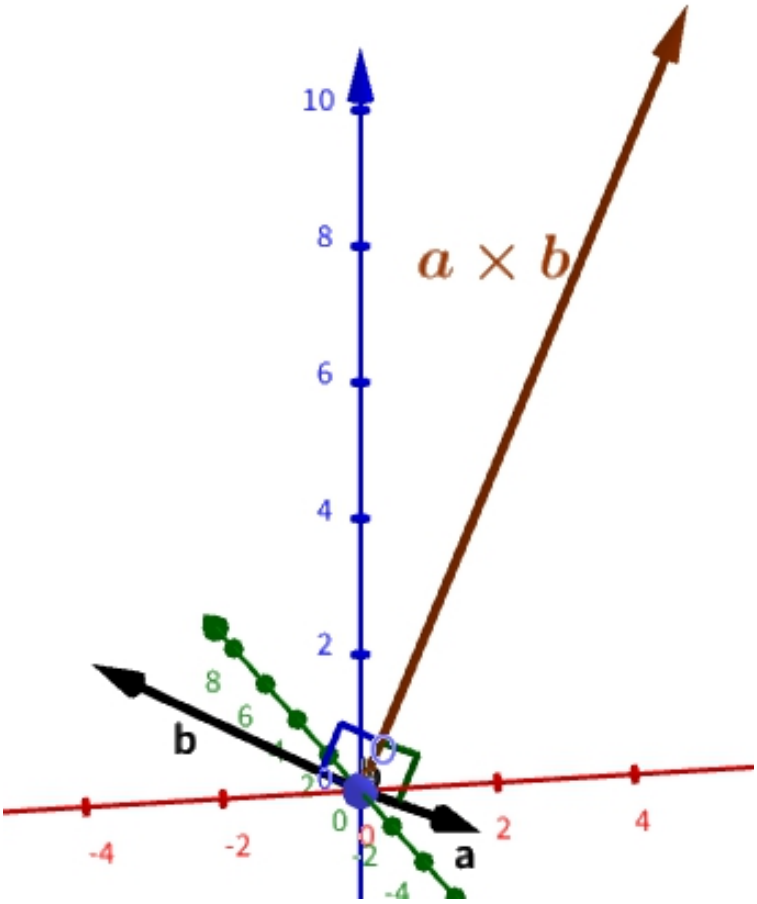


Fig.1.11 Cross product of two vectors.

If the two vectors are parallel, i.e. there is no point of intersection, then the cross product of the two vectors cannot exist. This relationship can be represented by Eq. 1.18.

$\|a \times b\| = \|a\| \|b\| \sin(\theta)$ (Eq. 1.18)
θ : angle between two vectors a, b
$\theta = 0$ or $180° \rightarrow \sin(\theta) = 0$, $\|a \times b\| = 0$

2 Matrix

2.1 Matrix

2.1.1 Creation of matrix

The matrix combines several vectors to represent a square structure organized into rows and columns. In the following code, matrix A with 3 rows and 2 columns (3×2) is the combination of two vectors a and b in the column direction.

```
In [1]: a=np.array([[2], [1],[-1]])
   ...: b=np.array([[-3],[4], [1]])
   ...: print(a)
[[ 2]
 [ 1]
 [-1]]

In [2]: print(b)
[[-3]
 [ 4]
 [ 1]]

In [3]: A=np.hstack((a, b))
   ...: print(A)
[[ 2 -3]
 [ 1  4]
 [-1  1]]
```

The np.hstack(a, b) used in the code above is a numpy function used to connect two objects in the direction of a column. Direct matrix construction

without vector joining also uses the np.array() function.

```
In [1]: B=np.array([[1,2,3], [4,5,6],[7,8,9]])
   ...: print(B)
[[1 2 3]
 [4 5 6]
 [7 8 9]]
```

2.1.2 Object slicing

When you create a vector or matrix, the index of each element is automatically assigned. The index is a zero-based integer value for each row and each column. Therefore, each element of the matrix has a row index and a column index. Table 2.1 shows the matrix and its indices.

Table 2.1. Index of Matrix

Row \ Column	Column 0	Column 1
Row 0	0 0	0 1
Row 1	1 0	1 1

The index is used to access each element in the object. Eq. 2.1 is a syntax for calling an element of an object.

```
Matrix[row, column]    (Eq. 2.1)
```

Using the syntax of Eq. 2.1, you can call multiple elements as well as a single element and create a submatrix of the original matrix. This is called slicing of objects.

Table 2.2 Slicing Rule

Rule	Contents
$A[r_1, c_2]$	value in row r_1 and column c_1
$A[:r_1, c_1:c_n]$	row: 0 ~ r_1, column: c_1 ~ c_{n-1}
A[:, :]	The colon (:) represents all ranges
A[-1, -2]	Negative (-) values in the index indicate backwards. For example -1 means first in the back

Note: The start of the index is 0.

```
In [1]: A=np.array([[2, 4, 9], [3, 6, 7]])
   ...: print(A)
[[2 4 9]
 [3 6 7]]

In [2]: A.shape
Out[2]: (2, 3)

In [3]: A[1,2]
Out[3]: 7

In [4]: A[:, 1]
Out[4]: array([4, 6])
```

2.1.3 Arithmetic operations

Between the matrices, the arithmetic operations are done between the same indices in element units. Therefore, the dimensions of the operands must be the same.

```
In [1]: m1=np.array([[2,8], [4,9]])
   ...: m2=np.array([[6,9],[1,0]])
   ...: m3=np.mat([[2,3,4,1]])
   ...: M=[m1, m2, m3]
   ...: for i,j in zip(M, range(len(M))):
   ...:     print(F"Shape of {'m'+str(j+1)}: {i.shape}")
Shape of m1: (2, 2)
Shape of m2: (2, 2)
Shape of m3: (1, 4)

In [2]: print(m1+m2)
[[ 8 17]
 [ 5  9]]

In [3]: try:
   ...:     m1+m3
   ...: except:
   ...:     print("The arithmetic operations \n \
   ...:           of the matrices must be of \n \
   ...:               the same dimension.")
The arithmetic operations
```

```
of the matrices must be of
    the same dimension.
```

2.1.4 Matrix product

Matrix is a combination of vectors. Therefore, like the dot product of a vector, you can compute the dot product between two matrices. This operation, the dot product between two matrices, is called the matrix product .

The matrix product of the two matrices consists of the operation between the elements of the row of the front matrix and the elements of the column of the rear matrix (See inner product).

$$\begin{bmatrix} x_{11} & x_{12} \\ x_{21} & x_{22} \end{bmatrix} \begin{bmatrix} y_{11} \\ y_{21} \end{bmatrix} = \begin{bmatrix} x_{11}y_{11}+x_{12}y_{21} \\ x_{21}y_{11}+x_{22}y_{21} \end{bmatrix} \quad \text{(Eq. 2.2)}$$

The system of equations can be expressed in the form of a matrix system to which matrix products can be applied.

$$\begin{matrix} a_{11}x+a_{12}y+a_{13}z \\ a_{21}x+a_{22}y+a_{23}z \\ a_{31}x+a_{32}y+a_{33}z \end{matrix} = \begin{bmatrix} a_{11}+a_{12}+a_{13} \\ a_{21}+a_{22}+a_{23} \\ a_{31}+a_{32}+a_{33} \end{bmatrix} \begin{bmatrix} x \\ y \\ z \end{bmatrix}$$

The matrix product between matrix A and two vectors u , v has the following relationship.

Establish $u^T v = v^T u$ on the dot product of two columns or row vectors ($matrix^T$ is a transpose matrix)

```
In [1]: u=np.array([-1,2,4]).reshape(3, 1)
   ...: print(u)
[[-1]
 [ 2]
```

```
 [ 4]]

In [2]: v=np.array([-2, 0, 5]).reshape(3, 1)

In [3]: print(np.equal(np.dot(u.T, v),np.dot(v.T, u)))
[[ True]]
```

The function np.equal(a, b) in the code above compares each element of the two vectors and returns "True" if they are equal, "False" otherwise.

$uv=v^T u^T$ between two row and column vectors

```
In [1]: u=np.array([-1,2,4])
   ...: v=np.array([-2, 0, 5]).reshape(-1,1)
   ...: print(np.dot(u, v))
[22]

In [2]: print(np.dot(v.T, u.T))
[22]
```

$(Au)v=u(A^T v)$ between two vectors (u, v) of the same dimension as matrix A

```
In [1]: A=np.array([1,3,-5, -2, 7,
   ...:       8, 4, 0, 6]).reshape(3,3)
   ...: print(A)
[[ 1  3 -5]
 [-2  7  8]
 [ 4  0  6]]

In [2]: A1=np.dot(np.dot(A, u).T, v)
   ...: print(A1)
[130]

In [3]: A2=np.dot(u.T, np.dot(A.T, v))
   ...: print(A2)
[130]
```

2.2 Special matrix

2.2.1 Transposed matrix

A matrix in which the rows and columns of m×n-dimensional matrix A are exchanged is called A 's transposed matrix and is represented by A^T . The dimension of the transposed matrix is n×m.

The relation of Eq. 2.3 is established between the matrix A and the transposed matrix.

$$A_{ij} = A^T{}_{ji} \text{ (Eq. 2.3)}$$

Transposed matrices can be created using the .T property of the numpy array object or the function transpose() .

```
In [1]: A=np.array([[2, 4, 9], [3, 6, 7]])
   ...: print(A)
[[2 4 9]
 [3 6 7]]

In [2]: print(np.transpose(A))
[[2 3]
 [4 6]
 [9 7]]

In [3]: print(A.T)
[[2 3]
 [4 6]
 [9 7]]
```

Transpose of the transpose matrix becomes the original matrix.

$$(A^T)^T = A$$

```
In [4]: print((A.T).T)
[[2 4 9]
 [3 6 7]]
```

The following relationship holds for the transpose of the matrix product between two matrices.

$$(A \cdot B)^T = B^T \cdot A^T$$

In the following code, the randint() function of the numpy.random module is used to create a random matrix object.

```
In [5]: B=np.random.randint(-10, 10, (3,3))
   ...: print(B)
[[ -3  -8  -9]
 [  4   0 -10]
 [ -5  -4  -6]]

# A·B
In [6]: AB=np.dot(A, B)

# BT·AT
In [7]: BtAt=np.dot(B.T, A.T)
   ...: print(np.equal(AB.T, BtAt))
[[ True  True]
 [ True  True]
 [ True  True]]
```

2.2.2 Square matrix

A matrix with the same row and column dimensions is called a square matrix, and the transpose of the square matrix is exchanged for the values of the elements other than the diagonal.

```
In [1]: A=np.array([[1,2,3],[4,5,6],[7,8,9]])
   ...: print(A)
[[1 2 3]
 [4 5 6]
 [7 8 9]]

In [2]: AT=A.T
   ...: print(AT)
[[1 4 7]
 [2 5 8]
 [3 6 9]]
```

2.2.3 Identity Matrix

The identity matrix is a matrix in which the diagonal elements are all 1 and all other elements are 0, and are represented by I_n .

n: # of rows or columns

The matrix product with the identity matrix makes that matrix itself.

```
AI = A (Eq. 2.4)
```

The np.eye(n) function can be used to create an n × n dimension identity matrix.

```
In [1]: print(np.eye(3))
[[1. 0. 0.]
 [0. 1. 0.]
 [0. 0. 1.]]
```

2.2.4 Trace

The sum of the diagonal elements of a square matrix is called the trace of the matrix and is represented by tr(A). It is calculated using the np.trace(A)

function.

```
In [1]: m=np.random.randint(1, 10, (3,3))
   ...: print(m)
[[9 8 9]
 [8 2 4]
 [5 5 4]]

In [2]: np.trace(m)
Out[2]: 15

In [3]: m_tr=0
   ...: for i in range(m.shape[0]):
   ...:     m_tr=m_tr+m[i, i]
   ...: m_tr
Out[3]: 15
```

2.2.5 Diagonal matix

A matrix with zero elements other than the main diagonal elements of a square matrix is called a diagonal matrix. Of course, a zero matrix with a principal diagonal element of zero and all elements equal to zero is also a diagonal matrix. The identity matrix is a representative diagonal matrix.

Diagonal matrices can be created using the np.diag() function.

```
In [1]: print(np.diag([1,2,3]))
[[1 0 0]
 [0 2 0]
 [0 0 3]]

In [2]: print(np.eye(3))
[[1. 0. 0.]
 [0. 1. 0.]
 [0. 0. 1.]]
```

2.2.6 Triangular matrix

The triangular matrix can be divided into a lower triangular matrix and an upper triangular matrix based on the diagonal elements of the matrix.

Lower triangular matrix
• A matrix in which all the elements above the principal diagonal elements are zero.
• Using np.tril()

Upper triangle matrix
• A matrix in which all the elements below the main diagonal elements are zero.
• Using np.triu()

```
In [1]: x=np.random.randint(1, 10, 9)\
   ...:     .reshape((3,3))
   ...: print(x)
[[5 7 9]
 [6 5 6]
 [1 7 9]]

In [2]: print(np.triu(x))
[[5 7 9]
 [0 5 6]
 [0 0 9]]

In [3]: print(np.triu(x, k=1))
[[0 7 9]
 [0 0 6]
 [0 0 0]]

In [4]: print(np.tril(x))
[[5 0 0]
 [6 5 0]
 [1 7 9]]
```

The triangular matrix has the following characteristics.

The transposed matrix of the lower triangular matrix becomes the upper triangular matrix and vice versa.

```
In [5]: tu=np.triu(x)
   ...: print(tu)
[[5 7 9]
 [0 5 6]
 [0 0 9]]

In [6]: print(tu.T)
[[5 0 0]
 [7 5 0]
 [9 6 9]]
```

If the diagonal elements of the triangular matrix are all 0, it becomes an irreversible matrix.

An irreversible matrix is a matrix that does not have an inverse matrix , and its determinant is 0.

```
In [7]: x1=np.triu(x, k=1)
   ...: print(x1)
[[0 7 9]
 [0 0 6]
 [0 0 0]]

In [8]: la.det(x1)
Out[8]: 0.0
```

The inverse matrix from the upper triangle matrix and the lower triangle matrix are the upper triangle matrix and the lower triangle matrix, respectively.

```
In [9]: print(tu)
[[5 7 9]
 [0 5 6]
 [0 0 9]]

In [10]: tu_inv=np.around(la.inv(tu), 3)
   ...: print(tu_inv)
```

```
[[ 0.2   -0.28  -0.013]
 [ 0.     0.2   -0.133]
 [ 0.     0.     0.111]]
```

2.2.7 Symmetric matrix

It is a square matrix that shows a symmetric relationship with respect to the main diagonal element as in Eq. 2.5.

$$\begin{bmatrix} a_{11} & b & c \\ b & a_{22} & d \\ c & d & a_{33} \end{bmatrix} \quad \text{(Eq. 2.5)}$$

A symmetric matrix can be created as the sum of the upper or lower triangular matrix and the transpose matrix of each matrix.

Using numpy's triu(A) and tril(A) functions, you can create an upper and lower triangular matrix based on matrix A.

```
In [1]: x=np.random.randint(-10, 10, (3,3))
   ...: y=np.random.randint(-10, 10, (3,3))
   ...: A=np.triu(x)+np.triu(x).T
   ...: print(A)
[[ -4  -5  -7]
 [ -5 -14  -6]
 [ -7  -6  18]]

In [2]: B=np.tril(y)+np.tril(y).T
   ...: print(B)
[[-18  -4   5]
 [ -4  12  -5]
 [  5  -5  18]]
```

The following relationship holds between the symmetry matrices A , B and scalar k .

The symmetric matrix and its transpose matrix are the same. $A=A^T$

```
In [3]: print(A==A.T)
[[ True  True  True]
 [ True  True  True]
 [ True  True  True]]
```

The two symmetric matrices $A \pm B$ are symmetric matrices.

```
In [4]: print(A+B)
[[-22  -9  -2]
 [ -9  -2 -11]
 [ -2 -11  36]]
```

The scalar product of a symmetric matrix is also a symmetric matrix.

```
In [5]: print(3*A)
[[-12 -15 -21]
 [-15 -42 -18]
 [-21 -18  54]]
```

In general, the product of two symmetric matrices is not a symmetric matrix, but Eq. 2.6 holds for the following special cases.

$$
\begin{aligned}
&AB = BA \\
&\Rightarrow (AB)^T = B^T A^T \\
&\qquad = BA \\
&\qquad = AB \quad \text{(Eq. 2.6)}
\end{aligned}
$$

```
In [1]: A=np.array([[1,2],[2,3]])
   ...: B=np.array([[-4,3],[3,-1]])
   ...: AB=np.dot(A, B)
   ...: BA=np.dot(B,A)
   ...: print(AB==BA)
[[ True  True]
 [ True  True]]
```

```
In [2]: Abt=(np.dot(A, B)).T
   ...: BtAt=np.dot(B.T, A.T)
   ...: print(Abt==BtAt)
[[ True  True]
 [ True  True]]

In [3]: print(Abt==AB)
[[ True  True]
 [ True  True]]
```

If A is a reversible (matrix having an inverse matrix) symmetric matrix, then A^{-1} is also a symmetric matrix.

```
In [1]: A=np.array([[-10, -1, 3],
   ...:             [ -1, 6, 1],
   ...:             [ 3, 1, -6]])
   ...: la.det(A)
Out[1]: 316.00000000000006

In [2]: A_inv=np.around(la.inv(A), 2)
   ...: print(A_inv)
[[-0.12 -0.01 -0.06]
 [-0.01  0.16  0.02]
 [-0.06  0.02 -0.19]]
```

2.3 Inverse matrix and determinant

2.3.1 Inverse matrix

When performing matrix multiplication on a matrix as in Eq. 2.7, the matrix that returns itself is called the identity matrix(I).

$$A \cdot B = A \rightarrow B = I \quad \text{(Eq. 2.7)}$$

An identity matrix is a square matrix with the same number of rows and columns, and can be created by the np.eye() function.

```
In [1]: print(np.eye(3))
[[1. 0. 0.]
 [0. 1. 0.]
 [0. 0. 1.]]
```

If both matrices satisfy Eq. 2.8, matrix B is the inverse matrix of A and is represented by A^{-1} .

$$A \cdot B = I \rightarrow B = A^{-1} \quad \text{(Eq. 2.8)}$$

A matrix with an inverse matrix is called an invertible matrix and can be calculated by the np.linalg.inv() function.

On a computer, operations are performed in binary, so the binary value closest to the value passed by the decimal number is converted and returned. In many real numbers, decimal and binary values do not match. For example, false returned in the following cases is an unexpected result:

```
In [1]: 0.1+0.2 == 0.3
Out[1]: False
```

The function `Decimal()` in Python's decimal module returns the closest number computed by the computer for any decimal number passed.

```
In [1]: import decimal
   ...: decimal.Decimal(0.1)
Out[1]: Decimal('0.100...')
```

These problems cannot be fundamentally solved. Instead, you can simplify it to rounding, conversion to strings, etc.

`place (object, condition, value)` function of the numpy module can also be used for this purpose. This function converts the specified value to the specified condition. In the following code, the function is used to convert values below 10^{-10} to 0.

```
In [1]: a=np.array([[1,3,-5],
   ...:             [-2,7,8],
   ...:             [4,0,6]])
   ...: a_inv=la.inv(a)
   ...: np.around(a_inv, 2)
   ...: re=np.dot(a, a_inv)
   ...: print(re)
[[ 1.00000000e+00  2.08166817e-17 -6.93889390e-17]
 [ 1.11022302e-16  1.00000000e+00  0.00000000e+00]
 [ 5.55111512e-17 -1.38777878e-17  1.00000000e+00]]

 # Substitute 1e-10 with 0
 # for the above result
In [2]: np.place(re, re<1e-10, 0)
   ...: print(re)
[[1. 0. 0.]
 [0. 1. 0.]
 [0. 0. 1.]]
```

The inverse matrix is used to calculate the solution of the simultaneous equation. For example, the following simultaneous equation can be represented as a matrix system.

$$\begin{aligned} x+y+2z &= 9 \\ 2x+4y-3z &= 1 \\ 3x+6y-5z &= 0 \end{aligned} \rightarrow \begin{bmatrix} 1 & 1 & 2 \\ 2 & 4 & -3 \\ 3 & 6 & -5 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} 9 \\ 1 \\ 0 \end{bmatrix}$$

As shown above, the matrix system consists of a matrix product between the unknown vector b and the coefficient matrix A corresponding to each unknown, and the resulting constant vector c .

If an inverse matrix of coefficient matrices exists, the solution of each unknown can be calculated by the process of Eq. 2.9.

$$A \cdot b = c \rightarrow A^{-1} \cdot A \cdot b = A^{-1} \cdot c$$
$$\rightarrow b = A^{-1} \cdot c \quad \text{(Eq. 2.9)}$$

```
In [1]: A=np.array([[1,1,2],
   ...:             [2,4,-3],
   ...:             [3,6,-5]])
   ...: c=np.array([[9],[1],[0]])
   ...: A_inv=la.inv(A)
   ...: re=np.dot(a_inv, c)

In [2]: print(np.around(re,3))
[[ 1.146]
 [ 1.344]
 [-0.764]]
```

The above code is calculated according to the process in Eq. 2.9. When the coefficient matrix is reversible, you can apply the function solve (A, b) of the numpy.linalg module with the algorithm of Eq. 2.9.

```
In [3]: print(la.solve(A, c))
[[1.]
 [2.]
 [3.]]
```

The solution to the above equations is unique. That is, the three equations meet at one coordinate. This state is called linear independence . Unlike these results, the state in which equations intersect at multiple coordinates is called linear dependence .

The above process is based on the assumption that the coefficient matrix is a square matrix and an inverse matrix exists. As a result, there is a unique solution. This result can be said to have an inverse matrix only when there is a single set of solutions to the unknowns in the equation.

Unlike the above, when the number of unknowns and the number of equations are different, the coefficient matrix is not a square matrix. In this case, the inverse matrix cannot be calculated by the above process. Instead, the Gaussian Jordan elimination method can be used to compute the inverse matrix. This elimination method returns a row echelon form matrix.

2.3.1.1 Reduced row echelon form(rref)

The general form of the row echelon form (ref) is as follows. A case where the first element other than 0 in each row is 1 is called reduced row echelon form (rref).

$$\begin{bmatrix} 0 & x_1 & x_2 & x_3 & \cdots \\ 0 & 0 & x_4 & x_5 & \cdots \\ 0 & 0 & 0 & x_6 & \cdots \\ \vdots & \vdots & \vdots & \vdots & \vdots \end{bmatrix}$$

A matrix can be converted to rreff through operations between rows. This method is called Gaussian-Jordan elimination.

The matrix's rref is converted to a simple form while retaining its unique characteristics. Therefore, it is a form of matrix that is very useful for inferring the properties of the matrix. In the following example, rref is applied to calculate the solution of a system of equations with a coefficient matrix rather than a square matrix.

$$\begin{aligned} x + y &= 2 \\ 2x + 4y &= -3 \\ 3x + 6y &= 5 \end{aligned}$$

The next matrix shows the above system of equations in matrix form. The colon in the matrix is used to distinguish the coefficient matrix from the

constant vector. The form of combining a coefficient matrix and a constant vector is called an augment matrix .

$$\left[\begin{array}{ccc:c} 1 & 1 & : & 2 \\ 2 & 4 & : & -3 \\ 3 & 6 & : & 5 \end{array}\right]$$

The left side of the colon in the augment matrix can be made an identity matrix using Gaussian-Jordan elimination. The right side of the colon is the value corresponding to each term on the left side. As a result, the values are the solutions of each unknown x, y. The following code is the calculation process that makes the left side of the augment matrix an identity matrix.

```
In [1]: a=np.array([[1,1,2],
   ...:             [2,4,-3],
   ...:             [3,6,5]])
   ...: a[1,:]=a[1,:]-2*a[0,:]
   ...: a[2,:]=a[2,:]-3*a[0,:]
   ...: a[1,:]=a[1,:]/2
   ...: a[0,:]=a[0,:]-a[1,:]
   ...: a[2,:]=a[2,:]-3*a[1,:]
   ...: a[2,:]=a[2,:]/8
   ...: a[0,:]=a[0,:]-5*a[2,:]
   ...: a[1,:]=a[1,:]+3*a[2,:]

In [2]: print(a)
[[1 0 0]
 [0 1 0]
 [0 0 1]]
```

Row 3 of the above result is 0x + 0y = 1 and cannot be established. As shown in Fig. 2.1, there are no coordinates where all three equations meet.

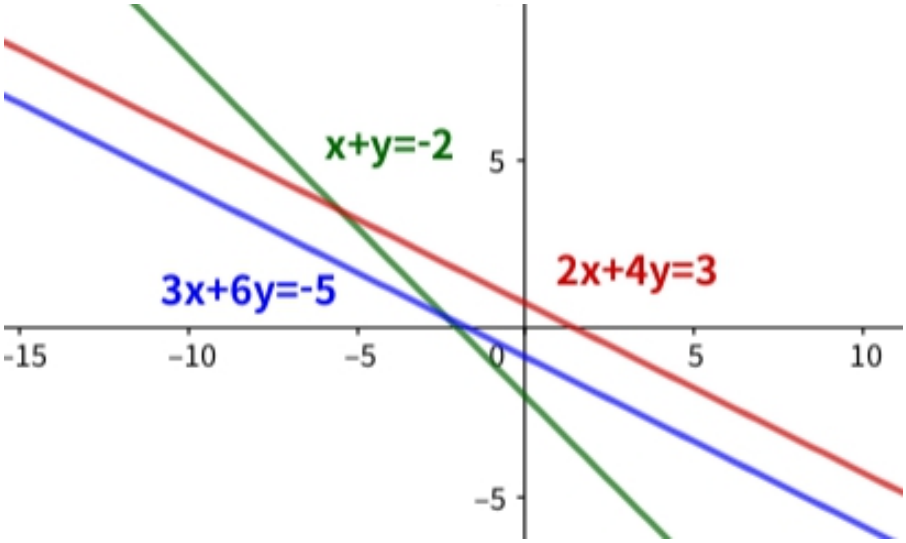


Fig. 2.1 Three straight lines without intersections.

There is no numpy function to compute the matrix rref, but you can use the function rref() from the Python library sympy. The sympy module also uses numpy's array object as its base type, so they are compatible with each other.

As a result, the function returns the matrix of rref and the number of columns with diagonal element 1 in the form of a tuple. Import sympy to use this library.

```
In [1]: from sympy import *
   ...: a=np.array([[1,1,2],
   ...:             [2,4,-3],
   ...:             [3,6,5]])
   ...: a1=Matrix(a)

In [2]: print(a1)
Matrix([[1, 1, 2], [2, 4, -3], [3, 6, 5]])

In [3]: print(a1.rref())
(Matrix([
[1, 0, 0],
[0, 1, 0],
[0, 0, 1]]), (0, 1, 2))
```

rref has the following characteristics.

1. As shown in Eq. 2.10, if the first nonzero element in each row is 1, this is called leading 1. This position is called the pivot position . The pivot column is the column containing the pivot position. Variables that correspond to this column are called leading variables , and variables that correspond to columns that are not pivot columns are called free variables .

$$\begin{bmatrix} 1(\text{leading}) & 0 & 0 & 0 \\ 0 & 1(\text{leading}) & 0 & 0 \\ 0 & 0 & 1(\text{leading}) & 3(\text{free}) \end{bmatrix} \quad \text{(Eq. 2.10)}$$

2. If all elements in row k are non-zero, the number of zeros in front of the leading 1 in row k + 1 is less than the number of zeros in row k.
3. If the elements are all zeros, all elements in the row immediately before that are not zero.
4. All remaining elements in the row containing leading 1 are zero.

Among the above characteristics, the form that satisfies 1, 2, and 3 is a row echelon form(ref), and through this form, a reduced row echelon form(rref) is created. In the following example, the solution is calculated using the rref.

$$\begin{aligned} a + b - c + 3d &= 0 \\ 3a + b - c - d &= 0 \\ 2a - b - 2c + d &= 0 \end{aligned}$$

```
In [1]: a=np.array([[1,1,-1,3, 0],
   ...:             [3,1,-1, -1, 0],
   ...:             [2,-1,-2, 1, 0]])
   ...: print(a)
[[ 1  1 -1  3  0]
 [ 3  1 -1 -1  0]
 [ 2 -1 -2  1  0]]

In [2]: a1=Matrix(a)
   ...: print(a1.rref())
```

```
(Matrix([
[1, 0, 0,    -2, 0],
[0, 1, 0,   5/3, 0],
[0, 0, 1, -10/3, 0]]), (0, 1, 2))
```

The above results are organized as follows.

$$\begin{pmatrix} a - d = 0 \\ b + 5/3d = 0 \\ c - 10/3d = 0 \end{pmatrix} \rightarrow \begin{pmatrix} a = d \\ b = -5/3d \\ c = 10/3d \end{pmatrix}$$

According to the above results, there are various a , b , and c values depending on the d value.

→In the above results, the pivot columns are columns 1, 2, and 3. In other words, there are 3 leading variables corresponding to the pivot column and 1 free variable. This result means that the values of the leading variable depend on the free variable.

When free variables exist, there are many solutions.

2.3.2 Determinant

The existence of an inverse matrix means that a unique solution set exists in the system of equations. That is, the existence of an inverse matrix for a coefficient matrix is a necessary condition that can have a unique solution. Conversely, if the inverse matrix of the coefficient matrix does not exist, various solutions exist or no solution exists. A matrix without an inverse matrix is called a singular matrix .

The matrix has a property that the inverse matrix does not exist when the determinant is 0. In other words, the determinant is the basis for determining whether a matrix is singular.

The determinant is represented as det(matrix) or |matrix| .

The determinant of a 2×2 matrix can be easily calculated as follows.

$$A = \begin{bmatrix} a & b \\ c & d \end{bmatrix} \Rightarrow \det(A) = |A| = |ad-bc|$$

The inverse matrix of a higher dimensional matrix can be computered using the minor and cofactor methods . This method is cumbersome by repeating the process of extracting a 2x2 submatrix from a matrix and calculating the determinant of the extracted matrix. Instead, the determinant of all square matrices can be calculated using the det() function of the np.linalg module.

```
In [1]: a=np.array([[1,1,2],
   ...:             [2,4,-3],
   ...:             [3,6,-5]])
   ...: print(a)
[[ 1  1  2]
 [ 2  4 -3]
 [ 3  6 -5]]

In [2]: a_inv=la.inv(a)
   ...: np.place(a_inv, a_inv < 1e-10, 0)
   ...: print(a_inv)
[[ 2.  0. 11.]
 [ 0. 11.  0.]
 [ 0.  3.  0.]]

In [3]: a_det=la.det(a)
   ...: np.round(a_det, 10)
Out[3]: -1.0
```

The determinant of a square matrix has the following characteristics

The determinant of the square matrix A and its transpose matrix A^T is the same.

$$\det(A) = \det(A^T)$$

```
In [1]: A=np.array([[1,2,3],
   ...:             [-4,4,6],
   ...:             [7,-8,9]])
   ...: re=np.around([la.det(A), la.det(A.T)], 4)
   ...: print(re)
[252. 252.]
```

If all the elements in one row or column of square matrix A are 0, then the following holds:

$$\det(A) = 0$$

```
In [2]: A[2,:]=[0,0,0]
   ...: print(A)
[[ 1  2  3]
 [-4  4  6]
 [ 0  0  0]]

In [3]: la.det(A)
Out[3]: 0.0
```

The following relationship between the square matrix A and B is established.

$$B[row, :]=k \times A[row, :] \rightarrow \det(B)=k \times \det(B)$$

```
In [1]: A=np.array([[2, 9, 5],
   ...:         [2, 9, 7],
   ...:         [7, 2, 8]])
   ...: B=np.array(A, copy=True)
   ...: B[0,:]=2*B[0,:]
   ...: print(np.around([la.det(A), la.det(B)], 4))
[118. 236.]
```

If the two rows of square matrix A are proportional,

$$\det(A) = 0.$$

```
In [1]: A=np.array([[2, 9, 5],
   ...:         [2, 9, 7],
   ...:         [4, 18, 14]])
   ...: print(A)
[[ 2  9  5]
 [ 2  9  7]
 [ 4 18 14]]

In [2]: la.det(A)
Out[2]: 0.0
```

If the square matrix A is a triangular matrix, the determinant is equal to the product of the principal diagonal elements.

```
In [1]: A=np.array([[1,6, 10],
   ...:         [0, 5, 9],
   ...:         [0,0, 12]])
   ...: print(A)
[[ 1  6 10]
 [ 0  5  9]
 [ 0  0 12]]

In [2]: np.round(la.det(A), 4)
Out[2]: 60.0
```

• The necessary sufficient condition for the square matrix A to be the reversible matrix is det (A) ≠ 0.
• The following relationship is established in two square matrices A and B of the same dimension.

$$\det(AB) = \det(A)\cdot\det(B)$$

```
In [1]: A=np.random.randint(10, size=(4,4))
   ...: B=np.random.randint(10, size=(4,4))
   ...: la.det(np.dot(A,B))
Out[1]: 104219.9...

In [2]: la.det(A)*la.det(B)
Out[2]: 104220.0...
```

If the square matrix A is a reversible matrix, the following holds:

$$\det(A)=\frac{1}{\det(A)}$$

```
In [3]: np.around([la.det(la.inv(A)),
   ...:            1/la.det(A)], 4)
Out[3]: array([-0.0052, -0.0052])
```

The determinant represents the area of the shape formed by the matrix.

```
In [1]: A=np.random.randint(5, size=(2,2))
...: print(A)
[[3, 0],
 [2, 1]])

In [2]: la.det(A)
Out[2]: 3.0000000000000004
```

Fig. 2.2 shows the shape produced by matrix a above.

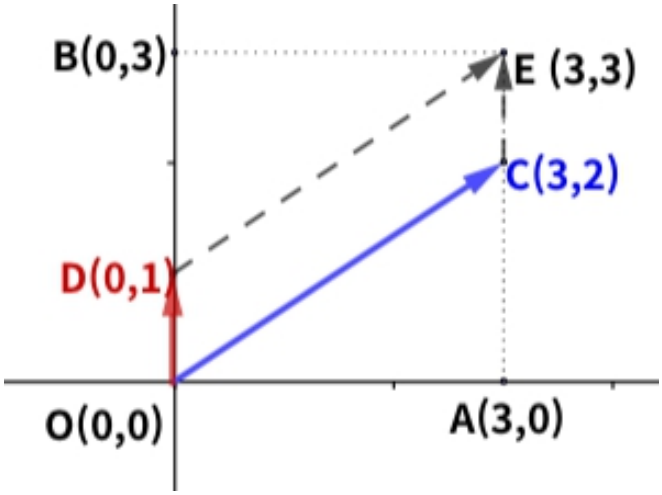


Fig. 2.2 Shape produced by matrix A.

From Fig. 2.2,

☐ CODE = ☐ AOBE - △ AOC - △ DBE

⇒ 9-(3×2×2)/2 = 3

2.4 LU decomposition

The reversible square matrix(non-specific square matrix) can be decomposed into a lower triangular matrix and an upper triangular matrix , and this decomposition method is called LU decomposition. In mathematics, a process called prime factor decomposition is applied to lower the higher-order equation to a smaller order than the original equation. These results make the characteristics of the expression more clear. Decomposition in vectors and matrices is also performed to better understand each characteristic. In other words, decomposition is a very useful way to understand the properties of the matrix.

Consider the following equation with a reversible matrix, A , as the coefficient matrix.

```
Ax = b
```

The coefficient matrix A can be expressed as the matrix product of the lower and upper triangular matrices as shown in Eq. 2.11.

```
A = LU     (Eq. 2.11)
```

When a matrix is decomposed into small units as shown in Eq. 2.11, the decomposed matrices can vary. This decomposition can be recognized as the process of factoring higher order equations. For example, quadratic equations can be decomposed in a variety of ways, such as cubic and linear, quadratic and quadratic. Like decomposition of these equations, matrix decomposition can also be expressed in several forms.

LU decomposition of matrix A is performed. Let's first determine whether the reversible matrix is.

```
In [1]: A=np.array([[1,-2,3],
   ...:         [2,-5,12],
   ...:         [0,2,-10]])
   ...: print(A)
[[  1  -2   3]
 [  2  -5  12]
```

```
 [  0   2 -10]]

In [2]: la.det(A)
Out[2]: -2.0
```

According to the above results, it can be seen that the determinant of matrix A is not 0, so it is a reversible matrix. The upper triangular matrix of the above matrix applies the np.triu() function. This matrix is used to calculate the lower triangular matrix in the next step.

$$A = LU \rightarrow AU^{-1} = LUU^{-1} = LI = L$$

```
In [3]: Au=np.triu(A)
   ...: print(Au)
[[  1  -2   3]
 [  0  -5  12]
 [  0   0 -10]]

In [4]: Al=np.dot(A, la.inv(Au))
   ...: print(np.around(Al, 2))
[[ 1.    0.    0.  ]
 [ 2.    0.2  -0.36]
 [ 0.   -0.4   0.52]]
```

The results are as follows. In the following code, np.allclose() is applied to determine whether two objects are equal.

```
In [5]: np.allclose(A, np.dot(Al, Au))
Out[5]: True
```

LU decomposition can be calculated using linalg.lu() in the scipy library. This function should consider the permutation matrix (P) as shown in Eq. 2.12.

A permutation matrix is simply a matrix that causes a row exchange of results. Therefore, the application of this matrix causes a change in the direction or position of each of the upper and lower triangle matrices, but does not change their essential meaning.

A = PLU (Eq. 2.12)

A: Original Matrix

P: Permutation Matrix

L: lower triangular matrix

U: upper triangular matrix

```
In [6]: from scipy import linalg
   ...: p, l, u=linalg.lu(A)
   ...: print(p)
[[0. 0. 1.]
 [1. 0. 0.]
 [0. 1. 0.]]

In [7]: print(l)
[[1.   0.   0.  ]
 [0.   1.   0.  ]
 [0.5  0.25 1.  ]]

In [8]: print(u)
[[  2.   -5.   12. ]
 [  0.    2.  -10. ]
 [  0.    0.   -0.5]]

In [9]: np.allclose(A, np.dot(np.dot(p,l),u))
Out[9]: True
```

3 Linear System

3.1 Linear Combination

If Eq. 3.1 is defined between vector set R = $\{v_1, v_2, \ldots, v_n\}$ and the scalar set c=$\{c_1, c_2, \ldots, c_n\}$, the expression is called linear combination .

$$y = c_1v_1+c_2v_2+\ldots+c_nv_n = \begin{bmatrix} v_1 & v_2 & \cdots & v_n \end{bmatrix} \begin{bmatrix} c_1 \\ c_2 \\ \vdots \\ c_n \end{bmatrix} \quad \text{(Eq. 3.1)}$$

R: The coefficient matrix of linear combination is called the standard matrix .

Scalar(weight): $c_1, c_2, \ldots, c_n$

For example, if Eq. 3.2 is established between the weight vectors w=<x, y> and the three vectors a_1 , a_2 , b , it is called a linear combination.

$$b = x \cdot a_1 + y \cdot a_2 \quad \text{(Eq. 3.2)}$$

```
In [1]: a1=np.array([1,-2,-5]).reshape(-1,1)
   ...: a2=np.array([2,5,6]).reshape(-1,1)
   ...: b=np.array([7,4,-3]).reshape(-1,1)
   ...: A=np.hstack([a1, a2])
   ...: print(A)
[[ 1  2]
 [-2  5]
 [-5  6]]

In [2]: print(b)
[[ 7]
```

```
[ 4]
[-3]]
```

The above result can be represented as:

$$\begin{bmatrix} 7 \\ 4 \\ -3 \end{bmatrix} = \begin{bmatrix} 1 & 2 \\ -2 & 5 \\ -5 & 6 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix}$$

Since the standard matrix(coefficient matrix) in the above equation is not a square matrix, the solution set can be calculated by applying reduced row echelon form (rref). That is, the rref() function is applied to the augment matrix, which is the combination of the standard matrix A , and the constant vector b .

```
In [3]: A_b=np.hstack([A, b])
   ...: print(A_b)
[[ 1  2  7]
 [-2  5  4]
 [-5  6 -3]]

In [4]: Matrix(A_b).rref()
Out[4]:
(Matrix([
 [1, 0, 3],
 [0, 1, 2],
 [0, 0, 0]]),
 (0, 1))
```

Fig. 3.1 shows that the three straight lines from the matrix system intersect at coordinates (3, 2). That is, since the above equations have a unique solution, a linear combination is established.

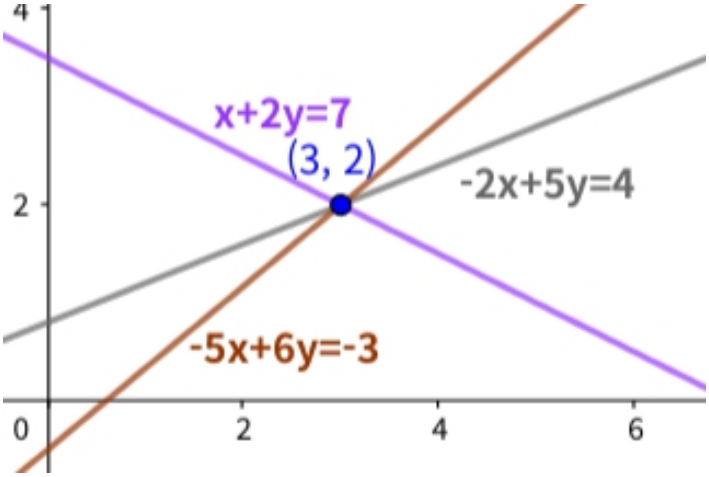


Fig. 3.1 The intersection of three straight lines.

The above example can represent vector b by proper adjustment of the vectors a_1, a_2, the column vectors of matrix A. In other words, it can be said that b exists in the space formed by a_1 and a_2. This relationship between a_1, a_2, and b can be represented using the term Span.

$$b=\text{Span}\{a_1, a_2\}$$

(Ex 3.1) Investigate the linear combination between the following vectors.

```
In [1]: a1=np.array([1,-2,3]).reshape(-1,1)
   ...: a2=np.array([5,-13, -3]).reshape(-1,1)
   ...: b=np.array([-3,8,1]).reshape(-1,1)
   ...: A_b=np.hstack([A, b])

In [2]: print(A_b)
[[ 1  2 -3]
 [-2  5  8]
 [-5  6  1]]

In [3]: Matrix(A_b).rref()
Out[52]:
(Matrix([
 [1, 0, 0],
 [0, 1, 0],
 [0, 0, 1]]),
 (0, 1, 2))
```

The 3rd row 0x + 0y = 1 of the above result cannot be established. As shown in Fig. 3.2, there is no intersection of three straight lines. In other words, since there is no solution of this system, there is no span between the three vectors. Such a system is called an inconsistent system .

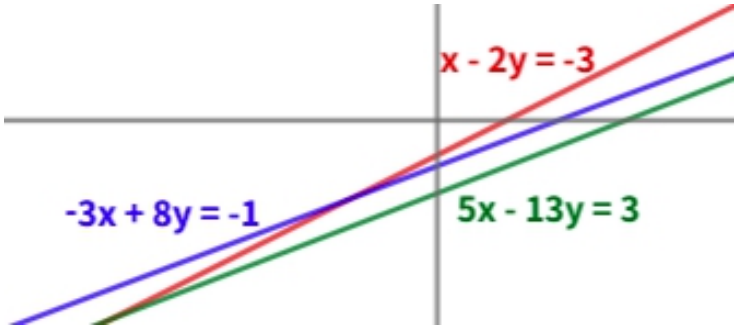


Fig. 3.2 Inconsistent system.

From the above process, we can define the following for the linear combination Ax = b of standard matrix A in m × n dimension, variable vector x, and constant vector b .

There is a set of solutions for the matrix system.
⇔ The column vectors constituting the standard matrix A form a linear combination with the constant vector b.
⇔ If the column vector of A is a subspace of R, b is also a subspace of R.

- $a_1 , a_2 , \ldots, a_n \subset R \Leftrightarrow b \subset R$, $R=\text{Span}\{a_1 , a_2 , \ldots, a_n \}$
- column vector of $A = a_1 , a_2 , \ldots, a_n$

The unique solution set for that linear combination exists.
⇔ A has a pivot position in every row.
⇔ There are no free variables.
⇔ An inverse matrix exists.

(Ex. 3.2) Calculate the solution set of Ax = b.

A=[<1,5,-2,0 >, <-3,1, 9,-5>, <4, -8, -1, 7>]
b=<-7, 9, 0>

```
In [1]: A=np.array([[1,5,-2,0],
   ...:         [-3,1, 9,-5],
   ...:         [4, -8, -1, 7]])
   ...: b=np.array([-7,9,0]).reshape(3,1)
   ...: A_b=np.hstack([A, b])

In [2]: print(A_b)
[[ 1  5 -2  0 -7]
 [-3  1  9 -5  9]
 [ 4 -8 -1  7  0]]

In [3]: Matrix(A_b).rref()
Out[3]:
(Matrix([
 [1, 0, 0,  8/7, -11/7],
 [0, 1, 0, -2/7,  -6/7],
 [0, 0, 1, -1/7,   4/7]]),
 (0, 1, 2))
```

The rref of the above result has three pivot columns . That is, since it has 3 leading variables and 1 free variable, it can have various solutions depending on the value of the free variable.

3.1.1 Homogeneous Linear Combination

Linear combination with a constant vector of 0 as in Eq. 3.3 is called homogeneous linear combination.

$$Ax = 0 \quad \text{(Eq. 3.3)}$$

Homogeneous linear combinations can be divided into two types according to a set of solutions.

- Trivial solution : Homogeneous linear combination with unique solution set
- Nontrivial solution : Homogeneous linear combination with various solution sets

Therefore, if homogeneous linear combination has at least one free variable, it has a nontrivial solution. In other words, it has a lot of solutions.

(Ex. 3.3) The following is a homogeneous linear combination. Compute solution set(s) for this combination.

$< 3x_1 + 5x_2 - 4x_3 = 0, -3x_1 - 2x_2 + 4x_3 = 0, 6x_1 + x_2 - 8x_3 = 0 >$
→ Augment Matrix=[<3, -3, 6>, <5,-2,1>, <-4,4,-8>, <0,0,0>]

```
In [1]: au=np.array([[3,5,-4,0],
   ...:              [-3,-2,4,0],
   ...:              [6,1,-8,0]])
   ...: Matrix(au).rref()
Out[1]:
(Matrix([
 [1, 0, -4/3, 0],
 [0, 1,    0, 0],
 [0, 0,    0, 0]]),
 (0, 1))
```

The rref of the above result contains 2 leading variables and 1 free variable. That is, it can have various solutions depending on the variable x_3 . It has nontrivial solution.

3.2 Linear independence and dependence

The homogeneous linear combination between R space vectors(v_1 , v_2 , …, v_p) and scalar(c_1 , c_2 ,…, c_p) can be expressed in the form of a matrix equation like Eq. 3.4.

$$c_1v_1 + c_2v_2 + \cdots + c_pv_p = 0$$

$$\rightarrow \begin{bmatrix} v_{11} & v_{12} & \cdots & v_{1p} \\ v_{21} & v_{22} & \cdots & v_{2p} \\ \vdots & \vdots & \ddots & \vdots \\ v_{n1} & v_{n2} & \cdots & v_{np} \end{bmatrix} \begin{bmatrix} c_1 \\ c_2 \\ \vdots \\ c_p \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ \vdots \\ 0 \end{bmatrix} \quad \text{(Eq.3.4)}$$

A homogeneous linear combination is said to be linear independent if it has a trivial solution , and linear dependent if it has a nontrivial solution to satisfy the above equation.

(Ex. 3.5) Determine the linear independence and linear dependence of the following system of equations.

$v_1 =< 1,2,3 >$
$v_2 =< 4,5,6 >$
$v_3 =< 2,1,0 >$

```
In [1]: v1=np.array([[1],[2],[3]])
   ...: v2=np.array([[4],[5],[6]])
   ...: v3=np.array([[2],[1],[0]])
   ...: A=np.hstack([v1,v2,v3])
   ...: print(A)
[[1 4 2]
 [2 5 1]
 [3 6 0]]
```

```
In [2]: np.round(la.det(A), 4)
Out[2]: -0.0
```

Matrix A is a 3 × 3 square matrix, so we can compute the determinant. The result is 0, so an inverse matrix cannot exist. This means that there is no unique solution. To confirm this, try applying rref.

```
In [3]: au=np.hstack([A, b])
   ...: print(au)
[[ 1  4  2 -7]
 [ 2  5  1  9]
 [ 3  6  0  0]]

In [4]: Matrix(A).rref()
Out[4]:
(Matrix([
 [1, 0, -2],
 [0, 1,  1],
 [0, 0,  0]]),
 (0, 1))
```

According to the above results, there are 2 leading variables and 1 free variable. In other words, it can be represented as:

$x_1 = 2x_3$, $x_2 = -x_3$

The above system can have various solutions depending on the variable x_3 . As a result, it has a non-trivial solution, so it is a linear dependent.

(Ex. 3.6) Determine whether matrix B is linearly independent.

Matrix B is a square matrix, so if it is a reversible matrix, the set of functions can be computed with the np.linalge.solve() function.

Bx = c → np.linalg.solve(B, c)

B=[< 0,1,4 >, < 1,2,-1 >, < 5,8,0 >]
c=< 0,0,0 >

```
In [1]: B=np.array([[0,1,4],
   ...:             [1,2,-1],
   ...:             [5,8,0]])
   ...: print(B)
[[ 0  1  4]
 [ 1  2 -1]
 [ 5  8  0]]

In [2]: round(la.det(B), 3)
Out[2]: -13.0

In [3]: c=np.array([0,0,0]).reshape(-1,1)
   ...: print(c)
[[0]
 [0]
 [0]]

In [4]: print(la.solve(B,c))
[[ 0.]
 [ 0.]
 [-0.]]

In [5]: au=np.hstack([B, c])
   ...: Matrix(au).rref()
Out[5]:
(Matrix([
 [1, 0, 0, 0],
 [0, 1, 0, 0],
 [0, 0, 1, 0]]),
 (0, 1, 2))
```

The determinant of matrix B is not zero. This result means that there is an inverse matrix of B and there is a unique solution of linear combination by the matrix. These results confirm that there are no free variables in the rref for the augment matrix of the linear combination. In conclusion, the matrix above is linearly independent.

A linear system based on a standard matrix of square dimensions has the same number of unknowns and equations. Under these conditions, the existence of the inverse matrix of the standard matrix can determine whether the system is independent. Therefore, the results of the above examples can be summarized as follows.

If the standard matrix is square and reversible, it has a trivial solution and is linearly independent.

(Ex. 3.7) In the following example, the standard matrix is not a square matrix.

v_1 =< 0,2,-1,1 >
v_2 =< -3,1,4,-4 >
v_3 =< 9,-7,-5,2 >
c=< 0,0,0,0 >

```
In [1]: v1=np.array([[0],[2],[-1], [1]])
   ...: v2=np.array([[-3],[1],[4],[-4]])
   ...: v3=np.array([[9],[-7],[-5],[2]])
   ...: c=np.zeros([4,1])
   ...: V=np.hstack([v1,v2,v3])

In [2]: print(V)
[[ 0 -3  9]
 [ 2  1 -7]
 [-1  4 -5]
 [ 1 -4  2]]

In [3]: au=np.hstack([V, c])

In [4]: Matrix(au).rref()
Out[4]:
(Matrix([
 [1, 0, 0, 0],
 [0, 1, 0, 0],
 [0, 0, 1, 0],
 [0, 0, 0, 0]]),
 (0, 1, 2))
```

In this example, the linear combination by the 4 x 3 dimension standard matrix, V , has the following characteristics.

- # of row > # of column
- # of knowns = # of column

The matrix V is linearly independent because it has trivial solution, although equations in the system are redundant.

(Ex. 3.8) In the following example, the number of variables is greater than the number of expressions.

$v_1 =< 1,-2,-4 >$
$v_2 =< 4,-7,-5 >$
$v_3 =< -3,5,7 >$
$v_4 =< 0,1,5 >$
$c =< 0,0,0 >$

```
In [1]: v1=np.array([[1],[-2],[-4]])
   ...: v2=np.array([[4],[-7],[-5]])
   ...: v3=np.array([[-3],[5],[7]])
   ...: v4=np.array([[0],[1],[5]])
   ...: c=np.zeros([3,1])
   ...: V=np.hstack([v1,v2,v3,v4])
   ...: print(V)
[[ 1  4 -3  0]
 [-2 -7  5  1]
 [-4 -5  7  5]]

In [2]: au=np.hstack([V, c])
   ...: Matrix(au).rref()
Out[2]:
(Matrix([
 [1, 0, 0, -3.0, 0],
 [0, 1, 0,    0, 0],
 [0, 0, 1, -1.0, 0]]),
 (0, 1, 2)))
```

From the matrix V above, the variable vector x should be 4×1. That is, $x=<x_1, x_2, x_3, x_4>$. However, there are three expressions created for each row. Therefore, it does not have a trivial solution.

From the result (au) of the rref of the augment matrix, x_4 is a free variable. Therefore, other variables depend on x_4.

$x_1 = x_4,\ x_2 = 0,\ x_3 = x_4$

In conclusion, there is a nontrivial solution, so it is linearly dependent. The above results can be summarized as follows.

- # of row > # of column, and # of unknowns = # of column
  → trivial solution → independence
- # of row < # of column, and # of equation < # of known
  → non-trivial solution → dependence

4 Vector space

A linear combination can be established by arithmetic connection between vectors and scalars. As a result, the vector exists as part of the space where the vectors used in the computation exist. The space where vectors exist is called a vector space.

In the vector space by linear combination established as in Eq. 3.1 , various arithmetic rules are established between vector and scalar as follows.

- $u \in V, v \in V \rightarrow u + v \in V$
  If u and v are vectors, the sum of the two is a vector.
- $\alpha \in C, u \in V \rightarrow \alpha u \in V$
  If α is a scalar (C) and u is a vector (V), the product of these two, αu, is a vector.
- $u+v = v+u$
  The sum of the vectors constitutes a commutative property.
- $u+(v+w) = (u+v) +w$
  The sum of the vectors constitutes an associative property.
- There is a zero vector with all elements 0.
- $u+(-u)=0$
- $\alpha, \beta \in C, u \in V \rightarrow \alpha(\beta u)=(\alpha \beta) u$
- $\alpha \in C, u, v \in V \rightarrow \alpha (u+v)=\alpha u + \alpha v$
  In the product of scalar and vector, the distribution property holds.
- $\alpha, \beta \in C, u \in V \rightarrow (\alpha + \beta)u =\alpha u + \beta u$
  There is a distribution property between two scalars and a vector.
- $u \in V \rightarrow 1 \bullet u =u$

4.1 Subspace

In general, vector space means a set of vectors. By the various laws in the vector space introduced above, vectors and scalar operations can create new vectors. The generated vectors exist in the space formed by the operand vectors. In other words, the space of the result vector is called the subspace of the operand vectors. This subspace is a concept similar to a subset of sets.

For example, if V and W exist in vector space, and $W \subseteq V$, that is, W is a subset of V, then W exists in V's subspace (H). Subspace has three characteristics:

- The 0 vector in all vector spaces (V) becomes the subspace (H).
- $u, v \in H \rightarrow u + v \in H$
- $v \in H \rightarrow cv \in H$, u, v: vector, c: scalar

That is, if a linear combination between vectors is established, the resulting vector becomes the subspace of the original vector.

For example, if two vectors $u = <1, 1, 1, 0>$ and $v = <3, -1, 0, 1>$ satisfy Eq. 4.1, av + bu becomes the subspace of vectors u and v.

$u, v \in R \rightarrow au + bv \subset R$ (Eq. 4.1)
a, b: scalar, u, v: vector

(Ex. 4.1) Let's decide whether the linear combination of the following equations is established.

$$\begin{aligned} x-3y &= 0 \\ x-y &= 0 \\ x &= 0 \\ y &= 0 \end{aligned}$$

$$\rightarrow \begin{bmatrix} 1 & -3 \\ 1 & -1 \\ 1 & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$$

```
In [1]: A=np.array([[ 1, -3],
   ...:     [-1, 1],
   ...:     [ 1, 0],
   ...:     [ 0, 1]])
```

```
   ...: c=np.zeros([4,1])
   ...: au=np.hstack([A, c])

In [2]: print(au)
[[ 1. -3.  0.]
 [-1.  1.  0.]
 [ 1.  0.  0.]
 [ 0.  1.  0.]]

In [3]: Matrix(au).rref()
Out[3]:
(Matrix([
 [1, 0, 0],
 [0, 1, 0],
 [0, 0, 0],
 [0, 0, 0]]),
 (0, 1))
```

In the above result, there is no free variable, so a trivial solution exists. That is, the linear combination of vectors u and v holds. Therefore, if the result of the linear combination is subspace(H), H is the subspace of the vector space where the vectors u and v exist. This result can be represented as:

```
H = span{u, v}
```

The above results can be summarized as follows.

If v_1 , v_2 , …, v_p is in vector space, the result of linear combination by the vectors, H, is the subspace of the vector space and can be expressed as a span as follows.

$$H = \text{Span}\{v_1, v_2, \ldots, v_p\}$$

(Ex. 4.2) If the three vectors are all in the 3 dimension vector space v_1 , v_2 , v_3 , let's try to determine the value of h to be a subspace of the vector space.

The problem is to determine if there is a linear combination between the vectors. However, the vector y contains the unknown h. For operations involving unknowns, use the symbols() function from the sympy library. This function converts symbols to unknowns that can be calculated, so various operations involving unknowns are possible.

$v_1 = < 1, -2, -2 >$
$v_2 = < 15, -4, -7 >$
$v_3 = < -3, 1, 0 >$
$c = < -4, 3, h >$

```
In [1]: v1=np.array([1,-2,-2]).reshape(-1,1)
   ...: v2=np.array([5,-4, -7]).reshape(-1,1)
   ...: v3=np.array([-3, 1, 0]).reshape(-1,1)
   ...: V=np.hstack([v1,v2,v3])
   ...: print(V)
[[ 1  5 -3]
 [-2 -4  1]
 [-2 -7  0]]

In [2]: h=symbols("h")
   ...: y=Matrix([[-4], [3], [h]])
   ...: print(y)
Matrix([[-4], [3], [h]])

In [3]: au=Matrix(V).row_join(y)
   ...: print(au)
Matrix([[1, 5, -3, -4],
   [-2, -4, 1, 3],
   [-2, -7, 0, h]])

In [4]: au.rref()
Out[4]:
(Matrix([
 [1, 0, 0,      h/3 - 5/3],
 [0, 1, 0, 10/21 - 5*h/21],
 [0, 0, 1,   11/7 - 2*h/7]]),
 (0, 1, 2))
```

According to the above results, there are no free variables. That is, the linear combination of vectors is established regardless of the value of h. Therefore, we can represent y by vectors v_1, v_2, v_3, so those vectors are spans of y. In other words, it becomes a subspace of a 3 dimension vector space where three vectors exist.

$y = \text{Span}\{v_1, v_2, v_3\}$

$v_1, v_2, v_3 \subset R^3 \rightarrow y \in R^3$

The subspace of R^n satisfies one of the following:

- A set of all solutions in the homogenous linear combination
- Set of all linear combinations of specific vectors

(Ex. 4.3) If H is a set of all vectors that satisfy the following equation at 4-dimensional coordinates (a, b, c, d), check if H is a 4-dimensional subspace.

$$\begin{aligned} a-2b+5c-d &= 0 \\ -a-b+c &= 0 \end{aligned}$$

$$\rightarrow \begin{bmatrix} 1 & -2 & 5 & -1 \\ -1 & -1 & 1 & 0 \end{bmatrix} \begin{bmatrix} a \\ -b \\ c \\ d \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$$

```
In [1]: A=np.array([[1,-2,5,-1],[-1, -1, 1, 0]])
   ...: c=np.zeros([2,1])
   ...: au=np.hstack([A, c])
   ...: print(au)
[[ 1. -2.  5. -1.  0.]
 [-1. -1.  1.  0.  0.]]

In [2]: Matrix(au).rref()
Out[2]:
(Matrix([
 [1, 0,  1.0, -0.333333333333333, 0],
 [0, 1, -2.0,  0.333333333333333, 0]]),
 (0, 1))
```

The above result has two free variables. In other words, the variables a and b depend on the values of c and d , so they are linear dependents with non-trivial solutions. As a result, the above homogeneous linear combination is established, so the various vectors from variables a, b, c , and d are 4-dimensional subspace.

4.1.1 Dimension of subspace

Although the equations in Ex. 4.3 are four-dimensional subspaces, is the subspace dimension 4-dimensional? If the above linear bond is linearly independent, the subspace H will also be 4-dimensional.

The results can be summarized as follows.

$$
\begin{aligned}
&< a, b, c, d > \\
&= < -c+1/3 \cdot d, 2 \cdot c-1/3 \cdot d, c, d>
\end{aligned}
$$

The components of H, a, b, c and d are interdependent. In other words, a vector corresponding to a and b can form a linear combination with c or d . Each vector A_1 , A_2 , A_3 , or A_4 of standard matrix A in Ex. 4.3 corresponds to variables a, b, c , and d , respectively. Therefore, let's check whether the following linear combination is linearly independent.

$$
\begin{aligned}
&aA_1 + bA_2 = c \\
&aA_1 + bA_2 = d \\
&\\
&A_1 =< 1, -1 > \\
&A_2 =< -2, -1 > \\
&A_3 =< 5, 1 > \\
&A_4 =< -1, 0 >
\end{aligned}
$$

```
In [1]: A1=np.array([1,-1]).reshape(-1,1)
   ...: A2=np.array([-2,-1]).reshape(-1,1)
   ...: A3=np.array([5,1]).reshape(-1,1)
   ...: A4=np.array([-1,0]).reshape(-1,1)

In [2]: x=[A1, A1, A3, A4]
   ...: for i, j in zip(range(1,5), x):
   ...:     print(F"{'A'+str(i)}: \n {j}" )
A1:
 [[ 1]
 [-1]]
A2:
 [[ 1]
 [-1]]
A3:
 [[5]
 [1]]
A4:
 [[-1]
 [ 0]]
```

```
# A1·a+A2·b= A3
In [3]: A_123=Matrix(np.hstack([A1,A2,A3])).rref()
   ...: A_123
Out[3]:
(Matrix([
 [1, 0,  1],
 [0, 1, -2]]),
 (0, 1))

# A1·a+A2·b= A4
In [4]: A_124=Matrix(np.hstack([A1,A2,A4])).rref()
   ...: A_124
Out[4]:
(Matrix([
 [1, 0, -1/3],
 [0, 1,  1/3]]),
 (0, 1))
```

According to the above results, both linear bonds are independent. That is, among the four vectors A_1 , A_2 , A_3 , and A_4 , the basis vectors are A_1 and A_2 , so the span of the example can be expressed as follows.

$$H = Span\{A_1 , A_2 \}$$
$$= Span\{<1, -1>, <-1, -1> \}$$

As described above, vectors that can express all linear combinations as linear independence are called basis vectors , and are the same as the vector corresponding to the pivot column in the rref of all target vectors. In other words, Span is composed of basis vectors. Therefore, the number of vectors that make up the span becomes the dimension of the subspace. Therefore, the subspace H in the example above is two-dimensional.

- The pivot column becomes the basis vector.
- Span consists of a basis vector and the number of the vectors is the dimension of the subspace.

4.2 Basis

The basis of a part is the basic frame. The set of vectors was called a vector space. Two or more vectors in that space can be linearly combined to create subspaces within that space. The basis of these spaces are linearly independent vectors.

In summary:

> In the vector space of linear algebra, the basis is linearly independent vectors.

If an linear combination is linearly independent, the vector space can be expanded or contracted by the sum or product of some scalar in the system. Therefore, it is possible to convert to simple form without losing the unique properties of vectors such as the direction and dimension of vectors. This means that all the vectors in the vector space can be expressed based on the base vectors. As a result, the direction and dimension of the basis can be used as the base coordinates of the vectors contained in the space.

> If H is a subspace of vector space V and the following conditions are satisfied,
>
> $B=\{b_1 , b_2 , \ldots, b_p \}$ is the basis of H.
>
> - B is linearly independent
> - $H=\{b_1 , b_2 , \ldots, b_p \}$
>
> That is,the vectors of set B are subspace of H and
> a linear combination is established.

As a result, the vector B set becomes the span of H (subspace) and means that the resulting vector by linear combination of B becomes H. Also, since this linear combination must be independent, it must have a trivial solution .

4.2.1 Standard basis

Each column vector of the identity matrix is Span and linearly independent for all vectors of the same dimension. For example, in the case of a three-dimensional identity matrix, the solution set of homogeneous linear combinations by the matrix is a zero vector. In other words, the system has a trivial solution, so its linear combination is linear independence. Also, because the standard matrix of this system is square and independent, it has a non-zero determinant.

```
In [1]: I3=np.eye(3)
   ...: c=np.zeros([3,1])
   ...: Matrix(np.hstack([I3, c])).rref()
Out[1]:
(Matrix([
 [  1,   0, 0, 0],
 [0.0,   1, 0, 0],
 [0.0, 0.0, 1, 0]]),
 (0, 1, 2))

In [2]: la.det(I3)
Out[2]: 1.0
```

• Each column vector of the identity matrix is linearly independent of all vectors.
  This vector is called the standard basis .
• If the vectors show the identity matrix in the rref , they are the basis.

(Ex. 4.4) Determine whether the three vectors are the basis.

$V=\{< 3, -4, -2 >, < 0, 1, 1 >, < 6, 7, 3 >\}$

```
In [1]: v1=np.array([3,0,6]).reshape(-1,1)
   ...: v2=np.array([-4, 1, 7]).reshape(-1,1)
   ...: v3=np.array([-2,1,5]).reshape(-1,1)
   ...: V=np.hstack([v1,v2,v3])
   ...: print(V)
[[ 3 -4 -2]
```

```
 [ 0  1  1]
 [ 6  7  5]]

In [2]: round(la.det(V), 3)
Out[2]: -18.0

In [3]: c=np.zeros([3,1])
   ...: Matrix(np.hstack([V, c])).rref()
Out[3]:
(Matrix([
 [1, 0, 0, 0],
 [0, 1, 0, 0],
 [0, 0, 1, 0]]),
 (0, 1, 2))
```

According to the above result, the determinant of the standard matrix V generated by combining the three vectors is not 0, so it has a trivial solution. These results are confirmed from the rref of the augmented matrix of the homogeneous linear combination, as shown in the following code. All column vectors of the augmented matrix are pivot columns without any free variables . That is, since the system is linearly independent, all column vectors of the standard matrix are the basis vectors.

> The pivot column in matrix A is the basis vectors.

(Ex. 4.5) Let's decide if the following vectors are the basis.

v_1 =< 0, 2, -1 >
v_2 =< 2, 2, 0 >
v_3 =< 6, 16, -5 >

```
In [1]: v1=np.array([0, 2, -1]).reshape(-1,1)
   ...: v2=np.array([2,2,0]).reshape(-1,1)
   ...: v3=np.array([6, 16, -5]).reshape(-1,1)
   ...: V=np.hstack([v1,v2,v3])
   ...: print(V)
[[ 0  2  6]
 [ 2  2 16]
 [-1  0 -5]]

In [2]: round(la.det(V),3)
Out[2]: 0.0

In [3]: c=np.zeros([3,1])
```

```
    ...: Matrix(np.hstack([V, c])).rref()
Out[3]:
(Matrix([
 [  1, 0, 5.0, 0],
 [0.0, 1, 3.0, 0],
 [  0, 0,   0, 0]]),
 (0, 1))
```

The above vectors have a nontrivial solution because the determinant is 0 and a free variable exists in the rref type. In other words, the linear combination of the vectors cannot be the basis because it is linearly dependent. Looking at the result of the rref above, columns 1 and 2 are pivot columns, which represent v_3 as a linear combination of the corresponding vectors.

$$\begin{bmatrix} 0 & 2 \\ 2 & 2 \\ -1 & 0 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 6 \\ 16 \\ -5 \end{bmatrix}$$

If the above combination is linearly independent, the standard matrix of the left term is the basis.

```
In [1]: V12=np.hstack([v1,v2])
    ...: print(V12)
[[ 0  2]
 [ 2  2]
 [-1  0]]

In [2]: Matrix(np.hstack([v12, v3])).rref()
Out[2]:
(Matrix([
 [1, 0, 5],
 [0, 1, 3],
 [0, 0, 0]]),
 (0, 1))
```

The above results indicate that it is linearly independent. That is, among the three vectors v_1, v_2, v_3, the basis vector is v_1, v_2. That is, the span in the subspace (H) of the above three vectors can be expressed as follows.

$H = Span\{v_1 , v_2 \}$

In subspace $H=\{v_1 , v_2 , \ldots, v_p \}$
- If one of the vectors (v_k) is linearly coupled with the other vector, the remaining vectors excluding v_k become the span of H.
- $H \neq 0 \rightarrow$ Span of H contains the basis.

(Ex. 4.6) Determine which space, v_1 or v_2 , is the basis of R^2 or R^3 .

$v_1 =< 1, -2, 3 >$
$v_2 =< -2, 7, -9 >$

```
In [1]: v1=np.array([1,-2,3]).reshape(-1,1)
   ...: v2=np.array([-2, 7, -9]).reshape(-1,1)
   ...: V=np.hstack([v1,v2])
   ...: print(V)
[[ 1 -2]
 [-2  7]
 [ 3 -9]]

In [2]: Matrix(V).rref()
Out[2]:
(Matrix([
 [1, 0],
 [0, 1],
 [0, 0]]),
 (0, 1))
```

Since the pivot column of matrix V is column 1 and 2, v_1 and v_2 are both basis vectors. In other words, since it forms two bases, it is the base of a 2-dimensional (R^2) space.

(Ex. 4.7) Determine the basis of the following vectors.

$v_1 =< 1, -3, 4 >$
$v_2 =< 6, 2, -1 >$
$v_3 =< 2, -2, 3 >$
$v_4 =< -4, -8, 9 >$

```
In [1]: v1=np.array([1,-2,3]).reshape(-1,1)
   ...: v2=np.array([-2, 7, -9]).reshape(-1,1)
   ...: V=np.hstack([v1,v2])
   ...: print(V)
[[ 1 -2]
 [-2  7]
 [ 3 -9]]

In [2]: Matrix(V).rref()
Out[2]:
(Matrix([
 [1, 0],
 [0, 1],
 [0, 0]]),
 (0, 1))
```

In the rref form, columns 1 and 2 are pivot columns, so they represent 2 leading variables and 2 free variables. In other words, there are two basis vectors, v_1, v_2.

$$H = \text{Span}\{v_1, v_2\}$$

Let's look at each linear combination between these base vectors and the rest of the vectors, v_3 or v_4.

$$\begin{bmatrix} 1 & 6 \\ -3 & 2 \\ 4 & -1 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} 2 \\ -2 \\ 3 \end{bmatrix}$$

```
In [3]: Matrix(np.hstack([v1, v2, v3])).rref()
Out[3]:
(Matrix([
 [1, 0, 4/5],
 [0, 1, 1/5],
 [0, 0,   0]]),
 (0, 1))
```

$$\begin{bmatrix} 1 & 6 \\ -3 & 2 \\ 4 & -1 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} -4 \\ -8 \\ 9 \end{bmatrix}$$

```
In [4]: Matrix(np.hstack([v1, v2, v4])).rref()
Out[4]:
(Matrix([
 [1, 0,  2],
 [0, 1, -1],
 [0, 0,  0]]),
 (0, 1))
```

Both of the above linear combinations are unique . These results indicate that all combinations are linearly independent .

4.3 Null space and Column space

4.3.1 Null Space

The null space (Nul A) of matrix A in m × n dimension is the set of all solutions of the homogeneous linear equation (Ax = 0). Therefore, null space basically contains a zero vector.

Null space = solution set of the homogeneous equation system

Determining whether vector u is included in the null space of matrix A is to determine if the variable vector x is replaced by the vector u in the homogeneous equation of Ax = 0.

```
A=[< 1,-3,-2 >, < 5, 9, 1 >]
u=< 5, 3,-2 >
```

```
In [1]: A=np.array([[1,-3,-2],[-5, 9, 1]])
   ...: print(A)
[[ 1 -3 -2]
 [-5  9  1]]

In [2]: u=np.array([5,3,-2]).reshape(-1,1)
   ...: print(u)
[[ 5]
 [ 3]
 [-2]]

In [3]: np.dot(A, u)
Out[3]:
array([[0],
       [0]])
```

According to the result above, vector u is a solution satisfying the equations.

∴ u ⊂ Nul A

m×n dimension A as the standard matrix (coefficient matrix)
- Homogeneous linear combination
- linear independent

When the above conditions are satisfied,
the following is established
- Composed of m expressions with n variables
- # of column = Number of variables ⇔ Nul A ⊂ R^n

(Ex 4.8) Calculate the null space of standard matrix A.

$-3x_1 + 6x_2 - x_3 + x_4 - 7x_5 = 0$
$x_1 - 2x_2 + 2x_3 + 3x_4 - x_5 = 0$
$2x_1 - 4x_2 + 5x_3 + 8x_4 - 4x_5 = 0$

→ A=[< -3, 1, 2 >, < 6, -2, -4 >,< -1, 2, 5 >, < 1, 3, 8 >, < -7, 1, -4 >]
x=< x_1 , x_2 , x_3 , x_4 , x_5 >
c=< 0, 0, 0 >

→ Ax=c

The above matrix equation can be expressed as a general linear system as follows.

$$\begin{bmatrix} -3 & 6 & -1 & 1 & -7 \\ 1 & -2 & 2 & 3 & 1 \\ 2 & -4 & 5 & 8 & -4 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix}$$

```
In [1]: A=np.array([[-3, 6, -1, 1, -7],
   ...:         [1,-2,2,3,-1],
   ...:         [2, -4, 5, 8, -4]])
   ...: c=np.zeros([3,1])
   ...: rrefM=Matrix(np.hstack([A,c])).rref()
   ...: rrefM
Out[1]:
```

```
(Matrix([
 [1, -2.0, 0, -1.0,  3.0, 0],
 [0,    0, 1,  2.0, -2.0, 0],
 [0,    0, 0,    0,    0, 0]]),
 (0, 2))
```

The above results indicate that the variables x_1 and x_3 depend on the remaining variables. Therefore, the solution vectors are as follows.

$$\mathrm{NulA}=\begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \end{bmatrix} = \begin{bmatrix} 2x_2+x_4-3x_5 \\ x_2 \\ -2x_4+2x_5 \\ x_4 \\ x_5 \end{bmatrix} = \begin{bmatrix} 2 & 1 & -3 \\ 1 & 0 & 0 \\ 0 & -2 & 2 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x_2 \\ x_4 \\ x_5 \end{bmatrix}$$

Each of the column vectors in the null space is the solution of a linear combination to A . In other words, the set of solutions for the linear combination varies depending on the values of x_2 , x_4 , and, x_5 .

```
In [2]: A=np.array([[-3,6,-1,1,-7],
   ...:        [1,-2,2,3,-1],
   ...:        [2,-4,5,8,-4]])
   ...: s1=np.array([2,1,0,0,0]).reshape(-1,1)
   ...: s2=np.array([1,0,-2,1,0]).reshape(-1,1)
   ...: s3=np.array([-3, 0, 2, 0,1]).reshape(-1,1)
   ...: print(np.dot(A, s1))
[[0]
 [0]
```

```
 [0]]

In [3]: print(np.dot(A,s2))
[[0]
 [0]
 [0]]

In [4]: print(np.dot(A,s3))
[[0]
 [0]
 [0]]
```

x_1, x_2, x_3, x_4, and x_5 depend on each expression in the null space resulting above. That is, x_1 and x_3 are determined by any x_2, x_4, and x_5. The basis of the three variables x_2, x_4, and x_5 are calculated by the homogeneous linear combination as the following expression. The basis can be checked with pivot columns from the rref shown in the following code.

```
In [5]: x=np.array([[2,1,-3],
   ...:         [1,0,0],
   ...:             [0,-2,2],
   ...:         [0,1,0],
   ...:         [0,0,1]])
   ...: Matrix(x).rref()
Out[5]:
(Matrix([
 [1, 0, 0],
 [0, 1, 0],
 [0, 0, 1],
 [0, 0, 0],
 [0, 0, 0]]),
 (0, 1, 2))
```

According to the above results, all columns are pivot columns and there are no free variables. Therefore, they are all basis vectors. Span of null space can be expressed as follows.

```
Nul A=Span {< 2, 1, 0, 0, 0 >, < 1, 0, -2, 1, 0 >,< -3, 0, 2, 0, 1 >}
```

The null space can be checked using the function nullspace() from the sympy library. The result of this function returns the column vector(s). In the following code, columns are converted to matrix form.

```
In [6]: nul_A=Matrix(A).nullspace()
   ...: nul_A
Out[6]:
[Matrix([
 [2],
 [1],
 [0],
 [0],
 [0]]),
 Matrix([
 [ 1],
 [ 0],
 [-2],
 [ 1],
 [ 0]]),
 Matrix([
 [-3],
 [ 0],
 [ 2],
 [ 0],
 [ 1]])]

In [7]: re=nul_A[0]
   ...: for i in range(1, len(nul_A)):
   ...:     re=np.hstack([re, nul_A[i]])
   ...: re
Out[7]:
array([[2, 1, -3],
       [1, 0, 0],
       [0, -2, 2],
       [0, 1, 0],
       [0, 0, 1]], dtype=object)
```

Summarizing the results above:

The dimension of null space

- dim Nul = # of free variables in the homogeneous equation.

4.3.2 Column space

The set of linearly independent vectors in a linear combination with the m × n dimensional matrix A is called column space and is represented by Col A .

Column space is a set of linearly independent basis vectors.

From example (4.8), columns 1 and 3 in the rref of matrix A are pivot columns. In other words, the two column vectors become the basis vector, and a linear combination of these base vectors and the rest of the vectors is established.

```
In [8]: print(A)
[[-3  6 -1  1 -7]
 [ 1 -2  2  3 -1]
 [ 2 -4  5  8 -4]]

In [9]: Matrix(A).rref()
Out[9]:
(Matrix([
 [1, -2, 0, -1,  3],
 [0,  0, 1,  2, -2],
 [0,  0, 0,  0,  0]]),
 (0, 2))
```

The linear combination of the basis vector of the matrix A and the rest of the vectors is as follows.

```
In [10]: A_basis=A[:,[0, 2]]
    ...: print(A_basis)
[[-3 -1]
 [ 1  2]
 [ 2  5]]

In [11]: A1=np.hstack([A_basis, A[:,1].reshape(-1,1)])
    ...: print(A1)
[[-3 -1  6]
 [ 1  2 -2]
 [ 2  5 -4]]

In [12]: Matrix(A1).rref()
Out[12]:
(Matrix([
 [1, 0, -2],
```

```
 [0, 1,  0],
 [0, 0,  0]]),
 (0, 1))

In [13]: A3=np.hstack([A_basis, A[:,3].reshape(-1,1)])
    ...: print(A3)
[[-3 -1  1]
 [ 1  2  3]
 [ 2  5  8]]

In [14]: Matrix(A3).rref()
Out[14]:
(Matrix([
 [1, 0, -1],
 [0, 1,  2],
 [0, 0,  0]]),
 (0, 1))

In [15]: A4=np.hstack([A_basis, A[:,4].reshape(-1,1)])
    ...: print(A4)
[[-3 -1 -7]
 [ 1  2 -1]
 [ 2  5 -4]]

In [16]: Matrix(A4).rref()
Out[16]:
(Matrix([
 [1, 0,  3],
 [0, 1, -2],
 [0, 0,  0]]),
 (0, 1))
```

All linear combinations are linearly independent. In other words, A_1, A_3, and A_4 vectors are all A_0, and A_2 vectors as Span. This will be the subspace of the vector space created by A_0, and A_2. As a result, the base vectors in matrix A are A_0, and A_2, which are the columnspaces.

The column space of the matrix can be checked by the sympy columnspace() function.

```
In [17]: col_A=Matrix(A).columnspace()

In [18]: col_A
Out[18]:
[Matrix([
```

```
[-3],
[ 1],
[ 2]]),
Matrix([
[-1],
[ 2],
[ 5]])]
```

```
Col A = {< -3, 1, 2 >, < -1, 2, 5 >}
```

The above result can be defined as:

> The column space consists of column vectors corresponding to the pivot column in the rref of the standard matrix.

(Ex. 4.9) Calculate the column space of the vector set W. W can be represented as matrix A.

```
W={< 6a-b, a+b, -7a >}
→ A=[< 6, 1, -7 >, < -1, 1, 0 >]
```

If all column vectors of matrix A are column space, then the homogeneous linear combination is linearly independent. That is, all column vectors are base vectors.

$$\begin{bmatrix} 6 & -1 \\ 1 & 1 \\ -7 & 0 \end{bmatrix}\begin{bmatrix} a \\ b \end{bmatrix}=\begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix}$$

```
In [1]: A=np.array([[6,-1],[1,1],[-7,0]])
   ...: c=np.zeros([3,1])
   ...: Matrix(np.hstack([A,c])).rref()
Out[1]:
(Matrix([
 [1, 0, 0],
 [0, 1, 0],
 [0, 0, 0]]),
 (0, 1))
```

Linear combination is linear independence, so that all the column vectors of A are the basis vector and become the column space Col A . Col A can be expressed as Span of two basis vectors as follows.

$$W=\text{Col } A=\text{Span } \{< 6, 1, -7 >, < -1, 1, 0 >\}$$

```
In [2]: col_A=Matrix(A).columnspace()

In [3]: re=col_A[0]
   ...: for i in range(1, len(col_A)):
   ...:     re=np.hstack([re, col_A[i]])

In [4]: print(re)
[[6 -1]
 [1 1]
 [-7 0]]
```

When transposing a matrix, the column vector of the original matrix becomes the row vector. In this case, the column space before transposition becomes the row space after transposition. The row space of matrix A is represented by row A .

(Ex. 4.10) Let's determine the column space of matrix M.

$$M=\begin{bmatrix} -2 & -5 & 8 & 0 & -17 \\ 1 & 3 & -5 & 1 & 5 \\ 3 & 11 & -19 & 7 & 1 \\ 1 & 7 & -13 & 5 & -3 \end{bmatrix}$$

```
In [1]: M=np.array([[-2,-5, 8, 0, -17],
   ...:         [1, 3, -5, 1, 5],
   ...:         [3, 11, -19, 7, 1],
   ...:         [1, 7, -13, 5, -3]])
   ...: print(M)
[[ -2  -5   8   0 -17]
 [  1   3  -5   1   5]
 [  3  11 -19   7   1]
 [  1   7 -13   5  -3]]

In [2]: c=np.zeros([4, 1])
```

```
In [3]: au=np.hstack([M, c])

In [4]: M_rref=Matrix(au).rref()
   ...: M_rref
Out[4]:
(Matrix([
 [1, 0,  1.0, 0,  1.0, 0],
 [0, 1, -2.0, 0,  3.0, 0],
 [0, 0,    0, 1, -5.0, 0],
 [0, 0,    0, 0,    0, 0]]),
 (0, 1, 3))
```

In the above result, since 1, 2, and 4 column vectors are pivot columns, the vectors corresponding to them become column spaces.

```
In [5]: col_M=Matrix(M).columnspace()

In [6]: re=col_M[0]
   ...: for i in range(1, len(col_M)):
   ...:     re=np.hstack([re, col_M[i]])

In [7]: print(re)
[[-2 -5 0]
 [1 3 1]
 [3 11 7]
 [1 7 5]]
```

The variables x_1, x_2, and x_4 of the linear combination of matrix M depend on x_3 and x_5 and can be expressed as

$$\begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \end{bmatrix} = \begin{bmatrix} -x_3 - x_5 \\ 2x_3 - 3x_5 \\ x_3 \\ 5x_4 \\ x_5 \end{bmatrix}$$

$$= x_3 \begin{bmatrix} -1 \\ 2 \\ 1 \\ 0 \\ 0 \end{bmatrix} + x_5 \begin{bmatrix} -1 \\ -3 \\ 0 \\ 5 \\ 1 \end{bmatrix}$$

The two vectors in the last right equation above are null space.

```
In [8]: nul_M=Matrix(M).nullspace()

In [9]: re=nul_M[0]

In [10]: for i in range(1, len(nul_M)):
    ...:     re=np.hstack([re, nul_M[i]])

In [11]: print(re)
[[-1 -1]
 [2 -3]
 [1 0]
 [0 5]
 [0 1]]
```

5 Coordinate system

5.1 Vector Coordinate System

If a linear combination has a trivial solution, all the column vectors in the system are the basis and the elements of the span . Linear combinations can be expressed as Eq. 5.1 using these basis vectors.

$$x = c_1 b_1 + c_2 b_2 + \dots + c_p b_p = Bc \quad \text{(Eq. 5.1)}$$

The left term x in Eq. 5.1 becomes the subspace of the vector space by the base vectors of the right term. The vector space of the bases is as set B below. That is, subspace H becomes the span of set B and can be expressed as Eq. 5.2.

$$B = \{b_1 , b_2 , \dots, b_p \}$$
$$H = \text{Span } \{b_1 , b_2 , \dots, b_p \} \quad \text{(Eq. 5.2)}$$

(Ex. 5.1) Are the vectors v_1 , v_2 the basis vector of the vector c?

$$v_1 = < 3, 6, 2 >$$
$$v_2 = < -1, 0, 1 >$$
$$c = < 3, 12, 7 >$$

```
In [1]: v1=np.array([3, 6, 2]).reshape(-1,1)
   ...: v2=np.array([-1,0,1]).reshape(-1,1)
   ...: c=np.array([3, 12, 7]).reshape(-1,1)

In [2]: au=np.hstack([v1,v2, c])

# Augment matrix
In [3]: print(au)
[[ 3 -1  3]
 [ 6  0 12]
```

```
 [ 2  1  7]]

In [4]: Matrix(au).rref()
Out[4]:
(Matrix([
 [1, 0, 2],
 [0, 1, 3],
 [0, 0, 0]]),
 (0, 1))
```

As shown in the above result, the linear combination of the three vectors is linearly independent, so v_1 and v_2 are basis vectors.

If V is a base matrix of base vectors at Vx = c, vector c is the result of considering the direction and magnitude of the base in vector x . This means that vector x is a modified coordinate system based on the basis vector of V . In other words, the meaning can be summarized as follows.

From Vx =cx,

- x vector: Coordinate by general coordinate system
- c vector: Coordinate of vector x as the coordinate system in which the base vectors are referenced

The most important reason for specifying the basis in vector space V is to determine the coordinate system that is based on that space. That is, the vector represented by the general coordinate system as shown in Fig. 5.1 can be modified with the coordinate system based on the bases.

Through a linear combination like Eq. 5.3, vector c is corrected to r in a modified coordinate system based on the base.

Basis · c = r (Eq. 5.3)
The coordinates of r in the coordinate system modified by the basis
= the coordinates of c in the general coordinate system

In normal coordinates, a vector c is the reference coordinate to represent a vector r in the modified coordinate system. This reference coordinate is called a coordinate vector . In other words, the trivial solution of a linear combination is called a coordinate vector .

(Ex. 5.2) Determine the r vector as a result of the linear combination of two basis vectors b_1 , b_2 and vector c.

$b_1 = < 1, 0 >$
$b_2 = < 1, 2 >$
$c = < -2, 3 >$

```
In [1]: b1=np.array([[1],[0]])
   ...: b2=np.array([[1],[2]])
   ...: c=np.array([[-2],[3]])
   ...: b=np.hstack([b1, b2])

In [2]: print(b)
[[1 1]
 [0 2]]

In [3]: print(c)
[[-2]
 [ 3]]

In [4]: r=np.dot(b, c)
   ...: print(r)
[[1]
 [6]]
```

The above results can be seen in Fig. 5.1.

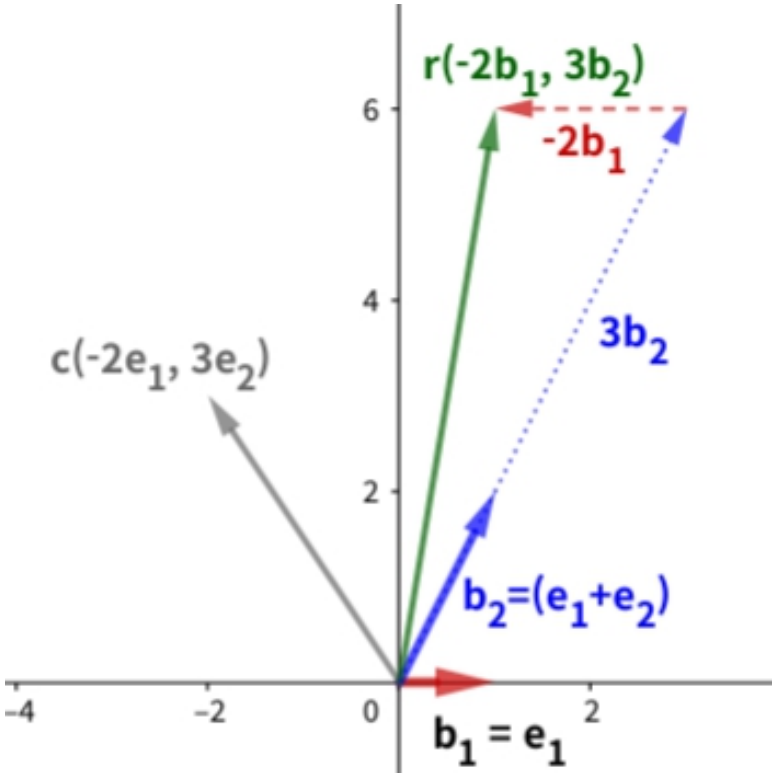


Fig. 5.1 Correction of coordinates by basis vectors.

As shown in Fig. 5.1, the result of Ex. 5.2, c can be represented by r in a coordinate system with base vectors b_1 and b_2 as reference coordinates.

The basic axes x (e_1 = <1,0>) and y (e_2 = <0,1>) are modified as the basis vectors b_1 and b_2 .

b_1 direction, scale unit 1
b_2 direction, scale unit 2

So c becomes r at the modified coordinates relative to b_1 and b_2 .

$c = -2e_1 + 3e_2 \rightarrow r = -2b_1 + 3b_2$
c and r are the results of changes in the reference coordinate system.

The above process can be organized as follows.

If P is a matrix composed of basis of R^n , the linear combination can be expressed as Eq. 5.4.

$x = c_1 b_1 + c_2 b_2 + \ldots + c_n b_n \rightarrow x = P \cdot c$ (Eq. 5.4)
$P = [b_1, b_2, \ldots, b_n]$

The base matrix is an invertible matrix that can be:

- $x = P \cdot c \rightarrow P^{-1} \cdot x = c$

(Ex. 5.3) Determine the basis vectors of the following polynomial.

$1+2t^2 = 0$
$4+t+5t^2 = 0$
$3+2t = 0$
$\rightarrow P = [< 1, 4, 3 >, < 0, 1, 2 >, <2, 5, 0 >]$
$T = < 1, t, t^2 >$
$c = < 0, 0, 0 >$
$\rightarrow P \cdot T = c$

Three polynomials can be expressed as linear combinations as follows:

$$\begin{bmatrix} 1 & 0 & 2 \\ 4 & 1 & 5 \\ 3 & 2 & 0 \end{bmatrix} \begin{bmatrix} 1 \\ t \\ t^2 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix}$$

```
In [1]: P=np.array([[1,0,2],[4,1,5],[3,2,0]])
   ...: c=np.array([0,0,0]).reshape(3,1)
   ...: au=np.hstack([P,c])
   ...: Matrix(au).rref()
Out[1]:
(Matrix([
 [1, 0,  2, 0],
 [0, 1, -3, 0],
 [0, 0,  0, 0]]),
 (0, 1))
```

According to the above results, free variables exist. In other words, it has a variety of solutions depending on the value of t^2 , so it is linearly dependent with a non-trivial solution. Therefore, the matrix P is not a base matrix.

As shown in the following result, the pivot column of P is columns 1 and 2. Using these as the basis vector, the linear coupling with the rest of the

column vectors is independent.

```
In [2]: Matrix(P).rref()
Out[2]:
(Matrix([
 [1, 0,  2],
 [0, 1, -3],
 [0, 0,  0]]),
 (0, 1))
```

Therefore, the basis space of the polynomial above can be expressed as

$$P = \text{Span}\ \{< 1, 4, 3 >, < 0, 1, 2 >\}$$

5.2 Dimension and Rank

The dimension of subspace H is the number of basis vectors that make up the space. The number of basis vectors is called the rank .

(Ex. 5.4) Determine the rank of the homogeneous equation.

$3x_1 + 6x_2 - x_3 + x_4 + 7x_5 = 0$
$x_1 - 2x_2 + 2x_3 + 3x_4 - x_5 = 0$
$2x_1 - x_2 + 5x_3 + 8x_4 - 4x_5 = 0$
→ A = [< -3, 1, 2 >,
< 6, -2, -4 >,
<-1, 2, 5 >,
< 1, 3, 8 >,
< 7, -1, -4 >]
c = < 0, 0, 0 >

```
In [1]: A=np.array([[-3,6,-1,1,7],
   ...:       [1,-2,2,3,-1],
   ...:       [2, -4, 5, 8, -4]])
   ...: c=np.array([0,0,0]).reshape(-1,1)
   ...: au=np.hstack([A,c])
   ...: print(au)
[[-3  6 -1  1  7  0]
 [ 1 -2  2  3 -1  0]
 [ 2 -4  5  8 -4  0]]

In [2]: Matrix(au).rref()
Out[2]:
(Matrix([
 [1, -2, 0, -1, 0, 0],
 [0,  0, 1,  2, 0, 0],
 [0,  0, 0,  0, 1, 0]]),
 (0, 2, 4))
```

According to the above results, the pivot columns are columns 1, 3, and 5, so the column vectors corresponding to these columns are the basis vectors of Ex. 5.3. Therefore, linear independence is established between each of the remaining column vectors and basis vectors.

```
In [3]: P=A[:, [0, 2, 4]]
   ...: print(P)
[[-3 -1  7]
 [ 1  2 -1]
 [ 2  5 -4]]

In [4]: Matrix(np.hstack([P, A[:,1].reshape(-1,1)])).rref()
Out[4]:
(Matrix([
 [1, 0, 0, -2],
 [0, 1, 0,  0],
 [0, 0, 1,  0]]),
 (0, 1, 2))

In [5]: Matrix(np.hstack([P, A[:,3].reshape(-1,1)])).rref()
Out[5]:
(Matrix([
 [1, 0, 0, -1],
 [0, 1, 0,  2],
 [0, 0, 1,  0]]),
 (0, 1, 2))
```

As a result, it can be expressed as:

```
A = Span {< -3, 1, 2 >, < -1, 2, 5 >, < 7. -1, -4 >}
Rank A = 3
```

The above results are summarized as follows.

In Matrix A,
Rank A = # of pivot column = # of basis = Dimension of subspace

As a result, in the linear system, all of the following are equivalent.

linear independent
≡ basis
≡ a unique solution (≈ trivial solution)
≡ The existence of a unique coordinate vector
≡ No free variable
≡ Rank A = m in the nxm dimension matrix A

$H = \text{Span}\{v_1 , v_2 \}$ means that v_1 and v_2 are basis vectors. In this case, the dimension of H is two-dimensional. In addition, since the linear combination with the base vectors has a trivial solution, the coordinate vector can be determined.

```
In [6]: v1=np.array([[1],[4],[-3]])
   ...: v2=np.array([[-2],[5],[-7]])
   ...: c=np.array([0,0,0]).reshape(-1,1)
   ...: Matrix(np.hstack([v1, v2, c])).rref()
Out[9]:
(Matrix([
 [1, 0, 0],
 [0, 1, 0],
 [0, 0, 0]]),
 (0, 1))
```

The solution of independent linear combinations is the coordinate vector. Therefore, the x vector of the following equation becomes the coordinate vector .

$$[v_1 , v_2]\cdot x = c$$

```
In [7]: r=np.array([[2],[9],[-7]])
   ...: print(r)
[[ 2]
 [ 9]
 [-7]]

In [8]: Matrix(np.hstack([v1,v2,r])).rref()
Out[8]:
(Matrix([
 [1, 0, 28/13],
 [0, 1,  1/13],
 [0, 0,     0]]),
 (0, 1))
```

According to the above result, the coordinate vector is $< {}^{28}/_{13} , {}^{1}/_{13} >$.

The rank is 2 because it is equal to the number of pivot columns. This can be checked with the np.matrix_rank() function.

```
In [9]: la.matrix_rank(np.hstack([v1, v2]))
Out[9]: 2
```

The above result shows that the number of pivot columns equals the dimension of the column space (dim Col), and the number of non-pivot columns equals the number of null spaces in the matrix. That is, the subspace H composed of v_1 and v_2 has 2 column spaces and 0 null spaces.

```
In [10]: Matrix(np.hstack([v1, v2])).columnspace()
Out[10]:
[Matrix([
 [ 1],
 [ 4],
 [-3]]),
 Matrix([
 [-2],
 [ 5],
 [-7]])]

In [11]: Matrix(np.hstack([v1, v2])).nullspace()
Out[11]: []
```

Therefore, the following holds.

Rank A of matrix A = dim Col A (dimension of column space of A)

In matrix A of m × n dimension

rank A + dim Nul A
= (# of pivot column) + (# of nonpivot column)
= n

(Ex. 5.5) Determine the rank of the matrix A.

```
In [1]: A=np.array([[3,0,-1],
   ...:             [3,0,-1],
   ...:             [4,0,5]])
   ...: print(A)
[[ 3  0 -1]
 [ 3  0 -1]
```

```
 [ 4  0  5]]

In [2]: Matrix(A).rref()
Out[2]:
(Matrix([
 [1, 0, 0],
 [0, 0, 1],
 [0, 0, 0]]),
 (0, 2))

In [3]: la.matrix_rank(A)
Out[3]: 2
```

From the rref of A , there are 2 pivot columns and 1 free variable. therefore

```
Rank A=2
Total number of columns
 = dim Col A+ dim Nul A
 = 2+1
 = 3
```

5.3 Eigenvector and Eigenvalue

In linear algebra, eigenvectors are vectors that are not zero vectors whose orientation does not change even after a linear transformation of the square matrix occurs. In other words, it means the basis vectors. The unique scalar value that can change the length of this eigenvector is called the eigenvalue , which is a specific value corresponding to each eigenvector.

(Ex. 5.6) Examine the relationship between matrix A and two vectors u and v.

```
A = [< -3, 1 >, < -2, 0 >]
u = < -1, 0 >
v = < 2, 1 >
```

```
In [1]: A=np.array([[3,-2],[1,0]])
   ...: u=np.array([[-1],[1]])
   ...: v=np.array([[2],[1]])

In [2]: Matrix(np.hstack([A, u])).rref()
Out[2]:
(Matrix([
 [1, 0, 1],
 [0, 1, 2]]),
 (0, 1))

In [3]: Matrix(np.hstack([A, v])).rref()
Out[3]:
(Matrix([
 [1, 0,   1],
 [0, 1, 1/2]]),
 (0, 1))
```

According to the above results, both linear combinations are linearly independent because the rank is 2 in each linear combination of matrix A and vectors u and v . In other words, all column vectors of matrix A are the basis. This matrix is called the base matrix.

The matrix product of u and v for matrix A is:

```
In [4]: Au=np.dot(A, u)
   ...: print(Au)
[[-5]
 [-1]]

In [5]: Av=np.dot(A, v)
   ...: print(Av)
[[4]
 [2]]
```

The results are shown in Fig. 5.2.

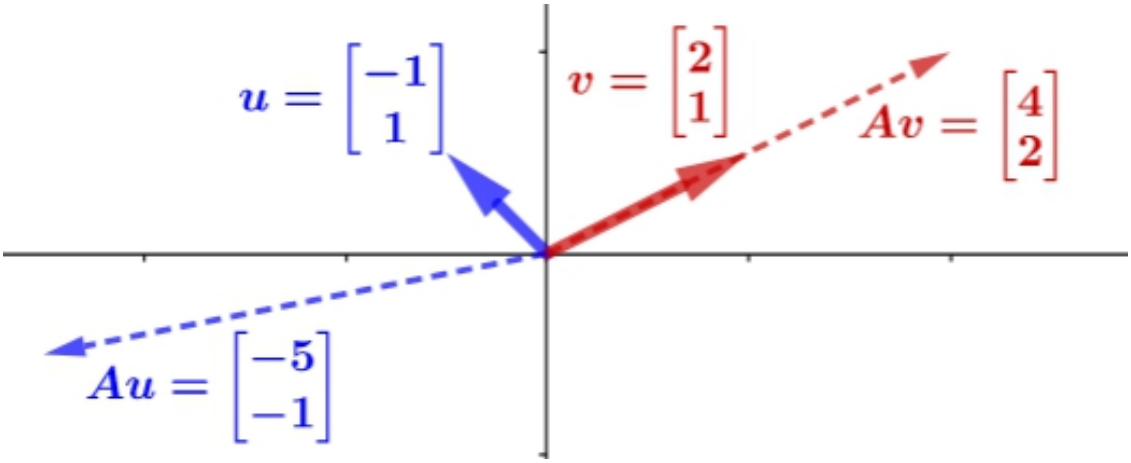


Fig. 5.2 Standard matrix and linear combination of two vectors (Ex. 5.5).

From Fig. 5.2, unlike Au , Av is the result of doubling v .

```
AV = 2v
```

The direction of the original vector v is the same as that of the resulting vector by linear combination (Av). Therefore, v and scalar 2 are called eigenvectors and eigenvalues of A , respectively.

Summarizing the above process,

For m × n-dimensional matrix A , there are scalar (λ) and non-zero vector (x) satisfying the left-hand side of Eq. 5.5, which are arranged as the right-

hand side of Equation 5.5. The right hand side of Eq. 5.5 is called the characteristic equation .

$$A \cdot x = \lambda x \rightarrow (A-\lambda I) \cdot x = 0 \quad (\text{Eq. 5.5})$$

(Ex. 5.7) Calculate the eigenvalues of matrix A.

$$A = [<3, 3>, <4, -1>]$$

If the eigenvalue of matrix A is a and the eigenvector is v, then Eq. 5.5 must be satisfied. This relationship can be rewritten as:

$$Av = av \rightarrow Av-av = 0 \rightarrow (A-aI)v = 0$$

Determinant represents the area of the shape produced by the matrix. In this sense, since Av and av are the same, the determinant of the characteristic equation derived from this equation is 0. In other words, A-aI must be 0 because v is not 0 by the definition of the eigenvector in the characteristic equation. As a result, the inverse of this expression cannot exist.

$$\det(A - aI) = 0$$

```
In [1]: a=symbols('a')
   ...: A=Matrix([[3,4],[3,-1]])
   ...: print(A)
Matrix([[3, 4], [3, -1]])

In [2]: I2=eye(2)
   ...: print(I2)
Matrix([[1, 0], [0, 1]])

#characteristic Eq.(=char_eq)
In [3]: char_eq=A-a*I2
   ...: print(char_eq)
Matrix([[3 - a, 4], [3, -a - 1]])

#determinant of char_eq.
In [4]: char_eq_det=factor(det(char_eq))
   ...: print(char_eq_det)
(a - 5)*(a + 3)
```

```
#determinant=0, solution of the char_eq.
In [5]: solve(char_eq_det)
Out[5]: [-3, 5]
```

In the code above, the eigenvalue a was made unknown and the value was calculated. In this process, the eye() of the sympy module was used to generate the identity matrix. In addition, the solve() function of the same module was used to calculate the unknowns of the equations created by the determinant of the characteristic equation.

As a result, the eigenvalues are -3 and 5. Based on this value, the eigenvector becomes the solution vector of the following linear combination.

$$\begin{bmatrix} 3 & 4 \\ 3 & -1 \end{bmatrix}\begin{bmatrix} v_{11} \\ v_{21} \end{bmatrix} = -3 \cdot \begin{bmatrix} v_{11} \\ v_{21} \end{bmatrix} \rightarrow \begin{bmatrix} 6 & 4 \\ 3 & 2 \end{bmatrix}\begin{bmatrix} v_{11} \\ v_{21} \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$$

$$\begin{bmatrix} 3 & 4 \\ 3 & -1 \end{bmatrix}\begin{bmatrix} v_{12} \\ v_{22} \end{bmatrix} = 5 \cdot \begin{bmatrix} v_{12} \\ v_{22} \end{bmatrix} \rightarrow \begin{bmatrix} 2 & 4 \\ 3 & -6 \end{bmatrix}\begin{bmatrix} v_{12} \\ v_{22} \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$$

```
# rref of characteristic equation
In [6]: (A+3*eye(2)).rref()
Out[6]:
(Matrix([
 [1, 2/3],
 [0,   0]]),
 (0,))

In [7]: (A+3*eye(2)).nullspace()
Out[7]:
[Matrix([
 [-2/3],
 [   1]])]

In [8]: (A-5*eye(2)).nullspace()
Out[8]:
[Matrix([
 [2],
 [1]])]
```

The rref of the characteristic expression, which is an irreversible matrix, contains free variables. In other words, a non trivial solution exists. In this case, the nullspace() function was applied to calculate the basis of various solutions. As a result, the eigenvalues and eigen-matrix of matrix A are as follows.

```
Eigenvalue={-3, 5}
Eigenvector={<-2/3, 1>, <2, 1>}
```

Since the eigenvector corresponding to each eigenvalue is the basis vector of various values, it can be represented variously according to an arbitrary scalar product.

The above process can be summarized as follows.

- A linear combination between n × n-dimensional matrix A , eigenvalue (λ), and eigenvector (x) is established.
- Characteristic formulas A - λI are irreversible matrices and determinants are zero.
- The eigenvector corresponding to the eigenvalue is the null space of the characteristic expression.
- The null spaces are subspaces of R^n and are called eigen-spaces as eigenvectors corresponding to λ.

The eigenmatrix composed of eigenvectors of matrix A can be calculated using the eig() function of the numpy.linalg module. This function returns the eigenvalues and eigenmatrix.

The matrix A created in the code above is a sympy object. Converting this to numpy's array object is done by the np.array(object, dtype = np.float) function.

```
In [9]: la.eig(np.array(A, dtype=np.float))
Out[9]:
(array([ 5., -3.]),
 array([[ 0.89442719, -0.5547002 ],
        [ 0.4472136 ,  0.83205029]]))
```

The eigenvectors by eig() differ from the values calculated above. However, the difference between the two results has a scalar-fold

relationship to each other and shows the same direction. As a result, it can be regarded as the same result.

(Ex. 5.8) If the eigenvalue of the next matrix is 7, let's determine the eigenvector.

$$A=\begin{bmatrix}1&6\\5&2\end{bmatrix} \rightarrow \begin{bmatrix}1&6\\5&2\end{bmatrix}\cdot\begin{bmatrix}x_1\\x_2\end{bmatrix}=7\begin{bmatrix}x_1\\x_2\end{bmatrix}$$

$$\rightarrow \left(\begin{bmatrix}1&6\\5&2\end{bmatrix}-7\begin{bmatrix}1&0\\0&1\end{bmatrix}\right)\cdot\begin{bmatrix}x_1\\x_2\end{bmatrix}=\begin{bmatrix}0\\0\end{bmatrix}$$

$$\rightarrow \begin{bmatrix}-6&6\\5&-5\end{bmatrix}\cdot\begin{bmatrix}x_1\\x_2\end{bmatrix}=\begin{bmatrix}0\\0\end{bmatrix}$$

The problem is to determine whether a linear combination is established for the characteristic expressions $A-7\cdot I$, which is the result of the calculation between the matrix A and the eigenvalue 7.

```
In [1]: A=np.array([[1,6],[5,2]])
   ...: i7=7*np.eye(2)
   ...: c=np.zeros([2,1])
   ...: A_i7=A-i7

In [2]: print(A_i7)
[[-6.  6.]
 [ 5. -5.]]

In [3]: Matrix(np.hstack([A_i7, c])).rref()
Out[3]:
(Matrix([
 [1, -1.0, 0],
 [0,    0, 0]]),
 (0,))

In [4]: Matrix(A_i7).nullspace()
Out[4]:
[Matrix([
 [1.0],
 [  1]])]
```

According to the result above, there is one pivot column and one free variable, that is, various solutions exist by free variable x_2 . As a result, 7 is an eigenvalue because there is a linear combination between matrix A and the eigenvalues, eigenvectors. The eigenvector of the base form corresponding to this eigenvalue is <1,1>.

```
eigenvalue= < 1, 1 >
```

The eigenvector is calculated using the eig() function.

```
In [5]: Aeig=la.eig(A)
   ...: Aeig
Out[5]:
(array([-4.,  7.]),
 array([[-0.76822128, -0.70710678],
        [ 0.6401844 , -0.70710678]]))
```

The above result is:

Eigenvalue	Eigenvector
-4	< 0.7682, 0.6402>
7	< 0.7071, -0.7071 >

The following equation must be established with the resulting eigenvector and eigenvalue.

$$Ax = \lambda x$$

```
In [6]: np.dot(A, Aeig[1][:,1])\
   ...:     ==Aeig[0][1]*Aeig[1][:,1]
Out[6]: array([ True,  True])
```

(Ex.5.9) Determine the eigenspace at eigenvalue 2 for the matrix A.

```
A = [< 4, 2, 2 >, < -1, 1, -1 >, < 6, 6, 8 >]
c = < 0, 0, 0 >
```

```
In [1]: A=np.array([[4,-1,6],
   ...:             [2,1,6],
   ...:             [2,-1,8]])
   ...: c=np.zeros([3,1])
   ...: print(A)
[[ 4 -1  6]
 [ 2  1  6]
 [ 2 -1  8]]

In [2]: Matrix(np.hstack([A, c])).rref()
Out[2]:
(Matrix([
 [1, 0, 0, 0],
 [0, 1, 0, 0],
 [0, 0, 1, 0]]),
 (0, 1, 2))

In [3]: Aeig=la.eig(A)

# EigenValue
In [4]: Aeig[0]
Out[4]: array([9., 2., 2.])

# EigenVector
In [5]: np.around(Aeig[1],3)
Out[5]:
array([[-0.577, -0.612,  0.321],
       [-0.577, -0.787, -0.911],
       [-0.577,  0.073, -0.259]])
```

Matrix A is the base matrix, and there are two eigenvectors corresponding to eigenvalue 2.

```
In [6]: x=Aeig[1][:,[1,2]]
   ...: print(np.around(x, 3))
[[-0.612  0.321]
```

```
 [-0.787 -0.911]
 [ 0.073 -0.259]]
```

The null space from the characteristic equation for the eigenvalue 2 is calculated as follows.

```
In [7]: A_I2=A-2*np.eye(3)
   ...: print(A_I2)
[[ 2. -1.  6.]
 [ 2. -1.  6.]
 [ 2. -1.  6.]]

In [8]: Matrix(A_I2).nullspace()
Out[8]:
[Matrix([
 [0.5],
 [  1],
 [  0]]),
 Matrix([
 [-3.0],
 [   0],
 [   1]])]
```

The null space of the characteristic expression becomes the eigenvector, and the eigenvector is the eigenspace as the basis vector.

The eigenvectors that make up the eigenspace are all base vectors. Therefore, linear combination by the eigenvectors(v_1 , v_2 , …, v_r) is linear independence.

(Ex. 5.10) Determine the eigenvalues and eigenvectors of the next matrix.

```
A=[<3, 0, 0>, <6, 0, 0>, <-8, 6, 2>]
```

```
In [1]: A=np.array([[3,6,-8],[0,0,6],[0,0,2]])
   ...: print(A)
[[ 3  6 -8]
 [ 0  0  6]
 [ 0  0  2]]

In [2]: Matrix(A).rref()
```

```
Out[2]:
(Matrix([
 [1, 2, 0],
 [0, 0, 1],
 [0, 0, 0]]),
 (0, 2))

In [3]: evalue, evec=la.eig(A)

In [4]: print(evalue)
[3. 0. 2.]

In [5]: print(np.around(evec, 3))
[[ 1.    -0.894 -0.953]
 [ 0.     0.447  0.286]
 [ 0.     0.     0.095]]
```

The rref of matrix A has two pivot columns and one free variable. The basis vector of the matrix is the column vector corresponding to the pivot column and becomes the column space of the matrix.

```
In [6]: Matrix(A).columnspace()
Out[6]:
[Matrix([
 [3],
 [0],
 [0]]),
 Matrix([
 [-8],
 [ 6],
 [ 2]])]
```

The above results show that there are 3 eigenvectors for 3 eigenvalues, but according to the column space, there are 2 basis vectors. The difference occurs because the eigenvalue 0 cannot have an eigenvector.

$$A \cdot x = 0x \rightarrow A \cdot x = 0$$

The linear combination of A as in the above equation may be either linear independence or linear dependence. If A is linear dependent, it violates the definition that eigenvalues and eigenvectors are computed from a reversible

matrix. Also, if it is linear independent, there is no characteristic equation related to the matrix. Therefore, eigenvectors also cannot exist.

```
Dimension of column space
  = dimension of rank
  = # of non zero eigenvalues
```

6 Transform

6.1 Kernel and Range

Transform refers to a function that assigns a number x to an expression and returns y, the result corresponding to that value. The range and number of values entered into the function cannot always determine the range or number of results. That is, depending on the function, one real input value may be output as a value containing multiple rational numbers.

The input range to the function is called the domain , and the range of all possible outputs is called the codomain . As an example, python uses the int(number) function to make all numbers into integers. In this case, the domain is a real number, but the codomain will be an integer.

```
In [1]: x=3.24
   ...: y=int(x)
   ...: y
Out[1]: 3
```

The result of a function in the codomain is called the image , and the set of these images is called the range . Range is a subset of codomain.

```
Range ⊆ Codomain
```

Fig. 6.1 shows the domain, codomain, and range.

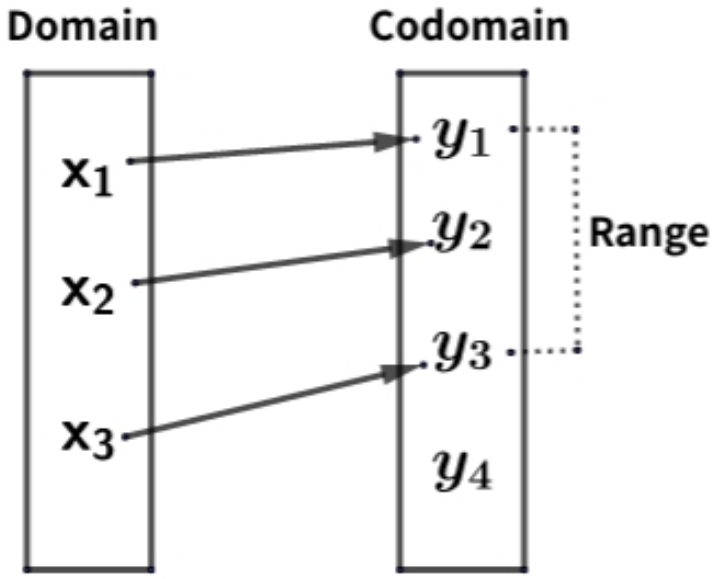


Fig. 6.1 Domain, Codomain, and Range.

The domain, codomain, and range of functions shown in Fig. 6.1 can be defined in three relationships as shown in Fig. 6.2.

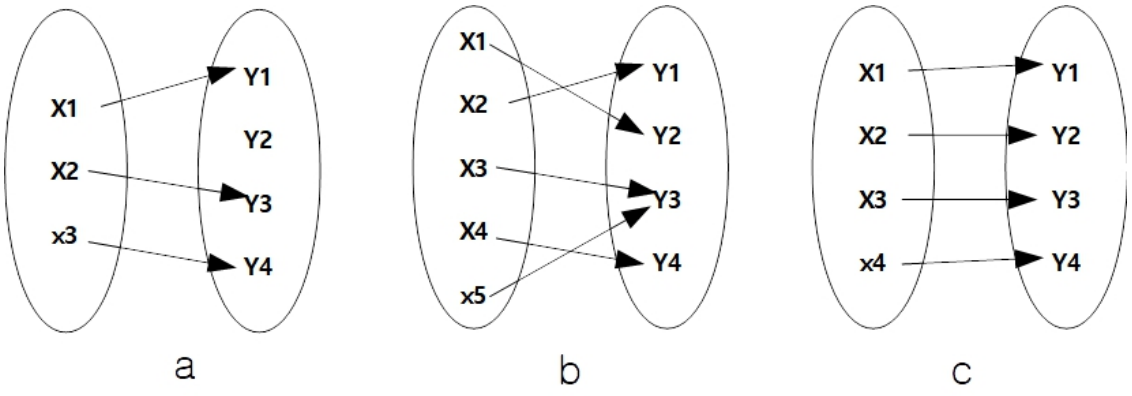


Fig. 6.2. Relationship between domain, codomain, and range.

• Injective (one-to-one): One-to-one correspondence between domain and codomain. Fig. 6.2 (a) and (c)
• Subjective (onto): range=codomain. Fig. 6.2 (b) and (c)

• Bijective : In case of one-to-one correspondence and co-domain and range are the same. Fig. 6.2 (c)

The linear bond can be expressed as Eq. 3.1. For example, the linear combination of two vectors can be represented as

$a_1 x_1 + a_2 x_2 = c$

a_1, a_2 : scalar

x_1, x_2, c: vector

The transformation of this linear combination is called a linear transformation. That is, the linear transformation is expressed by Eq. 6.1

$$f(a_1 x_1) + f(a_2 x_2) = f(a_1 x_1 + a_2 x_2) \quad \text{(Eq. 6.1)}$$

The domain in which all the ranges of these linear transformations are zero is called the kernel . If the linear transformation like Eq. 6.1 is defined as T, the above relationship is expressed as ker T as Eq. 6.2.

When the range is all 0, the domain is called the kernel.

$$\ker T = \{v \in Rn | T(v)=0\} \quad \text{(Eq. 6.2)}$$

(Ex. 6.1) The following is a linear transformation that converts from 2 dimension to 2 dimension. Let's determine the condition for ker T in this linear transformation.

$$T: R^2 \rightarrow R^2 , T(x, y) = T(x-y, 0)$$

The standard matrix that can cause the above transformation is:

$$T\left(\begin{bmatrix} x \\ y \end{bmatrix}\right) = x\begin{bmatrix} 1 \\ 0 \end{bmatrix} + y\begin{bmatrix} -1 \\ 0 \end{bmatrix} = \begin{bmatrix} 1 & -1 \\ 0 & 0 \end{bmatrix}\begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} x-y \\ 0 \end{bmatrix}$$

In the above equation, for ker T, all ranges must be 0, so the condition x = y must be satisfied. Therefore, it is as follows.

$$\ker T = \{(x, y) \in R^2 \mid y=x\}$$

A transformation that is bijective and takes place in the same dimension is called isomorphism . In other words, in isomorphism, the dimension of the domain and the codomain are the same.

(Ex. 6.2) For the conversion of T (x, y) = (y, x), ker T = 0 is injective.

$$T\left(\begin{bmatrix} x \\ y \end{bmatrix}\right)=x\begin{bmatrix} 0 \\ 1 \end{bmatrix}+y\begin{bmatrix} 1 \\ 0 \end{bmatrix}=\begin{bmatrix} y \\ y \end{bmatrix}\rightarrow\begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}\begin{bmatrix} x \\ y \end{bmatrix}=\begin{bmatrix} 0 \\ 0 \end{bmatrix}$$

The above equation is a linear combination of homogeneous equations. If the above expression has a trivial solution, it means one-to-one reaction and is injective .

```
In [1]: A=np.array([[0, 1], [1,0]])
   ...: la.det(A)
Out[1]: -1.0

In [2]: c=np.zeros([2,1])
   ...: Matrix(np.hstack([A, c])).rref()
Out[2]:
(Matrix([
 [  1, 0, 0],
 [0.0, 1, 0]]),
 (0, 1))
```

The determinant of standard matrix A is not 0, and the rref of that matrix indicates that all column vectors of A are pivot columns. In other words, the standard matrix A is a reversible matrix and has a trivial solution . Therefore, the above transformation is injective.

(Ex. 6.3) Let's decide if the next transformation is injective.

$T: R^3 \rightarrow R^3$
$T(x, y, z) = (x+2y-z, y+z, x+y-2z)$

```
In [1]: A=np.array([[1,2,-1], [0,1,1],[1,1,-2]])
   ...: la.det(A)
Out[1]: 0.0
```

```
In [2]: c=np.zeros([3,1])
   ...: Matrix(np.hstack([A, c])).rref()
Out[2]:
(Matrix([
 [  1, 0, -3.0, 0],
 [0.0, 1,  1.0, 0],
 [  0, 0,    0, 0]]),
 (0, 1))
```

The determinant of standard matrix A is 0, and the rref of the matrix contains free variables. That is, it is an irreversible matrix and has a non trivial solution. As a result, it is not injective as it can have multiple results for elements in the domain.

From the examples above, the following definitions are all equivalent.

ker T=0
⇔ This transformation is injective.
⇔ It has a tivial solution and is linear independent.

Consider the following system of equations.

$$\begin{aligned} a_1x+a_2y+a_3z &= 0 \\ a_4x+a_5y+a_6z &= 0 \\ a_7x+a_8y+a_9z &= 0 \end{aligned} \rightarrow \begin{bmatrix} a_1 & a_2 & a_3 \\ a_4 & a_5 & a_6 \\ a_7 & a_8 & a_9 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix}$$

If the above system is linearly independent and the following conditions are established, this transformation is bijective .

# of variable
= # of solution
= dimension of column space

This means that the column vectors of the standard matrix are all basis vectors. Since there is one solution for each variable, it is injective.

# of equaiton
= # of solution

= dimension of row space

It means that all the row vectors of the standard matrix are the basis vectors, and that the codomain (# of functions (expression)), and the range (solution) coincide, meaning bijective.

(Ex. 6.4) Let's decide how to respond between domain and codomain by the following matrix.

T(<x, y, z>) = <x, y>
T = [<1, 0>, <0, 1>, <0, 0>]

The above transformation can be expressed as a linear combination as follows:

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} x \\ y \end{bmatrix}$$

The above expression is satisfied for all numbers in z . In other words, it is not injective because there are various solutions. However, the number of equations and the number of solutions are the same. In other words, it is subjective because it has the same codaomian and range.

Not all subjects are injective. However, to be injective, it must be subjective. That is, in order for the number of variables to match the number of solutions, the number of expressions must be equal to or greater than the number of solutions.

The following explanations all indicate that matrix A is a reversible matrix.

Column and row vectors of A are linearly independent.
⇔ Ax = 0 has a trivial solution.
⇔ Ax = b has a unique solution.
⇔ A is reversible and det (A) ≠ 0.
⇔ λ = 0 is not an eigenvalue of A.
⇔ T is bijective.

6.2 Linear transformation

Linear transformation (T: U → V) is a function to move one vector space U to another vector space V . These functions must satisfy Eq. 6.3 and 4.

All u_1 , $u_2 \in U \rightarrow T(u_1 + u_2) = T(u_1) + T(u_2)$ (Eq. 6.3)
All $u \in U$ and $\alpha \in C \rightarrow T(\alpha u) = \alpha T(u)$ (Eq. 6.4)
U:vector
C: scalar

Eq. 6.3 and 4 are the same as the linear combination of vectors. This means that vectors with linear combinations can be linearly transformed.

(Ex. 6.5) Determine whether the following equation is a linear transformation.

$T(<x_1 , x_2 , x_3 >) = <2x_1 + x_3 , -4x_2 >$
A=[<2, 0, 1>, <0, -4, 0>]

If the above transformation for $<x_1 , x_2 , x_3 >$ is established, the matrix equation for T_1 and T_2 as the result of the transformation is as follows.

$$\begin{bmatrix} 2 & 0 & 1 \\ 0 & -4 & 0 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} T_1 \\ T_2 \end{bmatrix}$$

Since the linear transformation is included in the linear combination, it can be inferred by determining the linear combination property to the standard matrix A in the above Eq. 6.3 and 4.

```
In [1]: A=np.array([[2,0,1],[0,-4,0]])
   ...: print(A)
[[ 2  0  1]
 [ 0 -4  0]]

In [2]: Matrix(A).rref()
```

```
Out[2]:
(Matrix([
 [1, 0, 1/2],
 [0, 1,   0]]),
 (0, 1))
```

The rref of the standard matrix A is a linearly dependent containing one free variable, and a linear combination is established. Next, Eq. 6.3 and 4 determine if they are established for any three-dimensional vector.

```
In [3]: a=np.array([3, 6, 9])
   ...: b=np.array([2, 5, 8])

In [4]: def trans1(x):
    ...:     x1=2*x[0]+x[2]
    ...:     x2=-4*x[1]
    ...:     return(np.array([x1,x2]))

# T(a)+T(b) = T(a+b)
In [5]: trans1(a+b)==trans1(a)+trans1(b)
Out[5]: array([ True,  True])

# cT(a)=T(ca)
In [6]: c=6
   ...: 6*trans1(a) == trans1(6*a)
Out[6]: array([ True,  True])
```

The above transformation is a linear transformation because it satisfies the two conditions of the transformation.

(Ex. 6.6) Test whether the following equation is a linear transformation.

$$T(<x_1 ,x_2 ,x_3 >)=< 4x_1 +2x_2 , 0, x_1 +3x_3 -2 >$$

The standard matrix A for transform T from the right-hand side of the equation above is the same as object A in the code below. Based on that standard matrix A, a linear combination can be built.

```
In [1]: A=np.array([[4,2,0],
    ...:        [0,0,0],
    ...:        [1,0,3]])
    ...: print(A)
```

```
[[4 2 0]
 [0 0 0]
 [1 0 3]]

In [2]: c=np.array([[0],[0],[2]])
   ...: au=np.hstack([T, c])
   ...: print(au)
[[4 2 0 0]
 [0 0 0 0]
 [1 0 3 2]]

In [3]: Matrix(au).rref()
Out[3]:
(Matrix([
 [1, 0,  3,  2],
 [0, 1, -6, -4],
 [0, 0,  0,  0]]),
 (0, 1))
```

Applying any two vectors to Eq. 6.3 and 4 does not hold.

```
In [4]: a=np.array([3, 6, 9])
   ...: b=np.array([2, 5, 8])

In [5]: def trans2(x):
   ...:     x1=4*x[0]+2*x[1]
   ...:     x2=0
   ...:     x3=x[0]+3*x[2]-2
   ...:     return(np.array([x1,x2, x3]))

In [6]: trans2(a+b)==trans2(a)+trans2(b)
Out[6]: array([ True,  True, False])

In [7]: c=6
   ...: 6*trans2(a) == trans2(6*a)
Out[7]: array([ True,  True, False])
```

Linear combination is a requirement for linear transformation. That is, not all linear combinations satisfy the linear transformation.

As in the examples above, the linear transformation involves a function of a matrix. It can be represented as:

- A function in which both input and output parameters are vectors is called transformation.

• A transformation that satisfies the two conditions of Eq. 6.3 and 4 is called a linear transformation.
• The operation involving a matrix function is called matrix transfomation.

$$T(x) = Ax, \quad x \in \text{Vector}$$

(Ex. 6.7) Determine whether the next matrix transformation is a linear transformation.

T(<x, y, z>)=[< x, y>, <y, -z>]
A for T = [<1, 0>, <1, 1>, <0, -1>]

```
In [1]: A=np.array([[1,1,0],[0,1,-1]])
   ...: print(A)
[[ 1  1  0]
 [ 0  1 -1]]

In [2]: np.dot(A, a)+np.dot(A, b) == np.dot(A, a+b)
Out[2]: array([ True,  True])

In [3]: 3*np.dot(A, a) == np.dot(A, 3*a)
Out[3]: array([ True,  True])
```

Matrix transformation by matrix A is a linear transformation because it satisfies both conditions (Eq. 6.3 and 4).

About vector u
• A transformation that satisfies T (u) = 0 is called zero transformation .
• When it is transformed into itself, such as T (u) = u, it is called identity transformation .

In addition to Eq. 6.3 and 4, the linear transformation has the following characteristics.

Any linear transformation from one vector space to another can be represented by a matrix transformation .

As introduced in Dimensions and Axis in Ch. 1, one axis represents one dimension. Therefore, one row vector is regarded as one dimension. Based

on this, when a 1-dimensional 1 × 5 row vector is transposed, the structure is changed to a 5 × 1-dimensional column vector, which is converted into a vector in a 5-dimensional space with 5 axes. These changes can be visually represented using a base vector that represents only the unit values of each dimension, that is, one row with 1 and the other with 0. This basic vector is a column vector that composes a standard basis. For example, the following column vector can be expressed as a three-dimensional matrix structure that transforms each axis by scalar multiplication.

$$\begin{bmatrix} 3 \\ 5 \\ 7 \end{bmatrix} = 3\begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix} + 5\begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix} + 7\begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix}$$
$$= 3e_1 + 5e_2 + 7e_3$$
$$= \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}\begin{bmatrix} 3 \\ 5 \\ 7 \end{bmatrix}$$

The above e_1, e_2, and e_3 are called standard basis vectors, and by applying them, the original vector can be expressed in the form of scalar × standard basis vector . These transformations can be expressed as generalized as in Eq. 6.5.

$$x = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} = x_1 \cdot e_1 + x_2 \cdot e_2 + \cdots + x_n \cdot e_n \quad \text{(Eq. 6.5)}$$

The transformation of x in Eq. 6.5 can be expressed as Eq. 6.6 as a result of the transformation of each standard baise vector.

$$T(x) = x_1\, T(e_1) + x_2\, T(e_2) + \ldots + x_n\, T(e_n) \quad \text{(Eq. 6.6)}$$

Considering the transformation of each standard base vector in Eq. 6.6, Eq. 6.7 is established.

$$
\begin{aligned}
T(x) &= x_1\begin{bmatrix} a_{11} \\ a_{21} \\ \vdots \\ a_{m1} \end{bmatrix} + x_2\begin{bmatrix} a_{12} \\ a_{22} \\ \vdots \\ a_{m2} \end{bmatrix} + \cdots + x_n\begin{bmatrix} a_{1n} \\ a_{2n} \\ \vdots \\ a_{mn} \end{bmatrix} \\
&= \begin{bmatrix} a_{11}x_1 + a_{12}x_2 + \cdots + a_{1n}x_n \\ a_{21}x_1 + a_{22}x_2 + \cdots + a_{2n}x_n \\ \vdots \\ a_{m1}x_1 + a_{m2}x_2 + \cdots + a_{mn}x_n \end{bmatrix} \\
&= \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{bmatrix}\begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} \\
&= AX
\end{aligned}
\qquad \text{(Eq. 6.7)}
$$

In the conversion process of Eq. 6.7, the n-dimensional vector x can be changed to the m-dimensional AX . The standard matrix for this dimensional transformation can be expressed as Eq. 6.8.

$$T(x) = Ax,\ x \in R^n$$

$$
A = \begin{bmatrix} T(e_1) & T(e_2) & \cdots & T(e_n) \end{bmatrix} = \begin{bmatrix} 1 & 0 & \cdots & 0 \\ 0 & 1 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & 1 \end{bmatrix} \qquad \text{(Eq. 6.8)}
$$

(Ex. 6.8) The following shows the conversion from 2D to 3D (T: $R^2 \rightarrow R^3$). Let's try to determine if vectors b and c hold for transform T.

$T(< x_1 , x_2 >)=< x_1 -3x_2 , 3x_1 +5x_2 , -x_1 +7x_2 >$
A for T = [< 1, 3, -1 >, < -3, 5, 7 >]
b=< 3, 2, -5 >
c=< 3, 2, 5 >

That transformation can be expressed as a linear combination as follows:

$$\begin{bmatrix} 1 & -3 \\ 3 & 5 \\ -1 & 7 \end{bmatrix}\begin{bmatrix} x_1 \\ x_2 \end{bmatrix}=\begin{bmatrix} x_1-3x_2 \\ 3x_1+5x_2 \\ -x_1+7x_2 \end{bmatrix}$$

The above transformation can be expressed as T (u) = Au. The fact that vectors b and c hold for this transformation is equivalent to determining whether the next linear combination is true.

$$Au = b,\ Au = c$$

```
In [1]: A=np.array([[1,-3],[3, 5],[-1,7]])
   ...: print(A)
[[ 1 -3]
 [ 3  5]
 [-1  7]]

In [2]: b=np.array([[3, 2, -5]]).reshape(-1, 1)
   ...: c=np.array([[3, 2, 5]]).reshape(-1, 1)

In [3]: Matrix(np.hstack([A, b])).rref()
Out[3]:
(Matrix([
 [1, 0,  3/2],
 [0, 1, -1/2],
 [0, 0,    0]]),
 (0, 1))

In [4]: Matrix(np.hstack([A, c])).rref()
Out[4]:
(Matrix([
 [1, 0, 0],
 [0, 1, 0],
 [0, 0, 1]]),
 (0, 1, 2))
```

According to the above results, we can calculate the trivial solution from rref of the augment matrix of A and b . Therefore, b is transformed by T.

The three rows of the augment matrix of A and c can be expressed as follows.

$$0x_1 + 0x_2 = 1$$

The above equation cannot be established. In other words, linear combination with vector c does not hold. As a result, conversion by T also does not hold.

(Ex. 6.9) Let T decide if it is a linear transformation.

```
In [1]: T=np.array([[0,-1],[1,0]])
   ...: print(T)
[[ 0 -1]
 [ 1  0]]

In [2]: u=np.array([[4],[1]])
   ...: print(u)
[[4]
 [1]]

In [3]: v=np.array([[2],[3]])
   ...: print(v)
[[2]
 [3]]

In [4]: Tu=np.dot(T, u)
   ...: print(Tu)
[[-1]
 [ 4]]

In [5]: Tv=np.dot(T,v)
   ...: print(Tv)
[[-3]
 [ 2]]

In [6]: Tu+Tv==np.dot(T, (u+v))
Out[6]:
array([[ True],
       [ True]])
```

The above result shows T(u)+T(v)=T(u + v). Therefore, the transform T is linear. The results can be represented geometrically, as shown in Fig. 6.3.

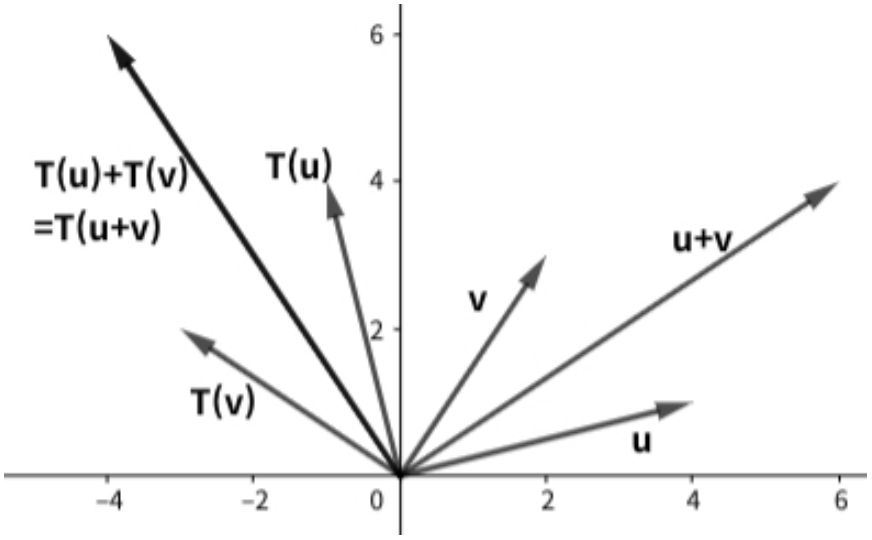


Fig. 6.3. Linear transformation of two vectors.

(Ex. 6.10) The vector b in the R^4 dimension by transform A determines all the vectors corresponding to the 0 vector.

$$A=\begin{bmatrix} 1 & -3 & 5 & -5 \\ 0 & 1 & -3 & 5 \\ 2 & -4 & 4 & -4 \end{bmatrix}$$

$$\rightarrow \begin{bmatrix} 1 & -3 & 5 & -5 \\ 0 & 1 & -3 & 5 \\ 2 & -4 & 4 & -4 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ w \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix}$$

```
In [1]: A=np.array([[1,-3,5,-5],
   ...:             [0,1,-3,5],
   ...:             [2,-4,4,4]])
   ...: print(A)
[[ 1 -3  5 -5]
 [ 0  1 -3  5]
 [ 2 -4  4  4]]

In [2]: c=np.array([0,0,0]).reshape(-1,1)
   ...: print(c)
[[0]
 [0]
 [0]]
```

```
In [3]: au=np.hstack([A,c])
    ...: print(au)
[[ 1 -3  5 -5  0]
 [ 0  1 -3  5  0]
 [ 2 -4  4  4  0]]

In [4]: Matrix(au).rref()
Out[4]:
(Matrix([
 [1, 0, -4, 0, 0],
 [0, 1, -3, 0, 0],
 [0, 0,  0, 1, 0]]),
 (0, 1, 3))
```

In the above result, since z is a free variable, various solutions exist depending on the value. The solution vector b is represented as follows.

```
b = < x, y, z, w > = < 4z, 3z, z, 0 >
```

In this case, it is not monotonous because it is not a one-to-one correspondence, but is subjective because the number of expressions and the number of results are the same. In other words, this linear combination is linearly dependent.

6.2.1 Special Linear Transform

Transformation in vector space is a different representation of a function that transfers to another vector space. Among these transformations, the special linear transformations that satisfy some of the characteristics are:

6.2.1.1 Linear transformation in the same dimension

Ab = c

$$A = \begin{bmatrix} 0.2 & 0.5 \\ 0.4 & 0.7 \end{bmatrix}, \; b = \begin{bmatrix} 3 \\ 2 \end{bmatrix}$$

```
In [1]: A=np.array([[0.2, 0.5],[0.4, 0.7]])
   ...: b=np.array([[3],[2]])
   ...: c=np.dot(A, b)
   ...: print(c)
[[1.6]
 [2.6]]
```

In the above process, vector b is moved to vector c by standard matrix A . In other words, on the normal scale x and y coordinates, b is moved based on each column vector of matrix A . This transformation takes place within the same dimension and is a shift from normal coordinates to the coordinate system where matrix A is the reference (Fig. 6.4).

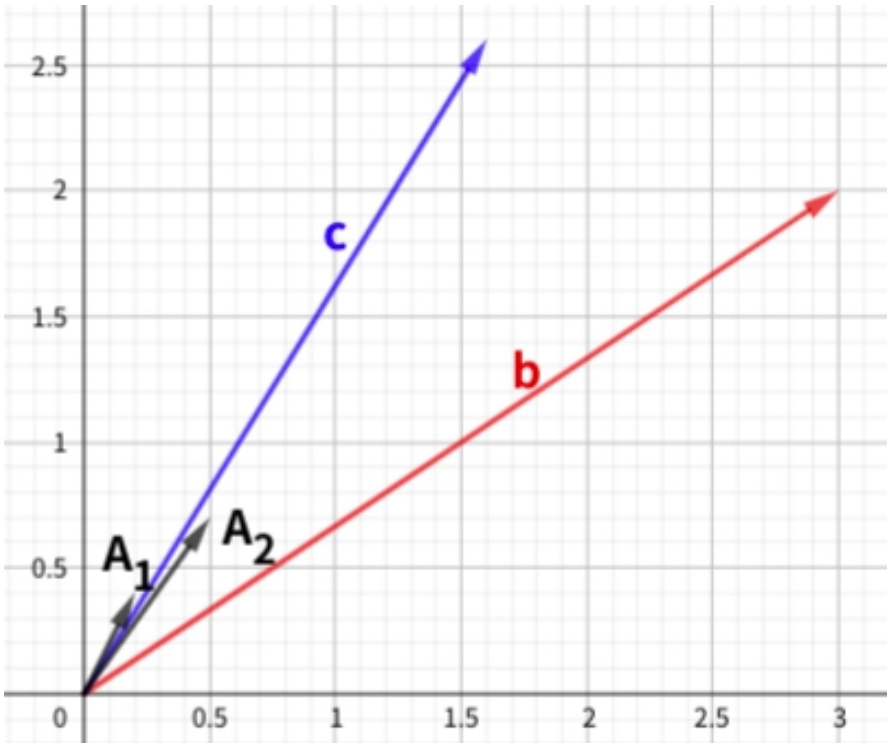


Fig. 6.4 Conversion to the same dimension.

6.2.1.2 Shifting a certain angle

The standard matrix in Eq. 6.9 is a linear transformation in the same dimension that rotates the vector counterclockwise by θ around the origin.

$$R_\theta = \begin{bmatrix} \cos(\theta) & -\sin(\theta) \\ \sin(\theta) & \cos(\theta) \end{bmatrix} \quad \text{(Eq 6.9)}$$

This is a vector b converted to 45° .

$R_\theta\ b = c$

np.sin() and np.cos() are applied to calculate the value of the angle. This function must be passed in radians as an argument. Therefore, the np.deg2rad() function was applied together to convert the angle to radians.

```
In [1]: rad=np.cos(np.deg2rad(45))
   ...: round(rad, 3)
Out[1]: 0.707

In [2]: R=np.array([[np.cos(rad), -np.sin(rad)],
   ...:        [np.sin(rad), np.cos(rad)]])
   ...: print(np.around(R, 3))
[[ 0.76 -0.65]
 [ 0.65  0.76]]

In [3]: b=np.array([[3],[2]])
   ...: c=np.dot(R, b)
   ...: print(np.around(c,3))
[[0.981]
 [3.469]]
```

The above conversion result is shown in Fig. 6.5. The two column vectors that make up the standard matrix R_θ are orthogonal to each other. The two vectors are the base vectors, and we have created vector c with vector b shifted 45° relative to these vectors. This represents the general coordinate system as the new coordinate system based on each column

vector of matrix A as above. As a result, it is the same dimension but converted to a different vector space.

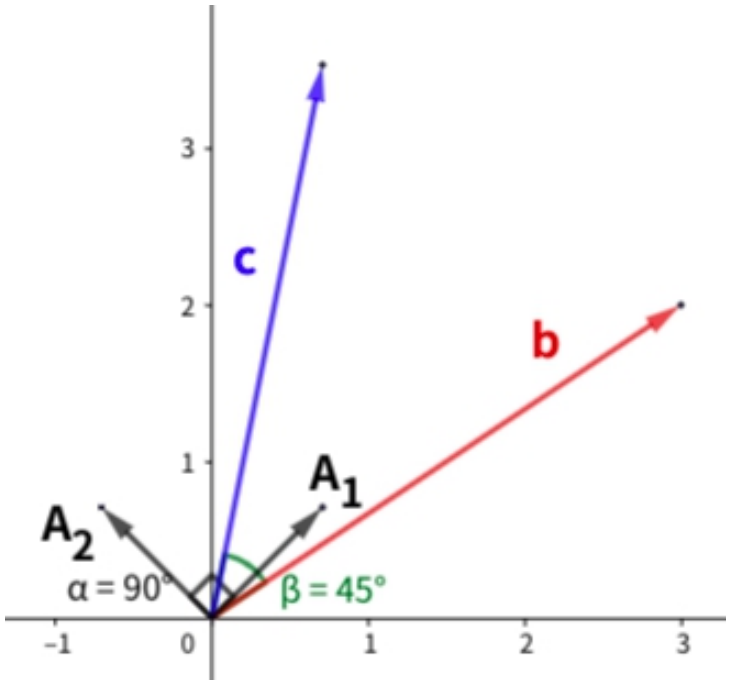


Fig. 6.5 Vector rotated 45° .

6.3 Orthogonal set and projection

6.3.1 Orthogonal set

If the dot product of a vector is 0, as in Eq. 6.10, the vector is orthogonal (see Orthogonal Vector). The set $\{u_1 , u_2 , ..., u_p \}$ of all vectors with zero dot product in the R^n dimension is called an orthogonal set.

$u_i \cdot u_j = 0, i \neq j$ (Eq 6.10)

(Ex. 6.11) Determine if the next vector set is an orthogonal set.

$$u_1=\begin{bmatrix} 3 \\ 1 \\ 1 \end{bmatrix}, u_2=\begin{bmatrix} -1 \\ 2 \\ 1 \end{bmatrix}, u_3=\begin{bmatrix} -1/2 \\ -2 \\ 7/2 \end{bmatrix}$$

```
In [1]: u1=np.array([3,1,1]).reshape(3,1)
   ...: u2=np.array([-1,2,1]).reshape(3,1)
   ...: u3=np.array([-1/2,-2,7/2]).reshape(3,1)
   ...: u1u2=np.dot(u1.T, u2)
   ...: print(u1u2)
[[0]]

In [2]: u1u3=np.dot(u1.T, u3)
   ...: print(u1u3)
[[0.]]

In [3]: u2u3=np.dot(u2.T,u3)
   ...: print(u2u3)
[[0.]]
```

According to the above results, all three products are orthogonal because the dot product between the vectors is all zero. The results can be seen in Fig. 6.6 showing the three vectors.

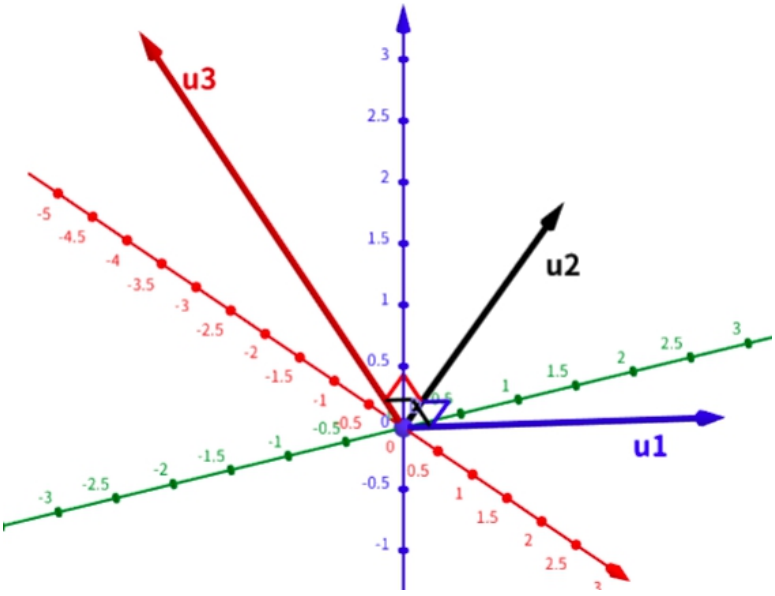


Fig. 6.6 Three vectors in orthogonal relationship.

Let's examine the linear combination of the above orthogonal sets.

```
In [4]: U=np.hstack([u1,u2,u3])
   ...: print(U)
[[ 3.  -1.  -0.5]
 [ 1.   2.  -2. ]
 [ 1.   1.   3.5]]

In [5]: c=np.zeros([3,1])
   ...: print(c)
[[0.]
 [0.]
 [0.]]

In [6]: au=np.hstack([U,c])
   ...: print(au)
[[ 3.  -1.  -0.5  0. ]
 [ 1.   2.  -2.   0. ]
 [ 1.   1.   3.5  0. ]]
```

```
In [7]: Matrix(au).rref()
Out[7]:
(Matrix([
 [1, 0, 0, 0],
 [0, 1, 0, 0],
 [0, 0, 1, 0]]),
 (0, 1, 2))
```

The rref of the augmentation matrix au has three pivot columns. In other words, the standard matrix U is the base matrix, and all variables of the above linear combination are leading variables . Therefore, linear combination is linear independence with a trivial solution.

These results can be summarized as follows.

If $S=\{u_1, u_2, ..., u_p\}$ composed of non-zero vectors is an orthogonal set, then S is linear independent and is the basis for the subspace of S.

$$0 = c_1 u_1 + c_2 u_2 + \dots + c_p u_p$$
$$0 \cdot u_1 = (c_1 u_1 + c_2 u_2 + \dots + c_p u_p) \cdot u_1$$
$$= c_1 (u_1 \cdot u_1) + \dots + c_p (u_p \cdot u_1)$$
$$= c_1 (u_1 \cdot u_1)$$

∵ Since they are all orthogonal, the dot product of each vector is zero.

$c_1, ..., c_p$: Scalar

Therefore, S is linear independent.

In the above equation, since $u_1, u_2, ..., u_p$ are the basis, the scalar value can be calculated as:

$$y \cdot u_1 = (c_1 u_1 + c_2 u_2 + \dots + c_p u_p) \cdot u_1$$
$$= c_1 (u_1 \cdot u_1) + \dots + c_p (u_p \cdot u_1)$$
$$= c_1 (u_1 \cdot u_1)$$
$$\therefore c_1 = \frac{y \cdot u_1}{u_1 \cdot u_1}$$

The above result can be generalized as in Eq. 6.11.

$$c_j = \frac{y \cdot u_j}{u_j \cdot u_j} \quad j=1,\cdots,p \qquad \text{(Eq. 6.11)}$$

(Ex. 6.12) Is it possible to linearly combine the vector y and matrix U in Ex. 6.11?

$$y = \begin{bmatrix} 6 \\ 1 \\ -8 \end{bmatrix}$$

$$= c_1u_1+c_2u_2+c_3u_3$$

$$= \begin{bmatrix} 3 & -1 & -1/2 \\ 1 & 2 & -2 \\ 1 & 1 & 7/2 \end{bmatrix} \begin{bmatrix} c_1 \\ c_2 \\ c_3 \end{bmatrix}$$

The problem is to determine the solution of the equation. The standard matrix of this equation is a base vector, so there is an obvious solution.

```
In [1]: print(U)
[[ 3.  -1.  -0.5]
 [ 1.   2.  -2. ]
 [ 1.   1.   3.5]]

In [2]: y=np.array([6,1,-8]).reshape(-1,1)
   ...: print(y)
[[ 6]
 [ 1]
 [-8]]

In [3]: c=la.solve(U, y)
   ...: print(c)
[[ 1.]
 [-2.]
 [-2.]]
```

As a result,

Linear combination by a standard matrix of orthogonal vectors is linear independent and a one-to-one response, or injective.

6.3.2 Orthogonal Projection

In Fig. 6.7, the vector z can be represented as the sum of two vectors.

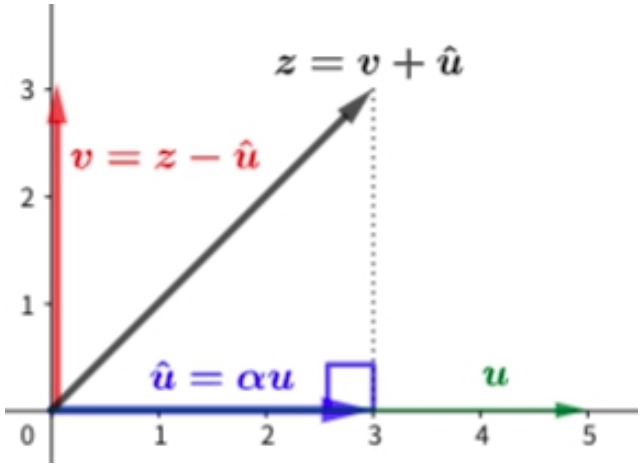


Fig. 6.7 Orthogonal decomposition of vectors.

In Fig. 6.7, û is the α-fold (scalar-fold) of the vector u . To satisfy the above expression for vector z , vector v and vector û must be orthogonal. In other words,

$$
\begin{aligned}
0 &= v\cdot \hat{u} \\
&= (z-\alpha u)\cdot u \\
&= z\cdot u-(\alpha u)\cdot u \\
&= z\cdot u-\alpha(u\cdot u)
\end{aligned}
$$

û can be expressed using Eq. 1.16 to calculate the projection. That is, û is calculated as:

$$\hat{u} = \frac{z\cdot u}{||u||^2}\cdot u = \frac{z\cdot u}{u\cdot u}\cdot u$$

Substituting $\hat{u}$ for $z = v + \hat{u}$, the vector v is calculated as in Eq. 6.12.

$$v = z - \frac{z \cdot u}{u \cdot u} \cdot u \qquad \text{(Eq. 6.12)}$$

The part where z is orthogonally projected onto the vector u is $\hat{u}$. Similarly, the vector v is the part of the orthogonal projection of z on the y-axis and is called the component of z orthogonal to the vector u .

(Ex. 6.13) If the vector z is an orthogonal projection to the vector u, determine the vector perpendicular to u.

z=< 7, 6 >
u=< 4, 2 >

```
In [1]: z=np.array([[7],[6]])
   ...: print(z)
[[7]
 [6]]

In [2]: u=np.array([[4],[2]])
   ...: print(u)
[[4]
 [2]]

In [3]: zu=np.dot(z.T, u)
   ...: print(zu)
[[40]]

In [4]: uu=np.dot(u.T,u)
   ...: print(uu)
[[20]]

In [5]: hat_u=zu/uu*u
   ...: print(hat_u)
[[8.]
 [4.]]

In [6]: v=z-hat_u
   ...: print(v)
[[-1.]
 [ 2.]]
```

Fig. 6.8 shows the results of this problem.

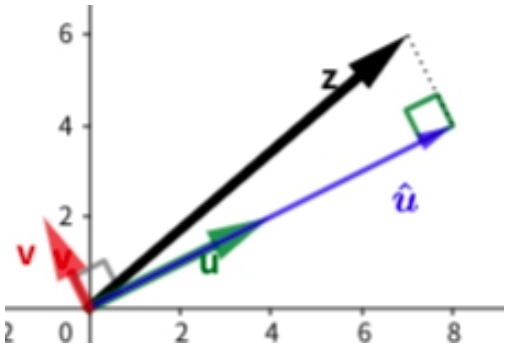


Fig. 6.8 Orthogonal decomposition of vector z .

As shown in Fig. 6.8, an vector can be decomposed into two vectors perpendicular to each other using orthogonal projection.

6.3.3 Orthonormal

Unit vectors orthogonal to each other are called orthonormal. Unit vectors orthogonal to each other are called orthonormal and become basis vectors. Orthonormal set $\{u_1 , u_2 , \ldots, u_p \}$ is the span of the space(W) composed of orthogonal vectors. It can be represented as:

$$W = \text{Span } \{u_1 , u_2 , \ldots, u_p \}$$

Since the linear combination of these vectors is linearly independent, it has a trivial solution. As a result, the orthonormal set is the orthonormal basis for W.

(Ex. 6.14) The next three vectors are the orthonormal basis of R^3 .

$$v_1=\begin{bmatrix} \frac{3}{\sqrt{11}} \\ \frac{1}{\sqrt{11}} \\ \frac{1}{\sqrt{11}} \end{bmatrix}, v_2=\begin{bmatrix} -\frac{1}{\sqrt{6}} \\ \frac{2}{\sqrt{6}} \\ \frac{1}{\sqrt{6}} \end{bmatrix}, v_3=\begin{bmatrix} -\frac{1}{\sqrt{66}} \\ -\frac{4}{\sqrt{66}} \\ \frac{7}{\sqrt{66}} \end{bmatrix}$$

```
In [1]: v1=np.array([3/11**0.5,
   ...:     1/11**0.5,
   ...:     1/11**0.5]).reshape(3,1)
   ...: print(v1)
[[0.90453403]
 [0.30151134]
 [0.30151134]]

In [2]: v2=np.array([-1/6**0.5,
   ...:     2/6**0.5,
   ...:     1/6**0.5]).reshape(3,1)
   ...: print(v2)
[[-0.40824829]
 [ 0.81649658]
 [ 0.40824829]]

In [3]: v3=np.array([-1/66**0.5,
   ...:     -4/66**0.5,
   ...:     7/66**0.5]).reshape(3,1)
   ...: print(v3)
[[-0.12309149]
 [-0.49236596]
 [ 0.86164044]]

In [4]: x=[v1, v2, v3]
   ...: y=["v1","v2","v3"]
   ...: for i in range(3):
   ...:     print(F"Norm of {y[i]}:\
   ...:           {np.around(la.norm(x[i]),1)}")
Norm of v1:          1.0
Norm of v2:          1.0
Norm of v3:          1.0

In [5]: for i in range(3):
   ...:     for j in range(i+1, 3):
   ...:         print(F"Dot product of {y[i]} and {y[j]}\
   ...:               :{np.round(np.dot(x[i].T, x[j]), 2)}")
```

```
Dot product of v1 and v2          :[[0.]]
Dot product of v1 and v3          :[[0.]]
```

The three vectors are orthogonal to each other and are all orthonormal vectors of length 1. In this case, the vectors are mutually independent, which is the basis of R^3 .

```
In [6]: V=np.hstack([v1,v2,v3])
   ...: print(np.around(V,3))
[[ 0.905 -0.408 -0.123]
 [ 0.302  0.816 -0.492]
 [ 0.302  0.408  0.862]]

In [7]: Matrix(V).rref()
Out[7]:
(Matrix([
 [1, 0, 0],
 [0, 1, 0],
 [0, 0, 1]]),
 (0, 1, 2))
```

A matrix consisting of a orthonormal base set has the same transpose and inverse matrix.

```
In [8]: np.around(V.T,2)==np.around(la.inv(V),2)
Out[8]:
array([[ True,  True,  True],
       [ True,  True,  True],
       [ True,  True,  True]])
```

As shown in Eq. 6.13, if the m × n matrix U has the same transpose and inverse matrix, the column vectors of this matrix are a set of orthonormal vectors.

$$U^T \cdot U = I \Leftrightarrow U^T = U^{-1} \quad \text{(Eq. 6.13)}$$

These features are used to judge orthonormal vectors.

$$\{a,b,c\}=\left\{\begin{bmatrix} a_1 \\ a_2 \\ a_3 \end{bmatrix}, \begin{bmatrix} b_1 \\ b_2 \\ b_3 \end{bmatrix}, \begin{bmatrix} c_1 \\ c_2 \\ c_3 \end{bmatrix}\right\}$$

The matrix U is made up of the orthonormal vectors a, b , and c . This matrix is called a orthonormal matrix.

$$U = \begin{bmatrix} a_1 & b_1 & c_1 \\ a_2 & b_2 & c_2 \\ a_3 & b_3 & c_3 \end{bmatrix}$$

Since the matrix U is a orthonormal matrix, $U^T \cdot U = I$ holds.

$$\begin{bmatrix} a_1 & a_2 & a_3 \\ b_1 & b_2 & b_3 \\ c_1 & c_2 & c_3 \end{bmatrix}\begin{bmatrix} a_1 & b_1 & c_1 \\ a_2 & b_2 & c_2 \\ a_3 & b_3 & c_3 \end{bmatrix}$$
$$= \begin{bmatrix} a_1^2+a_2^2+a_3^2 & a_1b_1+a_2b_2+a_3b_3 & a_1c_1+a_2c_2+a_3c_3 \\ a_1b_1+a_2b_2+a_3b_3 & b_1^2+b_2^2+b_3^2 & b_1c_1+b_2c_2+b_3c_3 \\ a_1c_1+a_2c_2+a_3c_3 & b_1c_1+b_2c_2+b_3c_3 & c_1^2+c_2^2+c_3^2 \end{bmatrix}$$
$$= \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

The above is reorganized as follows.

$$\begin{bmatrix} a^T\cdot a & a^T\cdot b & a^T\cdot c \\ a^T\cdot b & b^T\cdot b & b^T\cdot c \\ a^T\cdot c & b^T\cdot c & c^T\cdot c \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

From the above results, the following can be deduced.

$$a^T \cdot a = b^T \cdot b = c^T \cdot c = 1$$
$$a^T \cdot b = a^T \cdot c = b^T \cdot c = 0$$

The following holds between the m × n matrix U of orthonormal vectors and some vectors x and y in the R^n dimension.

- $\|Ux\| = \|x\|$
- $(Ux) \cdot (Uy) = x \cdot y$
- $x \cdot y = 0 \rightarrow (Ux) \cdot (Uy) = 0$

(Ex. 6.15) Check the rules mentioned above for the next orthonormal matrix U.

$$U = \begin{bmatrix} \sqrt{1/2} & 2/3 \\ \sqrt{1/2} & -2/3 \\ 0 & 1/3 \end{bmatrix}$$

```
In [1]: U=np.array([[1/2**0.5, 2/3],
   ...:        [1/2**0.5, -2/3],
   ...:        [0, 1/3]])
   ...: print(np.around(U, 3))
[[ 0.707  0.667]
 [ 0.707 -0.667]
 [ 0.     0.333]]

In [2]: UTU=np.dot(U.T, U)
   ...: print(np.round(UTU, 3))
[[1. 0.]
 [0. 1.]]

In [3]: x=np.array([[2**0.5],[3]])
   ...: y=np.array([[4], [6]])
   ...: Ux=np.dot(U,x)
   ...: Uy=np.dot(U, y)

#‖Ux‖=‖x‖
In [4]: la.norm(Ux)==la.norm(x)
Out[4]: True

#(Ux)·(Uy)=x·y
In [5]: UxUy=np.dot(Ux.T,Uy)
   ...: print(UxUy)
[[23.65685425]]
```

```
In [6]: xy=np.dot(x.T,y)
   ...: print(xy)
[[23.65685425]]

In [7]: UxUy==xy
Out[7]: array([[False]])

In [8]: np.around(UxUy, 3)==np.around(xy, 3)
Out[8]: array([[ True]])
```

The codes [7] and [8] , which are calculation result for UxUy == xy, are the opposite. This is a problem derived from the method of recognizing the value of the decimal point by the binary_ ethod used in computer operations, and it is necessary to specify a certain number of digits after the decimal point.

6.3.4 Gram-Schmidt Process

The Gram Schmidt process is a simple algorithm that generates an orthogonal or orthonormal basis in a subspace of R^n that is not zero.

(Ex. 6.16) As shown in Fig. 6.9, when W = Span $\{x_1 , x_2 \}$, determine the orthogonal basis $\{v_1 , v_2 \}$ in W.

$x_1 =< 3, 6, 0 >$
$x_2 =< 1, 2, 2 >$

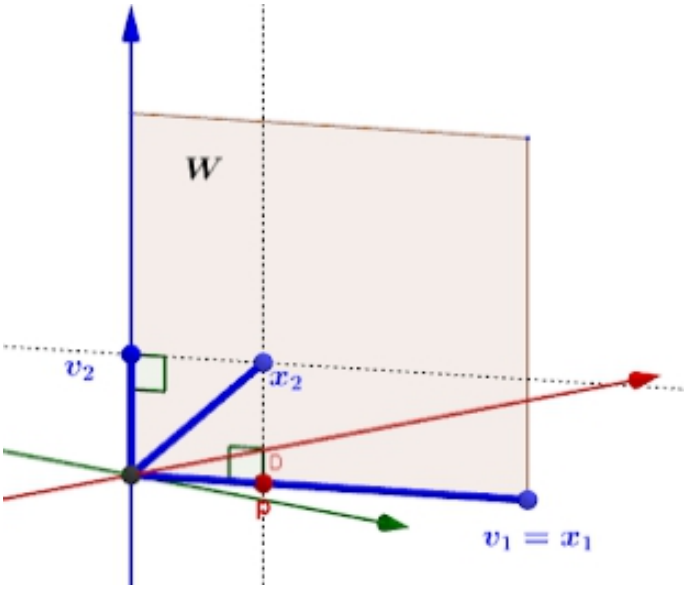


Fig 6.9 Orthogonal projection of two vectors.

As shown in Fig. 6.9, the orthogonal basis of space W can be represented by v_1, v_2, and vector x_2 is projected orthogonally onto vector x_1. In other words, you can assume the relationship $v_1 = x_1$, so x_2 can be decomposed as:

$$x_2 = v_2 + p$$

In the above equation, p is calculated by applying Eq. 6.12 and 13 using the orthogonal basis $x_1 = v_1$, and it is calculated as follows.

$$p = \frac{x_2 \cdot x_1}{x_1 \cdot x_1} x_1$$

$$v_2 = x_2 - \frac{x_2 \cdot x_1}{x_1 \cdot x_1} x_1$$

The user-defined function in the following code is for calculating the above expressions.

```
In [1]: x1=np.array([3,6,0]).reshape(-1,1)
   ...: print(x1)
[[3]
 [6]
 [0]]

In [2]: x2=np.array([1,2,2]).reshape(-1,1)
   ...: print(x2)
[[1]
 [2]
 [2]]

In [3]: def orthoCoefS(x, y):
   ...:     x=np.dot(x.T, y)
   ...:     y=np.dot(y.T, y)
   ...:     return(x/y)

In [4]: p=orthoCoefS(x2,x1)*x1
   ...: print(p)
[[1.]
 [2.]
 [0.]]

In [5]: v2=x2-p
   ...: print(v2)
[[0.]
 [0.]
 [2.]]

In [6]: x2==p+v2
Out[6]:
array([[ True],
       [ True],
       [ True]])
```

x_2 is the subspace of the space with basis vectors v_1 , v_2 as Span.

$$x_2 = \text{Span}\{v_1 , v_2 \}$$

The result is a linear combination of v_1 and v_2 where x_2 is generated and must be linearly independent.

```
In [7]: v=np.hstack([x1,v2])
   ...: print(v)
[[3. 0.]
```

```
 [6. 0.]
 [0. 2.]]

In [8]: au=np.hstack([v, x2])
   ...: print(au)
[[3. 0. 1.]
 [6. 0. 2.]
 [0. 2. 2.]]

In [9]: Matrix(au).rref()
Out[9]:
(Matrix([
 [  1, 0, 0.333...],
 [0.0, 1,       1.0],
 [  0, 0,       0]]),
 (0, 1))

In [10]: Matrix(v).columnspace()
Out[10]:
[Matrix([
 [3.0],
 [6.0],
 [0.0]]),
 Matrix([
 [0.0],
 [0.0],
 [2.0]])]

In [11]: la.matrix_rank(v)
Out[11]: 2
```

According to the above result, all columns of matrix v are pivot columns and the dimension and series of the column space of matrix v are equal. In other words, the above equation is linearly independent. Therefore, each column vector of v is a basis vector, which is a span of x_2 .

The calculation process of Ex. 6.16 is organized as follows.

1) $v_1 = x_1$

2) Since x_2 can be expressed as the sum of p, which is orthogonal projection of x_2 over v_2 , and v_1 , v_2 is organized as follows.

$$x_2 = v_2 + p = v_2 + \frac{x_2 \cdot x_1}{x_1 \cdot x_1} x_1 = v_2 + \frac{x_2 \cdot v_1}{v_1 \cdot v_1} v_1$$

$$v_2 = x_2 - \frac{x_2 \cdot x_1}{x_1 \cdot x_1} x_1 = v_1 + \frac{x_2 \cdot v_1}{v_1 \cdot v_1} v_1$$

This is a Gram-Schmit process that computes orthogonal basis vectors in a space. This process can be generalized as in Eq. 6.14. That is, if $\{x_1, x_2, \ldots, x_p\}$ is the basis of the non-zero subspace W in R^n, the orthogonal basis to W, $\{v_1, v_2, \ldots, v_p\}$, can be calculated as:

$$\begin{aligned}
v_1 &= x_1 \\
v_2 &= x_2 - \frac{x_2 \cdot v_1}{v_1 \cdot v_1} v_1 \\
v_3 &= x_3 - \frac{x_2 \cdot v_1}{v_1 \cdot v_1} v_1 - \frac{x_3 \cdot v_2}{v_2 \cdot v_2} v_2 \\
&\vdots \\
v_p &= x_p - \frac{x_2 \cdot v_1}{v_1 \cdot v_1} v_1 - \frac{x_3 \cdot v_2}{v_2 \cdot v_2} v_2 - \cdots - \frac{x_p \cdot v_{p-1}}{v_{p-1} \cdot v_{p-1}} v_{p-1}
\end{aligned} \quad \text{(Eq. 6.14)}$$

(Ex. 6.17) Calculate orthogonal basis vectors from the basis set $\{x_1, x_2, x_3\}$ of W, which is a subspace of R^4.

$$x_1 = \begin{bmatrix} 1 \\ 1 \\ 1 \\ 1 \end{bmatrix}, \quad x_2 = \begin{bmatrix} 0 \\ 1 \\ 1 \\ 1 \end{bmatrix}, \quad x_3 = \begin{bmatrix} 0 \\ 0 \\ 1 \\ 1 \end{bmatrix}$$

```
In [1]: x1=np.array([[1],[1],[1],[1]])
   ...: print(x1)
[[1]
 [1]
```

```
 [1]
 [1]]

In [2]: x2=np.array([[0],[1],[1],[1]])
   ...: print(x2)
[[0]
 [1]
 [1]
 [1]]

In [3]: x3=np.array([[0],[0],[1],[1]])
   ...: print(x3)
[[0]
 [0]
 [1]
 [1]]

In [4]: v1=x1
   ...: print(v1)
 [[1]
 [1]
 [1]
 [1]]

In [5]: v2=x2-orthoCoefS(x2,v1)
   ...: print(v2)
[[-0.75]
 [ 0.25]
 [ 0.25]
 [ 0.25]]

In [6]: v3=x3-orthoCoefS(x3, v1)*v1\
   ...:      -orthoCoefS(x3, v2)*v2
   ...: print(np.around(v3, 3))
[[ 0.   ]
 [-0.667]
 [ 0.333]
 [ 0.333]]
```

6.4 Similarity transformation

If the relationship of Eq. 6.15 is established between the square matrices A and B of the n × n dimension and the reversible matrix P, the two matrices are said to be similar, and this transformation is called a similarity transformation .

$$A = PBP^{-1} \equiv P^{-1} AP = B \quad \text{(Eq. 6.15)}$$

The similarity transformation in Eq. 6.15 can be summarized in the form of a characteristic equation by applying eigenvalues as follows.

$$\begin{aligned} B - \lambda I &= P^{-1} AP - \lambda P^{-1} P \\ &= P^{-1} (AP - \lambda P) \\ &= P^{-1} (A - \lambda I)P \end{aligned}$$

The determinant of the above equation is organized as follows.

$$\begin{aligned} \det(B - \lambda I) &= \det(P^{-1} (AP - \lambda P)) \\ &= \det(P^{-1}) \det((A - \lambda I)) \det(P) \\ &= \det(P^{-1}) \det(P) \det((A - \lambda I)) \\ &= \det(A - \lambda I) \\ \because \det(P^{-1}) \det(P) &= \det(P^{-1} P) \\ &= \det(I) \end{aligned}$$

As a result, the two matrices by similar transformation have the same characteristic equation and the same characteristic value.

6.4.1 Diagonalization

In many cases, the eigenvalues and eigenvectors of matrix A are useful for creating useful factorization (decomposition), such as the similarity

transformation represented by equation 6.16. Eq. 6.16 replaces matrix B in Eq. 6.15 with a diagonal matrix D .

$$A = PDP^{-1} \quad \text{(Eq. 6.16)}$$

Diagonal matrices are very useful for calculations as follows: For example, the power of a diagonal matrix is equal to the power of the diagonal elements of the diagonal matrix.

```
In [1]: A=np.array([[5,0],[0,5]])
   ...: print(A)
[[5 0]
 [0 5]]

In [2]: np.dot(A,A)
Out[2]:
array([[25,  0],
       [ 0, 25]])
```

As a result, the exponential power of the diagonal elements can be generalized as Eq. 6.17.

$$D^k = \begin{bmatrix} a^k & 0 \\ 0 & b^k \end{bmatrix} \quad \text{(Eq. 6.17)}$$

Applying the properties of this diagonal matrix makes it easier to calculate the exponential power of a decomposable matrix as shown in Equation 6.16. For example, the following matrix A can be decomposed to $A = PDP^{-1}$, so A^k of the matrix is easily processed by applying Eq. 6.17.

(Ex. 6.18) Examine Eq. 6.16 for the matrices A, P, and D.

$$A = [< 7, -4 >, < 2, 1 >]$$
$$P = [< 1, -1 >, < 1, -2 >]$$
$$D = [< 5, 0 >, < 0, 3 >]$$

```
In [1]: A=np.array([[7,2],[-4,1]])
   ...: print(A)
[[ 7  2]
```

```
 [-4  1]]

In [2]: P=np.array([[1,1],[-1, -2]])
   ...: print(P)
[[ 1  1]
 [-1 -2]]

In [3]: D=np.array([[5,0],[0,3]])
   ...: print(D)
[[5 0]
 [0 3]]

In [4]: print(A==np.dot(np.dot(P, D), la.inv(P)))
[[ True  True]
 [ True  True]]
```

From the above relationship, the following is established.

$$A^2 =(PDP^{-1})(PDP^{-1})=PDP^{-1} PDP^{-1} =PD^2 P^{-1} P$$
$$A^3 =(PD^2 P^{-1} P)(PDP^{-1})=PD^3 P^{-1}$$
$$\cdots$$
$$A^k =PD^k P^{-1}$$

The exponential power of an array object can be calculated using the linalg.matrix_power() function.

```
In [5]: D3=np.array([[5**3, 0],[0, 3**3]])
   ...: print(D3)
[[125   0]
 [  0  27]]

In [6]: A3=np.dot(np.dot(P, D3), la.inv(P))
   ...: print(A3)
[[ 223.   98.]
 [-196.  -71.]]

In [7]: print(la.matrix_power(A, 3))
[[ 223   98]
 [-196  -71]]
```

7 Decomposition

7.1 QR decomposition

In Chapter 6, we applied the Gram-Schmidt process to calculate an orthogonal vector that can decompose a vector. The original matrix can be decomposed into an orthogonal matrix generated by this process and its corresponding matrix. This decomposition is called QR decomposition.

If the m×n-dimensional matrix A is linearly independent, it can be decomposed as in Eq. 7.1.

A=QR (Eq. 7.1)
Q: m×n dimension matrix orthonormal to column space A (Col A)
R: n×n dimensional upper triangular reversible matrix. Diagonal elements are positive.

(Ex. 7.1) Calculate the QR decomposition of the following matrix A .
The column space of matrix A is calculated by columnspace() .

A=[<1,1,1,1>, <0,1, 1,1>, <0,0,1,1>]

```
In [1]: x1=np.array([[1],[1],[1],[1]])
   ...: print(x1)
[[1]
 [1]
 [1]
 [1]]

In [2]: x2=np.array([[0],[1],[1],[1]])
   ...: print(x2)
[[0]
 [1]
 [1]
 [1]]

In [3]: x3=np.array([[0],[0],[1],[1]])
```

```
    ...: print(x3)
[[0]
 [0]
 [1]
 [1]]

In [4]: A=np.hstack([x1,x2,x3])
    ...: print(A)
[[1 0 0]
 [1 1 0]
 [1 1 1]
 [1 1 1]]

In [5]: colA=Matrix(A).columnspace()
    ...: colA
Out[5]:
[Matrix([
 [1],
 [1],
 [1],
 [1]]),
 Matrix([
 [0],
 [1],
 [1],
 [1]]),
 Matrix([
 [0],
 [0],
 [1],
 [1]])]
```

According to the results above, all column vectors of A are basis vectors . The Gram-Schmidt process is applied to calculate the orthogonal basis of each column space (column vector) found above.

To apply Gram-Shcmirt process, user-defined function orthoCoef() was created and applied.

```
In [6]: def orthoCoefS(x, y):
    ...:     x=np.dot(x.T, y)
    ...:     y=np.dot(y.T, y)
    ...:     return(x/y)

In [7]: v1=x1
    ...: print(v1)
[[1]
```

```
 [1]
 [1]
 [1]]

In [8]: v2=x2-orthoCoefS(x2, v1)*v1
   ...: print(v2)
[[-0.75]
 [ 0.25]
 [ 0.25]
 [ 0.25]]

In [9]: v3=x3\
   ...:     -orthoCoefS(x3, v1)*v1\
   ...:         -orthoCoefS(x3, v2)*v2
   ...: print(v3)
[[ 0.        ]
 [-0.66666667]
 [ 0.33333333]
 [ 0.33333333]]
```

Since Q is orthogonal, we modify each vector as a unit vector.

```
In [10]: v1_u=1/la.norm(v1)*v1
   ...: v2_u=1/la.norm(v2)*v2
   ...: v3_u=1/la.norm(v3)*v3
   ...: Q=np.c_[v1_u, v2_u, v3_u]
   ...: print(np.around(Q,3))
[[ 0.5   -0.866   0.   ]
 [ 0.5    0.289  -0.816]
 [ 0.5    0.289   0.408]
 [ 0.5    0.289   0.408]]
```

R can be calculated by applying the above result Q to Eq. 7.1. In other words, the solution R can be calculated from the matrix equation of A = QR. In this process, since A is not a square matrix, an inverse matrix is not used, so it can be calculated using the rref of the augment matrix, which is a combination of Q and A .

$QR = A \rightarrow R = Q^{-1} A$
Augment Matrix = [Q : A]

That is, the above augment matrix AM is as follows.

```
In [11]: AM=np.hstack([Q,A])
    ...: print(np.around(AM, 3))
[[ 0.5   -0.866  0.     1.     0.     0.   ]
 [ 0.5    0.289 -0.816  1.     1.     0.   ]
 [ 0.5    0.289  0.408  1.     1.     1.   ]
 [ 0.5    0.289  0.408  1.     1.     1.   ]]
```

R is calculated using AM .

As shown in the following code, the result from the magnification matrix is a 4×3 matrix, not a square matrix. However, by definition, R must be a square matrix. Excluding the row where all elements are 0 in the rref of AM , it becomes a 3×3 square matrix, and this result is R .

```
In [12]: AM_rref=Matrix(AM).rref()
    ...: print(np.around(np.array(AM_rref[0], dtype=np.float), 3))
[[1.    0.    0.    2.    1.5     1.    ]
 [0.    1.    0.    0.    0.866   0.577]
 [0.    0.    1.    0.    0.      0.816]
 [0.    0.    0.    0.    0.      0.    ]]

In [13]: AM_rref[1]
Out[13]: (0, 1, 2)

In [14]: R=np.array(AM_rref[0][:3,3:], dtype=np.float)
    ...: print(np.around(R,3))
[[2.    1.5     1.    ]
 [0.    0.866   0.577]
 [0.    0.      0.816]]
```

Let's check the decomposition from these results.

```
In [15]: re=np.dot(Q,R)
    ...: np.place(re, re<1e-10, 0)
    ...: print(re)
[[1. 0. 0.]
 [1. 1. 0.]
 [1. 1. 1.]
 [1. 1. 1.]]

In [16]: print(A==re)
[[ True  True  True]
 [ True  True  True]
```

```
 [ True  True  True]
 [ True  True  True]]
```

The QR decomposition of the matrix can be calculated using the np.linalg.qr() function.

This function returns two results: Q and R as tuples. In addition, it may have the opposite sign from the result calculated above, but the final result is the same because the changed sign is commonly applied to Q and R.

```
In [17]: Q,R=la.qr(A)
    ...: print(np.around(Q, 3))
[[-0.5    0.866  0.   ]
 [-0.5   -0.289  0.816]
 [-0.5   -0.289 -0.408]
 [-0.5   -0.289 -0.408]]

In [18]: print(np.around(R, 3))
[[-2.    -1.5   -1.   ]
 [ 0.    -0.866 -0.577]
 [ 0.     0.    -0.816]]
```

7.2 Eigen-Decomposition

If a matrix A can be similarly transformed using the reversible matrix P and the diagonal matrix D , as in Equation 6.16, the matrix is said to be diagonalizable.

As shown in Eq. 5.5 , the following relationship holds between the eigenvalues and eigenvectors of a matrix.

$$\begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix}\begin{bmatrix} v_{11} \\ v_{21} \end{bmatrix}=\lambda_1\begin{bmatrix} v_{11} \\ v_{21} \end{bmatrix}$$

$$\begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix}\begin{bmatrix} v_{12} \\ v_{22} \end{bmatrix}=\lambda_2\begin{bmatrix} v_{12} \\ v_{22} \end{bmatrix}$$

If all the eigenvalues and eigenvectors are combined, the eigenvalue matrix becomes diagonal.

$$\begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix}\begin{bmatrix} v_{11} & v_{12} \\ v_{21} & v_{22} \end{bmatrix} = \begin{bmatrix} \lambda_1 & 0 \\ 0 & \lambda_2 \end{bmatrix}\begin{bmatrix} v_{11} & v_{12} \\ v_{21} & v_{22} \end{bmatrix}$$

If the eigenmatrix of the above expression is X , matrix A is arranged in the form of similarity transformation as in Eq. 7.2. This process is called eigendecomposition of the n×n square matrix.

$$AX = DX = XD \rightarrow A = XDX^{-1} \text{ (Eq. 7.2)}$$

Applying the eigenvalue matrix and eigenmatrix to Eq. 7.2 expands as Eq. 7.3.

$$\begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{bmatrix}$$

$$= \begin{bmatrix} x_{11} & x_{12} & \cdots & x_{1n} \\ x_{21} & 22 & \cdots & 2n \\ \vdots & \vdots & \ddots & \vdots \\ x_{n1} & x_{n2} & \cdots & x_{nn} \end{bmatrix} \begin{bmatrix} \lambda_1 & 0 & \cdots & 0 \\ 0 & \lambda_2 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & \lambda_n \end{bmatrix} \begin{bmatrix} x_{11} & x_{12} & \cdots & x_{1n} \\ x_{21} & x_{22} & \cdots & x_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ x_{n1} & x_{n2} & \cdots & x_{nn} \end{bmatrix}^{-1} \quad \text{(Eq. 7.3)}$$

(Ex. 7.2) Decompose by applying the above diagonalization to the following matrix A .

```
A=[<7,-4>, <2,1>]
```

```
In [1]: A=np.array([[7,2],[-4,1]])
   ...: print(A)
[[ 7  2]
 [-4  1]]

In [2]: d, P=la.eig(A)
   ...: print(d)
[5. 3.]

In [3]: print(P)
[[ 0.70710678 -0.4472136 ]
 [-0.70710678  0.89442719]]

In [4]: D=np.diag(d)
   ...: print(D)
[[5. 0.]
 [0. 3.]]

In [5]: P_1=la.inv(P)
   ...: print(P_1)
[[2.82842712 1.41421356]
 [2.23606798 2.23606798]]

In [6]: PDP_1=np.dot(np.dot(P, D), P_1)
   ...: print(PDP_1)
[[ 7.  2.]
 [-4.  1.]]

In [7]: print(A==PDP_1)
```

```
[[ True  True]
 [ True  True]]
```

It is organized as follows.

Diagonalization is possible if the n×n-dimensional square matrix A has n linearly independent eigenvectors. That is, diagonalization is possible when all eigenvectors are base vectors.

It has n eigenvectors.
⇔ There are n corresponding eigenvalues.
⇔ Eigenvalues are the diagonal elements of the diagonal matrix.
⇔ # of column dimensions = # of column spaces = Dimension of column spaces

All column vectors of the matrix are base vectors and are independent.
The inverse matrix exists. ⇔ Rank of matrix A = column dimension

(Ex. 7.3) Determine the diagonalization of the next matrix A , that is, the possibility of eigen-decomposition.

```
A=[<1, -3, 3>, <3, -5, 3>, <3, -3, 1>]
```

```
In [1]: A=np.array([[1,3,3],[-3,-5,-3],[3,3,1]])
   ...: print(A)
[[ 1  3  3]
 [-3 -5 -3]
 [ 3  3  1]]

In [2]: round(la.det(A), 3)
Out[2]: 4.0

In [3]: Matrix(A).columnspace()
Out[3]:
[Matrix([
 [ 1],
 [-3],
 [ 3]]),
 Matrix([
 [ 3],
 [-5],
 [ 3]]),
```

```
Matrix([
[ 3],
[-3],
[ 1]])]
```

Since matrix A has a non-zero determinant, an inverse matrix exists and all column vectors are base vectors belonging to the column space. Therefore, this matrix can be diagonalized and decomposed.

```
In [4]: d, P=la.eig(A)
   ...: D=np.diag(d)
   ...: print(D)
[[ 1.  0.  0.]
 [ 0. -2.  0.]
 [ 0.  0. -2.]]

In [5]: print(P)
[[-0.57735027 -0.78762616  0.42064462]
 [ 0.57735027  0.20744308 -0.81636981]
 [-0.57735027  0.58018308  0.3957252 ]]

In [6]: PDP_1=np.dot(np.dot(P, D), la.inv(P))
   ...: print(PDP_1)
[[ 1.  3.  3.]
 [-3. -5. -3.]
 [ 3.  3.  1.]]

In [7]: print(A==np.round(PDP_1, 0))
[[ True  True  True]
 [ True  True  True]
 [ True  True  True]]
```

7.3 Spectral decomposition

7.3.1 Diagonalization of symmetric matrix

The symmetric matrix has a symmetrical structure with the top and bottom symmetrical with respect to the main diagonal element and has the following characteristics.

As shown in Eq. 7.4, the symmetric matrix is the same as the original matrix and the transpose matrix.

$$A = A^T \quad \text{(Eq. 7.4)}$$

Eigenvectors corresponding to different eigenvalues have an orthogonal relationship and can be an orthonormal relationship

Eq. 7.5 can be derived from $Ax_2 = \lambda_2 x_2$ to indicate that the eigenvectors are orthogonal.

$$Ax_2 = \lambda_1 x_1{}^T x_2 = (\lambda_1 x_1)^T x_2 = (Ax_1)^T x_2 = x_1{}^T A^T x_2 = x_1{}^T Ax_2 = x_1{}^T \lambda_2 x_2 = \lambda_2 x1^T x_2 \quad \text{(Eq. 7.5)}$$

$\because Ax_1 = \lambda x_1 , A = A^T , Ax_2 = \lambda_2 x_2$

From Eq. 7.5

$$\lambda_1 x_1{}^T x_2 - \lambda_2 x_1{}^T x_2 = (\lambda_1 - \lambda_2) x_1{}^T x_2 = 0 \quad \text{(Eq. 7.6)}$$

Since the eigenvalues λ_1 and λ_2 in Eq. 7.6 are different, the dot product between the following vectors must be zero.

$$x_1^T x_2 = 0$$

As in the above equation, the dot product of two vectors is 0, which means they are orthogonal. Normalizing all of these eigenvectors, i.e., unit vectoring, gives an orthonormal relationship.

(Ex. 7.4) Determine the orthogonality between the eigenvectors of vector A.

A=[<6,-2,-1>, <-2,6,-1>, <-1,-1,5>]

```
In [1]: A=np.array([[6,-2,-1],
   ...:         [-2,6,-1],
   ...:         [-1,-1,5]])
   ...: print(A)
[[ 6 -2 -1]
 [-2  6 -1]
 [-1 -1  5]]

In [2]: d, P=la.eig(A)
   ...: print(np.around(P, 2))
[[ 0.58  0.71 -0.41]
 [ 0.58 -0.71 -0.41]
 [ 0.58 -0.    0.82]]

In [3]: x1x2=np.around(np.dot(P[:,0],P[:,1].\
   ...:                       reshape(-1,1)),3)
   ...: print(x1x2)
[-0.]

In [4]: x1x3=np.around(np.dot(P[:,0],P[:,2].\
   ...:                       reshape(-1,1)),3)
   ...: print(x1x3)
[-0.]

In [5]: x2x3=np.around(np.dot(P[:,1],P[:,2].\
   ...:                       reshape(-1,1)),3)
   ...: print(x2x3)
[0.]
```

The reversible symmetric matrix undergoes eigen decomposition as shown in Eq. 7.2. If A is a symmetric matrix, the eigenvector matrix becomes an orthogonal matrix, and the orthogonal matrix can be converted to

an orthonormal matrix. The transposition of this matrix is the same as the inverse matrix. Therefore, Eq. 7.7 holds.

$A = PDP^{-1} = PDP^{T}$ (Eq. 7.7)
$\because P^{T}P=I,\ P^{T}=P^{-1}$

D: Diagonal matrix from eigenvalues
P: Eigenmatrix

The diagonal matrix corresponding to matrix A in Ex. 7.4 is:

```
In [6]: d, P=la.eig(A)
   ...: D=np.diag(d)
   ...: print(D)
[[3. 0. 0.]
 [0. 8. 0.]
 [0. 0. 6.]]
```

The eigenmatrix P of A shown above is an orthogonal matrix and is an orthonormal matrix with a length of 1 for each vector. For this matrix, the transpose and inverse matrices are the same.

```
In [7]: print(np.around(P, 2))
[[ 0.58  0.71 -0.41]
 [ 0.58 -0.71 -0.41]
 [ 0.58 -0.    0.82]]

In [8]: for i in range(P.shape[1]):
   ...:     print(F"Norm of {'P'+str(i)}:\
   ...:           {la.norm(P[:,i])}")
Norm of P0:  1.0
Norm of P1:  1.0
Norm of P2:  1.0

In [9]: print(np.around(P.T, 2))
[[ 0.58  0.58  0.58]
 [ 0.71 -0.71 -0.  ]
 [-0.41 -0.41  0.82]]

In [10]: print(np.around(la.inv(P),2))
[[ 0.58  0.58  0.58]
 [ 0.71 -0.71 -0.  ]
 [-0.41 -0.41  0.82]]
```

```
In [11]: PDP_1=np.dot(np.dot(P, D), la.inv(P))

In [12]: print(PDP_1)
[[ 6. -2. -1.]
 [-2.  6. -1.]
 [-1. -1.  5.]]

In [13]: print(A==np.round(PDP_1, 0))
[[ True  True  True]
 [ True  True  True]
 [ True  True  True]]
```

It is organized as follows.

If matrix A is a symmetric matrix, that matrix is orthogonally diagonalizable.

(Ex. 7.5) Let's check if p in the diagonalization process of the following matrix B is an orthogonal matrix.

```
B=[<6,-2,-1>, <-2,6,-1>, <-1,-1,5>]
```

```
In [1]: B=np.array([[3,-2,4],[-2,6,2],[4,2,3]])
   ...: print(B)
[[ 3 -2  4]
 [-2  6  2]
 [ 4  2  3]]

In [2]: d_b,P_b=la.eig(B)
   ...: D_b=np.diag(d_b)
   ...: print(D_b)
[[ 7.  0.  0.]
 [ 0. -2.  0.]
 [ 0.  0.  7.]]

In [3]: print(np.around(P_b,2))
[[ 0.75 -0.67 -0.14]
 [-0.3  -0.33  0.93]
 [ 0.6   0.67  0.33]]

In [4]: round(la.det(P_b),3)
Out[4]: -0.982
```

```
In [5]: la.matrix_rank(P_b)
Out[5]: 3
```

The eigenmatrix of matrix B has an inverse matrix. Therefore, it has linear independence, that is, a trivial solution. Also, since the rank of the matrix is the same as the column dimension, the eigenmatrix is the base matrix. However, since the eigenvalues corresponding to the eigenvectors P_1 and P_3 are the same, these two vectors are not orthogonal.

```
In [6]: P01=np.around(np.dot(P_b[:,0],P_b[:,1]\
   ...:                    .reshape(-1,1)),3)
   ...: print(P01)
[-0.]

In [7]: P02=np.around(np.dot(P_b[:,0],P_b[:,2].\
   ...:                 reshape(-1,1)),3)
   ...: print(P02)
[-0.191]

In [8]: P03=np.around(np.dot(P_b[:,1],P_b[:,2].\
   ...:                    reshape(-1,1)),3)
   ...: print(P03)
[0.]
```

As in Figure 7.1, the eigenvectors P_1 and P_3 are not orthogonal. These vectors can be decomposed into a vector z_1 orthogonal to P_3 and a vector z_2 orthogonal to P_1 by applying the projection method introduced in Section 6.3.

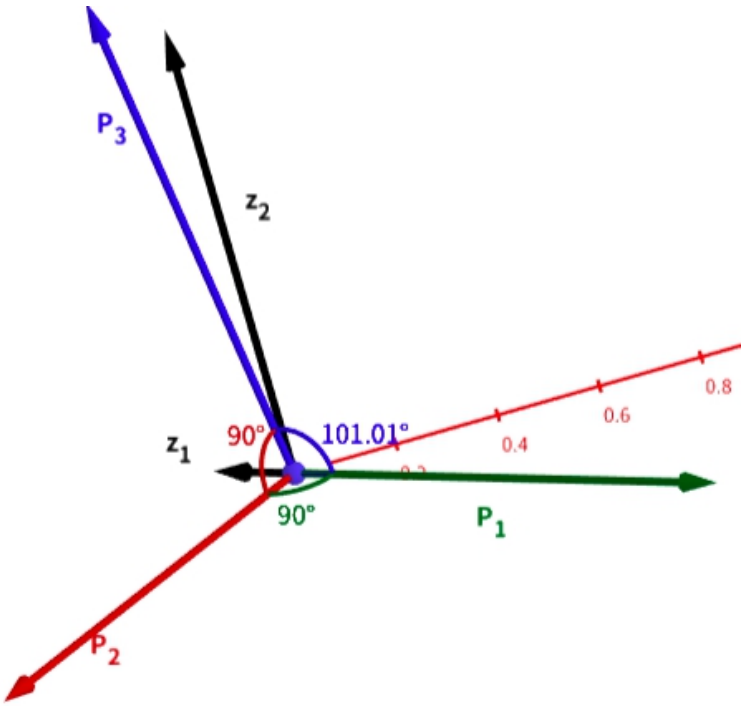


Fig. 7.1 Orthogonal decomposition of two vectors.

```
In [9]: def projectVector(x, y):
   ...:     numer=np.dot(x.T, y)
   ...:     denumer=np.dot(y, y.T)
   ...:     return(numer/denumer*y)

In [10]: z1=projectVector(P_b[:,2], P_b[:,0])
    ...: print(np.around(z1, 2))
[-0.14  0.06 -0.11]

In [11]: z2=P_b[:,2]-z1
    ...: print(np.around(z2, 2))
[-0.    0.88  0.44]

In [12]: print(np.around(np.dot(z1, z2.\
    ...:           reshape(-1,1)), 3))
[-0.]
```

Converting z_1, z_2 to unit vectors and replacing them with P_1, P_3 vectors produces an orthonormal matrix of matrix B. The orthonormal matrix has the same transpose and inverse matrix.

```
In [13]: z11=z1/la.norm(z1)
    ...: z21=z2/la.norm(z2)
    ...: P_n=np.vstack([z11, P_b[:,1], z21]).T
    ...: print(np.round(P_n, 2))
[[-0.75 -0.67 -0.  ]
 [ 0.3  -0.33  0.89]
 [-0.6   0.67  0.45]]

In [14]: P_n1=la.inv(P_n)
    ...: print(np.around(P_n1, 2))
[[-0.75  0.3  -0.6 ]
 [-0.67 -0.33  0.67]
 [ 0.    0.89  0.45]]

In [15]: print(np.around(P_n.T,2))
[[-0.75  0.3  -0.6 ]
 [-0.67 -0.33  0.67]
 [-0.    0.89  0.45]]
```

The eigen decomposition using the generated orthonormal matrix is established.

```
In [16]: P_nDP_n1=np.dot(np.dot(P_n, D_b), P_n1)
    ...: print(P_nDP_n1)
[[ 3. -2.  4.]
 [-2.  6.  2.]
 [ 4.  2.  3.]]

In [17]: print(B)
[[ 3 -2  4]
 [-2  6  2]
 [ 4  2  3]]
```

7.3.2 Spectral decomposition

The set of eigenvalues of matrix A is called the spectrum of A. The spectrum of the symmetric matrix has the following characteristics.

In the symmetric matrix A of n × n dimension
• A has n eigenvalues.
• Each eigenvalue λ and its multiple have the same effect.
⇒ In other words, if multiples of the eigenvalues are applied, only the length of the eigenvector changes, but the direction does not.
• Eigenvectors corresponding to other eigenvalues are orthogonal.
• A can be diagonalized orthogonally.

As an eigenvalue decomposition method, matrix A can be decomposed as shown in Eq. 7.8 by using a diagonal matrix D whose eigenvalues are diagonal elements and an eigenmatrix U consisting of eigenvectors.

$$A = UDU^{-1} = UDU^{T} \text{ (Eq. 7.8)}$$

```
In [1]: A=np.array([[7, 2],[2,4]])
   ...: print(A)
[[7 2]
 [2 4]]

In [2]: d, U=la.eig(A)
   ...: D=np.diag(d)
   ...: print(D)
[[8. 0.]
 [0. 3.]]

In [3]: U
   ...: print(U)
[[ 0.89442719 -0.4472136 ]
 [ 0.4472136   0.89442719]]
```

According to the results above, matrix A is a symmetric matrix and has different eigenvalues, so the eigenmatrix P of the matrix is a normal orthogonal matrix. For a normal orthogonal matrix, Eq. 7.9 holds.

$$U^{T} \cdot U = I \Leftrightarrow U^{T} = U^{-1} \text{ (Eq. 7.9)}$$

```
In [4]: print(np.dot(U, U.T))
[[1. 0.]
 [0. 1.]]
```

```
In [5]: print(la.inv(U)==U.T)
[[False  True]
 [ True  True]]
```

As a result, the eigen decomposition of matrix A is achieved.

```
In [6]: pdp_1=np.dot(np.dot(U, D), U.T)
   ...: print(pdp_1)
[[7. 2.]
 [2. 4.]]
```

Eq. 7.10 is a generalization of the above process.

$$
\begin{aligned}
A &= PDP^T \\
&= [u_1\ u_2\ \cdots\ u_n]\begin{bmatrix} \lambda_1 & 0 & \cdots & 0 \\ 0 & \lambda_2 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & \lambda_n \end{bmatrix}\begin{bmatrix} u_1^T \\ u_2^T \\ \vdots \\ u_n^T \end{bmatrix} \\
&= [\ \lambda_1 u_1\ \lambda_2 u_2\ \cdots\ \lambda_n u_n\ ]\begin{bmatrix} u_1^T \\ u_2^T \\ \vdots \\ u_n^T \end{bmatrix}
\end{aligned}
\qquad \text{(Eq. 7.10)}
$$

$\lambda_1, \lambda_2, \ldots, \lambda_n$: eigenvalues
$u_1, u_2, \ldots, u_n$: eigenvectors

Eq. 7.10 can be expressed as Eq. 7.11.

$$
\begin{aligned}
A &= \lambda_1 u_1 {u_1}^T + \lambda_2 u_2 {u_2}^T + \ldots + \lambda_n u_n {u_n}^T \\
&= u_1 \lambda_1 {u_1}^T + u_2 \lambda_2 {u_2}^T + \ldots + u_n \lambda_n {u_n}^T \\
&= U \cdot D \cdot U^T
\end{aligned}
\qquad \text{(Eq. 7.11)}
$$

The decomposition of the symmetric matrix A as in Eq. 7.11 is called the spectral decomposition .

7.4 Singular Value Decomposition

7.4.1 Quadratic forms

A quadratic expression like $ax^2 + bxy + cy^2$ can be expressed in matrix form as in Eq. 7.12. That is, in the R^n dimension, the quadratic form is defined by the function Q.

$Q(x) = x^T Ax$ (Eq. 7.12)
A : Symmetric matrix

The simplest quadratic form is $Q(x) = x^T Ix = \|x\|^2$. In Q above, the diagonal elements of symmetric matrix A are the coefficients of the quadratic term and the sum of the elements outside the diagonal is the coefficient of the primary terms. For example, the form of quadratic $ax^2 + bxy + cx^2$ can be expressed in the form of a matrix s shown in Eq. 7.13.

$$ax^2 + bxy + cy^2 \rightarrow \begin{bmatrix} a & \frac{b}{2} \\ \frac{b}{2} & c \end{bmatrix}$$ (Eq. 7.13)

In order to represent the quadratic expression as a code, the sympy package, which can consider symbols as numbers, has been applied. The symbols() function in this package can specify symbols as unknown.

```
In [1]: x1,x2=symbols("x1,x2")
   ...: X=Matrix(2,1, [x1, x2])
   ...: print(X)
Matrix([[x1], [x2]])
```

```
In [2]: A=Matrix([[4,0],[0, 3]])
   ...: print(A)
Matrix([[4, 0], [0, 3]])

In [3]: quadForm=X.T*A*X
   ...: print(quadForm)
Matrix([[4*x1**2 + 3*x2**2]])
```

In the above equation, matrix A is the diagonal matrix, and each diagonal element is the coefficient of the quadratic term. When there are diagonal elements:

```
In [1]: x1,x2, x3=symbols("x1,x2, x3")
   ...: X=Matrix(3,1, [x1, x2, x3])
   ...: print(X)
Matrix([[x1], [x2], [x3]])

In [2]: A=Matrix([[4,1,1],[1,3,1],[1,1,2]])
   ...: print(A)
Matrix([[4, 1, 1], [1, 3, 1], [1, 1, 2]])

In [3]: quadForm=X.T*A*X
   ...: print(expand(quadForm))
Matrix([[4*x1**2 + 2*x1*x2 + 2*x1*x3 + 3*x2**2 + 2*x2*x3 + 2*x3**2]])
```

From the results above, the non-diagonal elements represent the coefficient/2 of the crossing term, respectively. For example, Eq. 7.14 is a standard matrix for the coefficients of quadratic equations more complex than Eq. 7.13.

$$ax_1^{\,2} + bx_2^{\,2} + cx_3^{\,3} + dx_1 x_2 + ex_1 x_3 + fx_2 x_3$$

$$A=\begin{bmatrix} a & d/2 & e/2 \\ d/2 & b & f/2 \\ e/2 & f/2 & c \end{bmatrix} \quad \text{(Eq. 7.14)}$$

(Ex. 7. 6) The following quadratic equation is represented as a matrix.

$$Q(x) = 5x_1^{\,2} + 3x_2^{\,2} + 2x_3^{\,2} - x_1 x_2 + 8x_2 x_3$$

```
In [1]: A=np.diag([5.,3.,2.])
   ...: print(A)
[[5. 0. 0.]
 [0. 3. 0.]
 [0. 0. 2.]]

In [2]: A[0,1]=A[1,0]=-1/2
   ...: A[1,2]=A[2,1]=4
   ...: print(A)
[[ 5.  -0.5  0. ]
 [-0.5  3.   4. ]
 [ 0.   4.   2. ]]
```

The quadratic form of the matrix has the same form as the similarity transform . In other words, by adjusting the matrix A above, you can create a variety of similarity matrices. If the result of the eigen decomposition of symmetric matrix A is applied to Equation 7.12, the Q of the original matrix is converted to a simple equation with the cross term removed.

Since the standard (coefficient) matrix A of the quadratic is a symmetric matrix, the eigen matrix becomes an orthogonal base matrix , so Eq. 7.7 holds. Re-expressing this expression is as follows.

$$A = PDP^{-1} = PDP^{T}$$
D: Diagonal matrix with eigenvalues as diagonal elements

In the above equation, P is the base matrix. Therefore, some vector x can be represented in the form of a linear combination with this matrix P.

$$x = Py$$

That is, x is the coordinate vector corresponding to the base matrix P. The above linear combination can be applied to Eq. 7.12 as shown in Eq. 7.15.

$$x^{T} Ax = (Py)^{T} A(Py) = y^{T} P^{T} APy = y^{T} (P^{T} AP)y = y^{T} Dy \quad \text{(Eq. 7.15)}$$
$$\because A = PDP^{-1} \rightarrow P^{-1} AP = D = P^{T} AP$$

(Ex. 7. 7) Let's create a new quadratic equation by removing the cross-product using the quadratic matrix of the following equation.

$$Q(x) = x_1^{\,2} - 8x_1\,x_2 - 5x_2^{\,2}$$

```
In [1]: A=np.diag([1,-5])
   ...: A[0,1]=A[1,0]=-4
   ...: print(A)
[[ 1 -4]
 [-4 -5]]

In [2]: d, P=la.eig(A)
   ...: D=np.diag(d)
   ...: print(D)
[[ 3.  0.]
 [ 0. -7.]]

In [3]: print(P)
[[ 0.89442719  0.4472136 ]
 [-0.4472136   0.89442719]]

In [4]: y1, y2=symbols("y1, y2")
   ...: y=Matrix(2,1,[y1,y2])
   ...: print(y)
Matrix([[y1], [y2]])

In [5]: print(y.T*Matrix(D)*y)
Matrix([[3.0*y1**2 - 7.0*y2**2]])
```

Test Eq. 7.15 for an arbitrary vector x and a vector y resulting from a linear combination with the vector.

```
In [6]: x=np.array([1,2]).reshape(-1,1)
   ...: print(X)
Matrix([[x1], [x2], [x3]])

In [7]: y=np.dot(la.inv(P), x)
   ...: print(np.around(y, 3))
[[0.   ]
 [2.236]]

In [8]: xPx=np.dot(np.dot(x.T, A), x)
   ...: print(xPx)
[[-35]]

In [9]: yDy=np.dot(np.dot(y.T, D), y)
   ...: print(yDy)
[[-35.]]
```

As a result, complex quadratic expressions containing cross products, which are products of other variables, can be converted to a simple form using Eq. 7.15.

The above transformation can convert a vector that follows a complex expression into a vector that follows a simple expression. These transformations are based on quadratic eigenvectors. In other words, as shown in Fig. 7.2, the reference coordinates are converted into a system using eigenvectors.

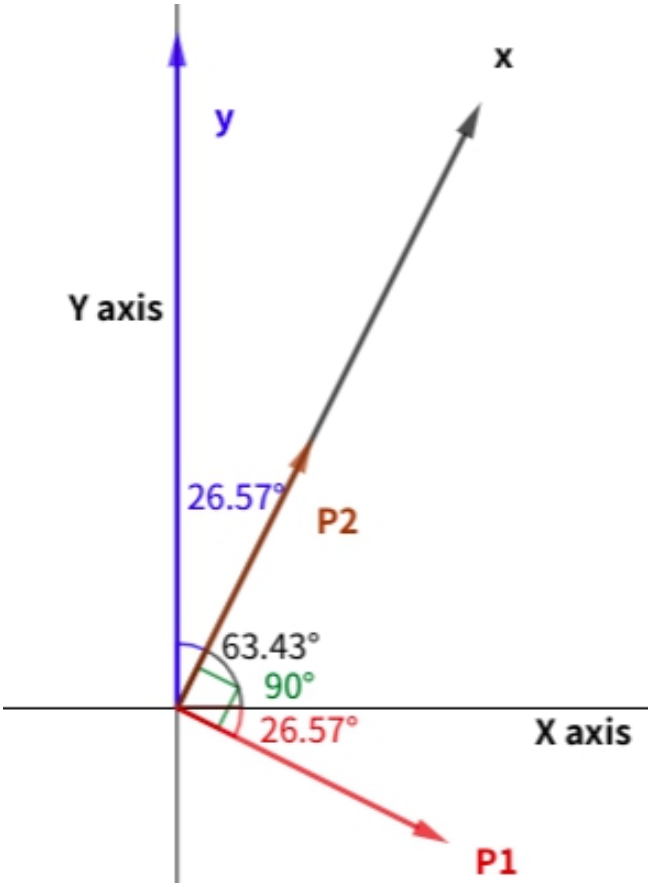


Fig. 7.2 Vector transform by eigen-matrix.

7.4.2 Singular value decomposition

7.4.2.1 Singular value of m×n matrix

As in Eq. 7.16, the relationship of the eigenvector(v) corresponding to the eigenvalue (λ) of $A^T A$ is established, and if the eigenvector is the orthonormal basis , Eq. 7.17 holds because the transpose matrix and the inverse matrix are the same.

$$A^T A v_i = \lambda_i v_i \quad \text{(Eq 7.16)}$$

$$\begin{aligned} \|Av_i\|^2 &= (Av_i)^T (Av_i) \\ &= v_i^T A^T A v_i \\ &= v_i^T (\lambda_i v_i) \\ &= \lambda_i v_i^T v_i \\ &= \lambda_i \quad \text{(Eq 7.17)} \end{aligned}$$

λ_i is not negative.

The square root of the eigenvalue of $A^T A$ is called the singular value of matrix A and is represented by $\sigma_1 , \sigma_2 , \ldots , \sigma_n$. In other words, this relationship is expressed as Eq. 7.18.

$$\sigma_i = \sqrt{\lambda_i} = \|Av_i\|, \quad 1 \le i \le n \quad \text{(Eq. 7.18)}$$

(Ex. 7.8) Determine the singular value of matrix A.

$$A = \begin{bmatrix} 4 & 11 & 14 \\ 8 & 7 & -2 \end{bmatrix}$$

```
In [1]: A=np.array([[4, 11, 14],[8, 7, -2]])
   ...: print(A)
[[ 4 11 14]
 [ 8  7 -2]]

In [2]: AA=np.dot(A.T, A)
   ...: print(AA)
[[ 80 100  40]
 [100 170 140]
 [ 40 140 200]]
```

```
In [3]: d, p=la.eig(AA)
   ...: print(np.around(d, 3))
[360.  -0.  90.]

In [4]: print(np.around(p, 2))
[[-0.33 -0.67 -0.67]
 [-0.67  0.67 -0.33]
 [-0.67 -0.33  0.67]]

In [5]: for i in range(len(d)):
   ...:     if d[i]<-1e-5 and d[i]>1e-5:
   ...:         d[i]=0
   ...: print(d)
[360.   0.  90.]

In [6]: s=np.sqrt(d)
   ...: print(np.around(s,2))
[18.97  0.    9.49]
```

If the $A^T A$'s eigenmatrix is orthonormal , its transpose and inverse matrices must be the same.

```
In [7]: print(np.around(p.T, 2))
[[-0.33 -0.67 -0.67]
 [-0.67  0.67 -0.33]
 [-0.67 -0.33  0.67]]

In [8]: print(np.around(la.inv(p),2))
[[-0.33 -0.67 -0.67]
 [-0.67  0.67 -0.33]
 [-0.67 -0.33  0.67]]
```

However, since the eigenvector corresponding to the eigenvalue 0 is not the basis, there are 2 orthogonal basis vectors among the eigenvectors above. The number of base vectors in the matrix is represented by the rank and the number of column spaces.

```
In [9]: la.matrix_rank(AA)
Out[9]: 2

In [10]: Matrix(A).columnspace()
Out[10]:
[Matrix([
 [4],
```

```
[8]]),
Matrix([
[11],
[ 7]])]
```

From the above results, it can be seen that the eigenvectors p_1 and p_3 are orthogonal, and the linear combination with each vector, Ap_1 and Ap_3, is also orthogonal.

```
In [11]: Ap1=np.dot(A, p[:,0].reshape(-1,1))
    ...: print(Ap1)
[[-18.]
 [ -6.]]

In [12]: Ap2=np.dot(A, p[:,2].reshape(-1,1))
    ...: print(Ap2)
[[ 3.]
 [-9.]]

In [13]: print(np.round(np.dot(Ap1.T, Ap2), 3))
[[-0.]]
```

The results can be organized as follows.

• Eigenvectors v_1 , v_2 , ... corresponding to r non-zero eigenvalues of A^T A. , v_n is the orthonormal basis
• Av_1 , Av_2 ,… , Av_n is the orthonormal basis of Col A (A's column space)
If the above two conditions are met, Rank A becomes r.

7.4.2.2 Singular value decomposition

The decomposition of m × n-dimensional matrix A is related to the diagonal matrix (Σ, D) as in Eq. 7.19.

$$\Sigma=\begin{bmatrix} D & 0 \\ 0 & 0 \end{bmatrix} \quad \text{(Eq. 7.19)}$$

In the above form, D is a diagonal matrix of r × r dimension with a non-zero singular value as a diagonal element. Where r is a number less than or equal to the minimum of m and n, and Σ has the same dimension as A .

If the eigenvectors and eigenvalues of $A^T A$ for matrix A are called v_i and λ_i , respectively, and there are r non-zero eigenvectors, Av_1 , Av_2 ,… as described in the previous section. , Av_r are orthogonal and normalized vectors u_1 , u_2 , ..., u_r are orthonormal. In other words, it can be expressed as Eq. 7.20.

$$u_i=\frac{1}{||Av_i||}Av_i=\frac{1}{\sigma_i}Av_i$$
$$\rightarrow Av_i=\sigma_i u_i \quad 1\leq i\leq r \quad \text{(Eq. 7.20)}$$

The AV in Eq. 7.20 can be expressed as Eq. 7.21 by extending to all parts where the eigenvalue becomes 0.

$$\begin{aligned} AV&=\begin{bmatrix} Av_1 & Av_2 & \cdots & Av_r & 0 & 0 & \cdots & 0 \end{bmatrix} \\ &=\begin{bmatrix} \sigma_1 u_1 & \sigma_2 u_2 & \cdots & \sigma_r u_r & 0 & 0 & \cdots & 0 \end{bmatrix} \\ &=\begin{bmatrix} u_1 & u_2 & \cdots & u_r & 0 & 0 & \cdots & 0 \end{bmatrix} \begin{bmatrix} \sigma_1 & 0 & \cdots & 0 & \cdots & 0 \\ 0 & \sigma_2 & \cdots & 0 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots & \vdots & \vdots \\ 0 & 0 & \cdots & \sigma_r & \cdots & 0 \\ 0 & 0 & \cdots & 0 & \cdots & 0 \end{bmatrix} \end{aligned} \quad \text{(Eq. 7.21)}$$

Since U is an orthogonal matrix, $U^T U = I$ holds, so matrix A is decomposed as in Eq. 7.22.

$$A = U\Sigma V^T \quad \text{(Eq. 7.22)}$$

This decomposition is called singular value decomposition (SVD).

(Ex. 7.9) Perform singular value decomposition of matrix A.

```
A=[<4,8>, <11,7>, <14, -2>]
```

```
In [1]: A=np.array([[4, 11, 14],[8, 7, -2]])
   ...: print(A)
[[ 4 11 14]
 [ 8  7 -2]]

In [2]: AA=np.dot(A.T,A)
   ...: print(AA)
[[ 80 100  40]
 [100 170 140]
 [ 40 140 200]]

In [3]: d, p=la.eig(AA)
   ...: print(np.around(d, 2))
[360.  -0.  90.]

In [4]: print(np.around(p, 2))
[[-0.33 -0.67 -0.67]
 [-0.67  0.67 -0.33]
 [-0.67 -0.33  0.67]]
```

Among the A^T A's eigenvalues(d), the singular value(s) is calculated from non-zero values.

```
In [5]: s=np.sqrt([d[0], d[2]])
   ...: print(np.around(s, 2))
[18.97  9.49]
```

From the singular value above, the Σ matrix Smat is calculated by combining the singular value matrix D and the zero vector.

```
In [6]: D=np.diag(s)
   ...: print(np.around(D, 2))
[[18.97  0.  ]
 [ 0.    9.49]]

In [7]: Smat=np.hstack([D, np.zeros([2,1])])
   ...: print(np.around(Smat, 2))
```

```
[[18.97  0.    0.  ]
 [ 0.    9.49  0.  ]]
```

Compute the orthogonal coordinates from matrix A .

```
In [8]: u1=1/s[0]*np.dot(A, p[:,0].reshape(-1,1))
   ...: print(np.around(u1, 2))
[[-0.95]
 [-0.32]]

In [9]: u2=1/s[1]*np.dot(A, p[:,2].reshape(-1,1))
   ...: print(np.around(u2, 2))
[[ 0.32]
 [-0.95]]
```

The above process can be replaced by the numpy.linalg function svd() .

```
In [10]: A=np.array([[4, 11, 14],[8, 7, -2]])
   ...: U,S,VT=la.svd(A)
   ...: print(np.around(U, 2))
[[-0.95 -0.32]
 [-0.32  0.95]]

In [11]: print(np.around(S, 2))
[18.97  9.49]

In [12]: print(np.around(VT, 2))
[[-0.33 -0.67 -0.67]
 [ 0.67  0.33 -0.67]
 [-0.67  0.67 -0.33]]
```

Check $A=U\Sigma V^T$

```
In [13]: Smat=np.hstack([np.diag(S),np.zeros([2,1])])
   ...: print(np.around(Smat, 2))
[[18.97  0.    0.  ]
 [ 0.    9.49  0.  ]]

In [14]: A1=np.dot(np.dot(U, Smat),VT)
   ...: print(A1)
[[ 4. 11. 14.]
 [ 8.  7. -2.]]
```

```
In [15]: print(np.around(A1, 0)==np.around(A, 0))
[[ True  True  True]
 [ True  True  True]]
```

(Ex. 7.10) Perform the singular value decomposition of matrix B.

```
In [1]: B=np.array([[1,-1],[-2,2],[2,-2]])
   ...: print(B)
[[ 1 -1]
 [-2  2]
 [ 2 -2]]

In [2]: U,S, VT=la.svd(B)
   ...: print(np.around(U, 2))
[[-0.33  0.67 -0.67]
 [ 0.67  0.67  0.33]
 [-0.67  0.33  0.67]]

In [3]: print(np.around(S, 2))
[4.24 0.  ]

In [4]: print(np.around(VT, 2))
[[-0.71  0.71]
 [ 0.71  0.71]]

In [5]: Smat=np.vstack([np.diag(S),\
   ...:      np.zeros([1,2])])
   ...: print(np.around(Smat, 2))
[[4.24 0.  ]
 [0.   0.  ]
 [0.   0.  ]]

In [6]: print(np.around(\
   ...:                np.dot(np.dot(U, Smat), VT), 1)\
   ...:       ==np.dot(B, 1))
[[ True  True]
 [ True  True]
 [ True  True]]
```

Appendix A Functions

Functions and methods of the numpy(np) module

```
import numpy as np
import numpy.linalg as la
from scipy import linalg
from sympy as *
```

arr. ndim
• Return dimension of np object arr

arr. reshape(# of row, # of column)
• If # of elements = # of rows × # of columns
Rearrange arrays (vectors or matrices) according to the number of rows and columns specified
• If the argument specific to a row or column is -1, the dimension changes based on other arguments.

arr. shape
• Returns the number of rows and columns of object arr as a tuple

arr. T
•Transpose the vector a
• Returns the same result as the function np. transpose (a)

la. det (M)
• Calculate the determinant of the matrix M

la. eig (M)
• Return eigenvalues and eigenvectors of matrix M in tuple

la. inv (M)
• Create an inverse matrix of M
• Equivalent to linalg.inv() in the scipy.linalg module.

la. matrix_power (M, k)
• Calculate M^k for matrix M

la. matrix_rank (M)
• Returns the rank of matrix M

la. norm (v)
• Calculate norm of vector v

la. qr (M)
• QR decomposition of matrix M and return Q and R as tuples

la. solve (a, b)
• Calculate the value of unknown x from the equation $a \cdot x = b$

la. svd (M)
• Singular value decomposition of matrix M → $M=U\Sigma V^T$
• Returns the orthonormal matrix (U), singular value, and transpose matrix (V^T) of the eigenmatrix
Σ: Diagonal matrices with the same dimension as M with the singular value as the diagonal element, and all other than singular values are 0.

np. array (x)
• Create array object x

np. allclose (A, B, rtol=1e $^{-05}$, atol=1e $^{-08}$)
• Returns True if all elements of objects A and B are equal, False otherwise
• Each element value is compared based on the relative threshold (rtol) and absolute threshold (atol).
The default values of rtol and atol are compared to 5 digits and 8 digits after the decimal point, respectively.

np. arccos (x)
• Returns the reciprocal of the cos value (y=cos(x) → x=arccos(y))

np. concatenate ((a,b,c,…), axis=0)
• Create a new object that combines multiple array objects
• axis: Specifies the axis to be combined
0: Column direction (default)
1: Row direction

np. cos (rad)
• Returns the cosin value corresponding to the radian value

np. cross (x, y)
• Returns the cross product between vectors x and y
• Vectors x and y must be row vectors.

np. deg2rad (θ)
• Convert θ value to radian value(=np. radians (θ))

np. diag ([values])
• Create a diagonal matrix where the specified values are diagonal elements.

np. diag (M, k=0)
• Returns the diagonal element of the matrix M according to the specified k
• k specifies the diagonal position to return in the matrix
k = 0: Main diagonal (default)

Integer with k > 0: move upwards relative to the main diagonal
Integer with k < 0: move downward relative to the main diagonal

np. dot (a, b)
• Returns the matrix product of array objects a and b

np. equal (a, b)
• Determine that the array objects a and b are the same in elements-wise.

np. eye (# of row or column)
• Create identity matrix of specified number

np. hstack ((a, b, c,…))
• Combine objects in row direction (= np.c_[a, b, c,…])

np. place (x, cond, vals)
• Convert the part corresponding to the condition (cond) in object x to the specified value (vals)

np.random. randint (start, end, size=())
• Generate a random number(integer) of the size specified in the range of [start, end-1]
• size: Specify the size and shape of the result
If it is one integer, the result is returned as a list.
If specified in a tuple format such as "(number of rows, number of columns)", return as an array

np. rad2deg (rad)
• Convert radian value to θ

np. sin (rad)
• Returns the sine value corresponding to the radian value

np. trace (M)
• Calculate the sum of the diagonal elements of the square matrix M

np. triu (M, k=0)
• Create an upper triangular matrix by converting all elements below diagonal elements according to k in matrix M to 0
• For argument k, see np.diag().

np. tril (M, k=0)
• Create a lower triangular matrix by converting all elements above the diagonal elements according to k in matrix M to 0
• For argument k, see np.diag().

np. vstack ((a, b, c,…))
• Combine objects in column direction (= np.r_ [a, b, c,…])

scipy.linalg. linalg.lu (M)
• Perform LU decomposition of matrix M

sympy module functions and methods

obj1. column_join (obj2)
• Combine obj1 and obj2 in column direction

M. columnspace ()
• Returns the column space of matrix M

eye (n)
• Create n-dimensional identity matrix

Matrix (obj)
• Convert object obj to matrix
• For numpy array objects, convert to sympy matrix objects

M. nullspace ()
• Returns the null space of matrix M

obj1. row_join (obj2)
• Combine obj1 and obj2 in row direction

M. rref ()
• Create rref of sympy object.
• The result of this function is returned as a tuple like this:
  (rref, column space column number)

solve (expression or matrix)
• Compute unknown solution to homogeneous equation or matrix = 0

symbols (“symbol(s)”)
• Designate symbol as unknown
• Allows you to calculate symbolic solutions like mathematical equations.

Index

A

B

C

D

E

F

G

I

K

L

M

N

O

P

R

S

T

V

Z

Table of Contents