



# Programming meets Mathematics: **Linear Algebra**

Sunglok Choi, Assistant Professor, Ph.D.  
Computer Science and Engineering Department, SeoulTech  
[sunglok@seoultech.ac.kr](mailto:sunglok@seoultech.ac.kr) | <https://mint-lab.github.io/>

# Programming meets Mathematics

## ~~■~~ ~~Calculus~~ **Differentiation**

- Visualization using [Matplotlib](#)
- Differentiation
  - Meaning: The **slope** of a tangent line ( $\sim$  linear approximation) of a function
  - Finding a **derivative** using [rules](#), [SymPy](#), and numerical differentiation

## ■ **Linear Algebra**

## ■ **Optimization**

## ■ **Probability**

## ■ **Information Theory**

# Getting Started from Line Fitting

(1, 4)



(4, 2)

- Line representation:  $y = wx + b$  ( $y = -\frac{2}{3}x + \frac{14}{3}$ )
- Slope  $w = \frac{2-4}{4-1} = -\frac{2}{3}$
- Y intercept  $b = 4 - m \cdot 1 = \frac{14}{3}$

How about finding a line from three points?

# Getting Started from Line Fitting

(1, 4)



(4, 2)

- Line representation:  $ax + by + c = 0$   
( $2x + 3y - 14 = 0$ ;  $4x + 6y - 28 = 0$ )
- Its shorter form:  $\mathbf{n}^T \mathbf{x} + c = 0$   
( $\mathbf{n} = \begin{bmatrix} a \\ b \end{bmatrix}$  and  $\mathbf{x} = \begin{bmatrix} x \\ y \end{bmatrix}$ )

Normal vector



Can it represent a vertical line such as  $x = 1$ ?

# Linear Algebra

- [Algebra](#) is the study of mathematical objects and their manipulating rules (e.g. arithmetic operations).
- [Linear algebra](#) is the mathematical branch on [linear equations](#) represented in [vector spaces](#).

- [Linear equation](#)

- e.g.  $ax + b = 0$  (one-variable),  $ax + by + c = 0$  (two-variable), ...,  $a_1x_1 + a_2x_2 + \dots + a_nx_n + b = 0$  ( $n$ -variable)
- cf. Nonlinear equations:  $y = 0.1x^3 - 0.8x^2 - 1.5x + 5.4$ ,  $y = \sin x$

- [Vector space](#) (a.k.a. linear space): A set of vectors which can be 1) added together and 2) multiplied by scalars

- [Scalar](#): A single element

- e.g.  $x$  (one-dimensional)

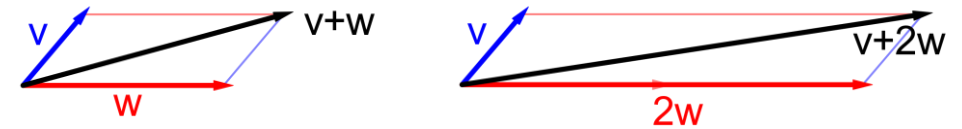
- [Vector](#): A finite ordered list of elements (a.k.a. [tuple](#))

- e.g.  $[x, y]$  (two-dimensional row vector), ...,  $[x_1, x_2, \dots, x_n]$  ( $n$ -dimensional row vector)

- [Matrix](#): A rectangular array (table) of elements arranged in **rows** and **columns** (a.k.a. [tensor](#))

- e.g.  $m \times n$  matrix

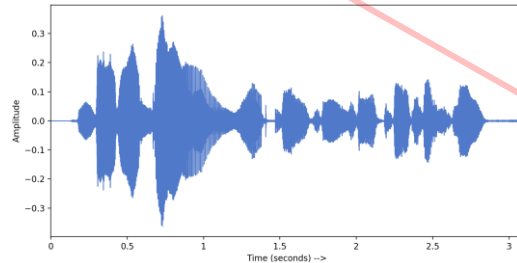
$$\begin{matrix} & \begin{matrix} 1 & 2 & \dots & n \end{matrix} \\ \begin{matrix} 1 \\ 2 \\ 3 \\ \vdots \\ m \end{matrix} & \begin{bmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ a_{21} & a_{22} & \dots & a_{2n} \\ a_{31} & a_{32} & \dots & a_{3n} \\ \vdots & \vdots & \vdots & \vdots \\ a_{m1} & a_{m2} & \dots & a_{mn} \end{bmatrix} \end{matrix}$$



# Vector

## ▪ Why a vector?

- Representing physical quantities
  - e.g. [Speed](#) (magnitude; **속력** in Korean) vs. [velocity](#) (magnitude+*direction*; **속도** in Korean)
- Representing data and models
  - e.g. A point  $[1, 4]$ , a line  $2x + 3y - 14 = 0 \rightarrow [2, 3, -14]$ , and its **normal vector**  $[2, 3]$  or  $[0.554 \dots, 8.832 \dots]$
  - e.g. Sound data



## ▪ Vector operations

- Vector addition (subtraction):  $\mathbf{a} + \mathbf{b}$
- Scalar multiplication (division):  $a \mathbf{b}$
- [Vector multiplication](#)
  - [Dot product](#) (a.k.a. scalar product):  $\mathbf{a} \cdot \mathbf{b} = \sum_{i=1}^n a_i b_i = a_1 b_1 + a_2 b_2 + \dots + a_n b_n$  or  $\mathbf{a} \cdot \mathbf{b} = \|\mathbf{a}\| \|\mathbf{b}\| \cos \theta$
  - [Cross product](#) (a.k.a. vector product):  $\mathbf{a} \times \mathbf{b} = \|\mathbf{a}\| \|\mathbf{b}\| \sin \theta \mathbf{n}$

# Vector

- **Vector operations** (cont'd)

- [Vector multiplication](#)

- [Dot product](#) (a.k.a. scalar product)

- Algebraic definition:  $\mathbf{a} \cdot \mathbf{b} = \sum_{i=1}^n a_i b_i = a_1 b_1 + a_2 b_2 + \dots + a_n b_n$

- As matrix multiplication:  $\mathbf{a} \cdot \mathbf{b} = \mathbf{a}^T \mathbf{b}$  where  $\mathbf{a}$  and  $\mathbf{b}$  are column vectors.

- $\mathbf{a} \cdot \mathbf{b} = \mathbf{a} \mathbf{b}^T$  where  $\mathbf{a}$  and  $\mathbf{b}$  are row vectors.

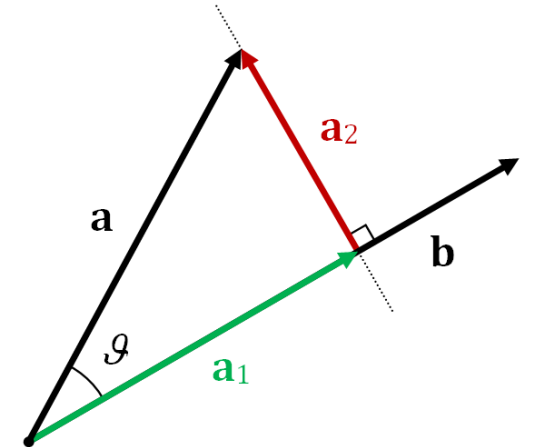
- e.g.  $[3, 2] \cdot [5, 1] = 17$

- e.g.  $[3, 2, 9] \cdot [5, 1, -2] = -1$

- Geometric definition:  $\mathbf{a} \cdot \mathbf{b} = \|\mathbf{a}\| \|\mathbf{b}\| \cos \theta$

- Meaning: [Scalar projection](#) → similarity of two vectors ([cosine similarity](#))

$$\cos \theta = \frac{\mathbf{a} \cdot \mathbf{b}}{\|\mathbf{a}\| \|\mathbf{b}\|}$$



# Vector

## ▪ Vector operations (cont'd)

### – [Vector multiplication](#)

- [Cross product](#) (a.k.a. vector product) is defined **only in a three-dimensional space**.

- Definition:  $\mathbf{a} \times \mathbf{b} = \|\mathbf{a}\| \|\mathbf{b}\| \sin \theta \hat{\mathbf{n}}$

where  $\hat{\mathbf{n}}$  is a unit vector **orthogonal** (수직 in Korean) to a plane containing  $\mathbf{a}$  and  $\mathbf{b}$

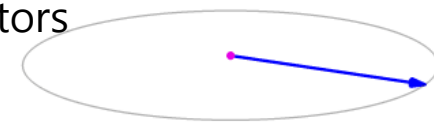
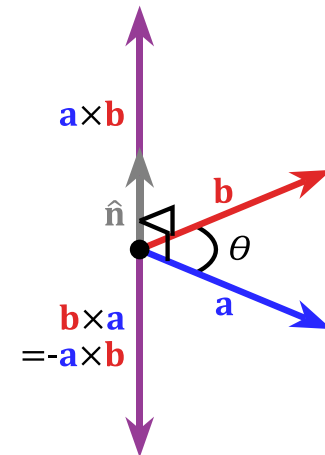
- Meaning

- Representing angular velocity, angular momentum, and torque
- Calculating the area of the [parallelogram](#) (평행사변형 in Korean) made by two vectors
- More applications in [computational geometry](#)

- Computing (using matrix multiplication)

$$\mathbf{a} \times \mathbf{b} = \begin{vmatrix} \mathbf{i} & \mathbf{j} & \mathbf{k} \\ a_1 & a_2 & a_3 \\ b_1 & b_2 & b_3 \end{vmatrix} = (a_2 b_3 - a_3 b_2) \mathbf{i} - (a_1 b_3 - a_3 b_1) \mathbf{j} + (a_1 b_2 - a_2 b_1) \mathbf{k}$$

- e.g.  $[1, 4, 1] \times [4, 2, 1] = [2, 3, -14]$





# Vector

## ▪ Vector operations (cont'd)

### – Vector multiplication

#### • Cross product

(a.k.a. vector product) is defined only in a three-dimensional space.

– Definition:  $\mathbf{a} \times \mathbf{b} = \|\mathbf{a}\| \|\mathbf{b}\| \sin \theta \mathbf{n}$

where  $\mathbf{n}$  is a unit vector orthogonal (수직 in Korean) to a plane containing  $\mathbf{a}$  and  $\mathbf{b}$

– Meaning

- Representing angular velocity, angular momentum, and torque
- Calculating the area of the parallelogram (평행사변형 in Korean) made by two vectors
- More applications in computational geometry

– Computing (using matrix multiplication)

$$\mathbf{a} \times \mathbf{b} = \begin{vmatrix} \mathbf{i} & \mathbf{j} & \mathbf{k} \\ a_1 & a_2 & a_3 \\ b_1 & b_2 & b_3 \end{vmatrix} = (a_2 b_3 - a_3 b_2) \mathbf{i} - (a_1 b_3 - a_3 b_1) \mathbf{j} + (a_1 b_2 - a_2 b_1) \mathbf{k}$$

– e.g.  $[1, 4, 1] \times [4, 2, 1] = [2, 3, -14]$

▪ Line representation:  $ax + by + c = 0$

$$(2x + 3y - 14 = 0)$$

(1, 4)

(4, 2)

- Area of the parallelogram made by two vectors
- Line from two points
- Intersection point from two lines
- Distance between two *skew* lines
- Volume of the parallelepiped made by three vectors

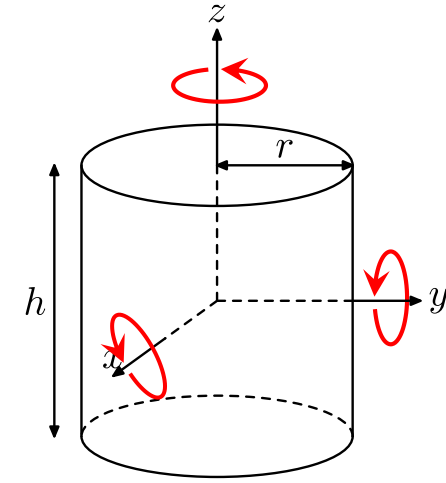
# Matrix

## ▪ Why a matrix?

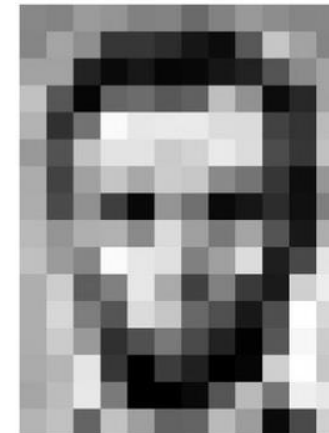
- Representing physical quantities
  - e.g. [Moment of inertia](#) in the 3D space (a.k.a. [inertia tensor](#))
- Representing data and models
  - e.g. Images, multiple vectors
  - e.g. [State-space representation](#) (control theory), [camera projection](#)
- Representing geometric transformation
  - e.g. [Rigid transformation](#), [camera projection](#)
- Solving a [system of linear equations](#) (선형연립방정식 in Korean)
  - e.g. A system of equations for a line from (1, 4) and (4, 2)

$$4 = w \cdot 1 + b$$

$$2 = w \cdot 4 + b$$



$$I = \begin{bmatrix} \frac{1}{12}m(3r^2 + h^2) & 0 & 0 \\ 0 & \frac{1}{12}m(3r^2 + h^2) & 0 \\ 0 & 0 & \frac{1}{2}mr^2 \end{bmatrix}$$



187	163	174	168	160	162	129	183	172	163	166	166
156	182	163	74	76	62	33	17	116	210	180	154
180	180	50	14	34	6	10	33	48	106	169	181
206	106	6	124	131	111	120	204	166	15	66	180
194	66	157	281	237	239	239	228	227	87	71	201
172	106	207	233	233	214	220	239	228	96	74	206
188	88	179	209	185	215	211	158	139	75	20	169
189	97	165	84	10	168	134	11	31	62	22	148
199	168	191	192	158	227	178	143	182	106	96	190
205	174	168	282	236	231	148	178	228	43	95	234
190	216	156	149	236	187	86	150	79	38	218	241
190	224	147	108	227	210	127	103	36	101	255	224
190	214	173	66	103	143	95	90	2	106	249	215
187	196	236	75	1	81	47	0	6	217	256	211
183	202	237	148	0	0	12	108	200	136	243	236
196	206	123	207	177	121	123	200	175	13	94	218

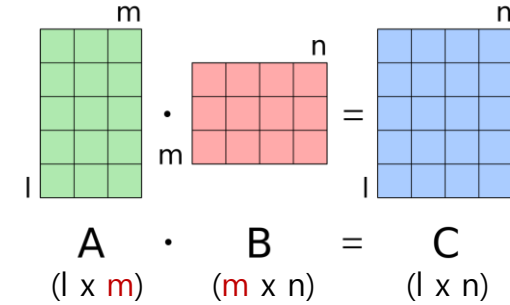
# Matrix

## ▪ Matrix operations

- Matrix addition (subtraction):  $A + B$
- Scalar multiplication (division):  $a B$
- **Matrix multiplication** (a.k.a. matrix product):  $AB$ 
  - **Not commutative** (교환법칙 in Korean):  $AB \neq BA$ 
    - Example) ~~Proof~~ Verification using [SymPy](#)  
`import sympy as sp`

```
a11, a12, a21, a22 = sp.symbols('a11 a12 a21 a22')
b11, b12, b21, b22 = sp.symbols('b11 b12 b21 b22')
A = sp.Matrix([[a11, a12], [a21, a22]])
B = sp.Matrix([[b11, b12], [b21, b22]])
```

```
print(A+B == B+A) # True
print(A*B == B*A) # False
print(A*B - B*A)  # Check their difference
```



# Matrix

- **Matrix operations** (cont'd)

- **Matrix inverse** ( $AA^{-1} = A^{-1}A = I$ ; ~ 역수 in Korean)
  - Invertible matrix (a.k.a. non-singular matrix, non-degenerate matrix) if A is
    - Square (A:  $n \times n$  matrix)
    - Linearly independent rows/columns
      - Full rank ( $\text{rank}(A) = n$ )
      - Non-zero determinant ( $\det(A) \neq 0$  or  $|A| \neq 0$ )
- Pseudo-inverse (~ a generalized matrix inverse)
  - **Not necessarily square** (A:  $m \times n$  matrix,  $A^+$ :  $n \times m$  matrix)
  - Left inverse ( $A^+A = I_n$ ):  $A^+ = (A^T A)^{-1} A^T$ 
    - If A has linearly independent columns ( $\text{rank}(A) = n$ )
  - Right inverse ( $AA^+ = I_m$ ):  $A^+ = A^T (AA^T)^{-1}$ 
    - If A has linearly independent rows ( $\text{rank}(A) = m$ )

# Matrix

- **Matrix operations** (cont'd)

- **Matrix inverse**

- Example) Line fitting from two points, (1, 4) and (4, 2)

- Line representation:  $y = wx + b$

- A system of equations:  $w \cdot 1 + b = 4$

$$w \cdot 4 + b = 2$$



$$\begin{bmatrix} 1 & 1 \\ 4 & 1 \end{bmatrix} \begin{bmatrix} w \\ b \end{bmatrix} = \begin{bmatrix} 4 \\ 2 \end{bmatrix}$$

$$(A\mathbf{x} = \mathbf{b})$$

$$\longrightarrow \mathbf{x} = A^{-1}\mathbf{b}$$

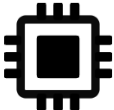













- Pseudo-inverse

- Example) Line fitting from more than two points such as (1, 4), (4, 2), and (7, 1)

- $A = \begin{bmatrix} 1 & 1 \\ 4 & 1 \\ 7 & 1 \end{bmatrix}$  and  $\mathbf{b} = \begin{bmatrix} 4 \\ 2 \\ 1 \end{bmatrix}$

# NumPy: A Matrix Library in Python

- [NumPy](#) supports multi-dimensional arrays and matrices with their high-level mathematical functions.
  - It is fundamental for scientific and numerical computing in Python.
  - References: [Documentation](#), [Tutorials/Books/Talks](#), and [Cheatsheet](#) (made by DataCamp)

<b>Quantum Computing</b>  <a href="#">QuTiP</a> <a href="#">PyQuil</a> <a href="#">Qiskit</a>	<b>Statistical Computing</b>  <a href="#">Pandas</a> <a href="#">statsmodels</a> <a href="#">Xarray</a> <a href="#">Seaborn</a>	<b>Signal Processing</b>  <a href="#">SciPy</a> <a href="#">PyWavelets</a> <a href="#">python-control</a>	<b>Image Processing</b>  <a href="#">Scikit-image</a> <a href="#">OpenCV</a> <a href="#">Mahotas</a>	<b>Graphs and Networks</b>  <a href="#">NetworkX</a> <a href="#">graph-tool</a> <a href="#">igraph</a> <a href="#">PyGSP</a>	<b>Astronomy Processes</b>  <a href="#">AstroPy</a> <a href="#">SunPy</a> <a href="#">SpacePy</a>	<b>Cognitive Psychology</b>  <a href="#">PsychoPy</a>
<b>Bioinformatics</b>  <a href="#">BioPython</a> <a href="#">Scikit-Bio</a> <a href="#">PyEnsembl</a> <a href="#">ETE</a>	<b>Bayesian Inference</b>  <a href="#">PyStan</a> <a href="#">PyMC3</a> <a href="#">ArviZ</a> <a href="#">emcee</a>	<b>Mathematical Analysis</b>  <a href="#">SciPy</a> <a href="#">SymPy</a> <a href="#">cvxpy</a> <a href="#">FEniCS</a>	<b>Chemistry</b>  <a href="#">Cantera</a> <a href="#">MDAnalysis</a> <a href="#">RDKit</a>	<b>Geoscience</b>  <a href="#">Pangeo</a> <a href="#">Simpeg</a> <a href="#">ObsPy</a> <a href="#">Fatiando a Terra</a>	<b>Geographic Processing</b>  <a href="#">Shapely</a> <a href="#">GeoPandas</a> <a href="#">Folium</a>	<b>Architecture &amp; Engineering</b>  <a href="#">COMPAS</a> <a href="#">City Energy Analyst</a> <a href="#">Sverchok</a>

# NumPy:

- NumPy
  - It is
  - Refere

## Python For Data Science Cheat Sheet

### NumPy Basics

Learn Python for Data Science Interactively at [www.DataCamp.com](https://www.datacamp.com)



#### NumPy

The NumPy library is the core library for scientific computing in Python. It provides a high-performance multidimensional array object, and tools for working with these arrays.

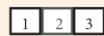
Use the following import convention:

```
>>> import numpy as np
```

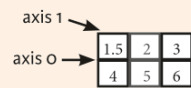


#### NumPy Arrays

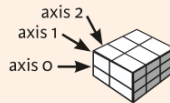
##### 1D array



##### 2D array



##### 3D array



#### Creating Arrays

```
>>> a = np.array([1,2,3])
>>> b = np.array([(1.5,2,3), (4,5,6)], dtype = float)
>>> c = np.array([(1.5,2,3), (4,5,6)], [(3,2,1), (4,5,6)]),
dtype = float)
```

#### Initial Placeholders

```
>>> np.zeros((3,4))
>>> np.ones((2,3,4),dtype=np.int16)
>>> d = np.arange(10,25,5)

>>> np.linspace(0,2,9)

>>> e = np.full((2,2),7)
>>> f = np.eye(2)
>>> np.random.random((2,2))
>>> np.empty((3,2))
```

Create an array of zeros  
Create an array of ones  
Create an array of evenly spaced values (step value)  
Create an array of evenly spaced values (number of samples)  
Create a constant array  
Create a 2X2 identity matrix  
Create an array with random values  
Create an empty array

#### I/O

##### Saving & Loading On Disk

```
>>> np.save('my_array', a)
>>> np.savez('array.npz', a, b)
>>> np.load('my_array.npy')
```

##### Saving & Loading Text Files

```
>>> np.loadtxt("myfile.txt")
>>> np.genfromtxt("my_file.csv", delimiter=',')
>>> np.savetxt("myarray.txt", a, delimiter=" ")
```

#### Data Types

```
>>> np.int64
>>> np.float32
>>> np.complex
>>> np.bool
>>> np.object
>>> np.string_
>>> np.unicode_
```

Signed 64-bit integer types  
Standard double-precision floating point  
Complex numbers represented by 128 floats  
Boolean type storing TRUE and FALSE values  
Python object type  
Fixed-length string type  
Fixed-length unicode type

#### Inspecting Your Array

```
>>> a.shape
>>> len(a)
>>> b.ndim
>>> e.size
>>> b.dtype
>>> b.dtype.name
>>> b.astype(int)
```

Array dimensions  
Length of array  
Number of array dimensions  
Number of array elements  
Data type of array elements  
Name of data type  
Convert an array to a different type

#### Asking For Help

```
>>> np.info(np.ndarray.dtype)
```

#### Array Mathematics

##### Arithmetic Operations

```
>>> g = a - b
array([[ -0.5,  0. ,  0. ],
       [-3. , -3. , -3. ]])
>>> np.subtract(a,b)
>>> b + a
array([[ 2.5,  4. ,  6. ],
       [ 5. ,  7. ,  9. ]])
>>> np.add(b,a)
>>> a / b
array([[ 0.66666667,  1. ,  1. ],
       [ 0.25 ,  0.4 ,  0.5 ]])
>>> np.divide(a,b)
>>> a * b
array([[ 1.5,  4. ,  9. ],
       [ 4. , 10. , 18. ]])
>>> np.multiply(a,b)
>>> np.exp(b)
>>> np.sqrt(b)
>>> np.sin(a)
>>> np.cos(b)
>>> np.log(a)
>>> e.dot(f)
array([[ 7. ,  7. ],
       [ 7. ,  7.]])
```

Subtraction  
Subtraction  
Addition  
Addition  
Division  
Division  
Multiplication  
Multiplication  
Exponentiation  
Square root  
Print sines of an array  
Element-wise cosine  
Element-wise natural logarithm  
Dot product

##### Comparison

```
>>> a == b
array([[False,  True,  True],
       [False, False, False]], dtype=bool)
>>> a < 2
array([[True, False, False], dtype=bool)
>>> np.array_equal(a, b)
```

Element-wise comparison  
Element-wise comparison  
Array-wise comparison

##### Aggregate Functions

```
>>> a.sum()
>>> a.min()
>>> b.max(axis=0)
>>> b.cumsum(axis=1)
>>> a.mean()
>>> b.median()
>>> a.corrcoef()
>>> np.std(b)
```

Array-wise sum  
Array-wise minimum value  
Maximum value of an array row  
Cumulative sum of the elements  
Mean  
Median  
Correlation coefficient  
Standard deviation

#### Copying Arrays

```
>>> h = a.view()
>>> np.copy(a)
>>> h = a.copy()
```

Create a view of the array with the same data  
Create a copy of the array  
Create a deep copy of the array

#### Sorting Arrays

```
>>> a.sort()
>>> c.sort(axis=0)
```

Sort an array  
Sort the elements of an array's axis

#### Subsetting, Slicing, Indexing

Also see Lists

##### Subsetting

```
>>> a[2]
3
```



Select the element at the 2nd index

```
>>> b[1,2]
6.0
```



Select the element at row 1 column 2 (equivalent to b[1][2])

##### Slicing

```
>>> a[0:2]
array([1, 2])
```



Select items at index 0 and 1

```
>>> b[0:2,1]
array([ 2.,  5.])
```



Select items at rows 0 and 1 in column 1

```
>>> b[:1]
array([[1.5, 2., 3.]])
```



Select all items at row 0 (equivalent to b[0:1, :])

```
>>> c[1,...]
array([[ 3.,  2.,  1.],
       [ 4.,  5.,  6.]])
```



Same as [1, :, :]

```
>>> a[: :-1]
array([3, 2, 1])
```

Reversed array a

##### Boolean Indexing

```
>>> a[a<2]
array([1])
```



Select elements from a less than 2

##### Fancy Indexing

```
>>> b[[1, 0, 1, 0], [0, 1, 2, 0]]
array([ 4. ,  2. ,  6. ,  1.5])
```

Select elements (1,0), (0,1), (1,2) and (0,0)

```
>>> b[[1, 0, 1, 0]][:,[0,1,2,0]]
array([[ 4.,  5.,  6.,  4. ],
       [ 1.5,  2.,  3.,  1.5]])
```

Select a subset of the matrix's rows and columns

#### Array Manipulation

##### Transposing Array

```
>>> i = np.transpose(b)
>>> i.T
```

Permute array dimensions  
Permute array dimensions

##### Changing Array Shape

```
>>> b.ravel()
>>> g.reshape(3,-2)
```

Flatten the array  
Reshape, but don't change data

##### Adding/Removing Elements

```
>>> h.resize((2,6))
>>> np.append(h,g)
>>> np.insert(a, 1, 5)
>>> np.delete(a, [1])
```

Return a new array with shape (2,6)  
Append items to an array  
Insert items in an array  
Delete items from an array

##### Combining Arrays

```
>>> np.concatenate((a,d),axis=0)
array([ 1,  2,  3, 10, 15, 20])
```

Concatenate arrays

```
>>> np.vstack((a,b))
array([[ 1. ,  2. ,  3. ],
       [ 1.5,  2. ,  3. ],
       [ 4. ,  5. ,  6. ]])
```

Stack arrays vertically (row-wise)

```
>>> np.r_[e,f]
```

Stack arrays vertically (row-wise)

```
>>> np.hstack((e,f))
array([[ 7. ,  7. ,  1. ,  0. ],
       [ 7. ,  7. ,  0. ,  1.]])
```

Stack arrays horizontally (column-wise)

```
>>> np.column_stack((a,d))
array([[ 1, 10],
       [ 2, 15],
       [ 3, 20]])
```

Create stacked column-wise arrays

```
>>> np.c_[a,d]
```

Create stacked column-wise arrays

##### Splitting Arrays

```
>>> np.hsplit(a,3)
[array([1]),array([2]),array([3])]
>>> np.vsplit(c,2)
[array([[ 1.5,  2. ,  1. ],
       [ 4. ,  5. ,  6. ]]),
array([[ 3. ,  2. ,  3. ],
       [ 4. ,  5. ,  6. ]])]
```

Split the array horizontally at the 3rd index  
Split the array vertically at the 2nd index



# NumPy: A Matrix Library in Python

- Usage example) Creating n-dimensional arrays (1/2)
  - `numpy.array` can contain a **homogenous** (~ same) data type. (vs. list and tuple)

```
import numpy as np
```

```
# 1. Create an array from a composite data (list or tuple, not set and dictionary)
```

```
A = np.array([3, 29, 82])
```

```
B = np.array(((3., 29, 82), (10, 18, 84)))
```

```
C = np.array([[3, 29, 82], [10, 18, 84]], dtype=float)
```

```
D = np.array([3, 29, 'Choi'])
```

```
E = np.array([3], [29], [82])
```

```
print(A.ndim, A.size, A.shape, A.dtype) # 1 3 (3,) int32
```

```
# cf. np.array_equal(A, A.T) == True
```

```
print(B.ndim, B.size, B.shape, B.dtype) # 2 6 (2, 3) float64
```

```
print(C.ndim, C.size, C.shape, C.dtype) # 2 6 (2, 3) float64
```

```
print(D.ndim, D.size, D.shape, D.dtype) # 1 3 (3,) <U11 (Unicode)
```

```
# cf. array(['3', '29', 'Choi'])
```

```
print(E.ndim, E.size, E.shape, E.dtype) # 2 3 (3,1) int32
```

```
# cf. np.array_equal(E, E.T) == False
```

```
# because E.T == array([[3, 29, 82]])
```



# NumPy: A Matrix Library in Python

- Usage example) Creating n-dimensional arrays (2/2)
  - `numpy.array` can contain a **homogenous** (~ same) data type. (vs. list and tuple)

```
import numpy as np
```

```
# 2. Create an array using initializers
```

```
F = np.zeros((3, 2))          # Create a 3x2 array filled with 0 (default: float64)
G = np.ones((3, 2))           # Create a 2x3 array filled with 1
H = np.eye(3, dtype=np.float32) # Create a 3x3 identity matrix (single-precision)
I = np.empty((3, 2))          # == np.zeros((3, 2))
J = np.empty((0, 9))           # [ ] with size of (0, 9)
K = np.arange(0, 1, 0.2)       # Step 0.2: array([0., 0.2, 0.4, 0.6, 0.8])
L = np.linspace(0, 1, 5)       # Number 5: array([0., 0.25, 0.50, 0.75, 1.])
M = np.random.random((3, 2))   # == np.random.uniform(size=(3, 2)) cf. normal()
```

# NumPy: A Matrix Library in Python

- Usage example) Indexing and slicing
  - `numpy.array` can access elements of a n-dim array not only with `[r][c]` but also with `[r,c]`.
  - `numpy.array` can access elements with a Boolean array (called **logical indexing**).

```
import numpy as np
A = np.array(((3., 29, 82), (10, 18, 84)))
```

	0	1	2
0	3	29	82
1	10	18	84

```
# 1. Indexing and slicing
```

```
A[1][1]                # 18.0
A[1, 1]                # 18.0 cf. list[1, 1] does not work!
A[1,1:2]               # array([18.])
A[1,:]                # Get a row:      array([10., 18., 84.])
A[:,2]                # Get a column:   array([82., 84.])
A[0:2,0:2]            # Get a submatrix: array([[3., 29.], [10., 18.]])
```

```
# 2. Logical indexing
```

```
A > 80                 # array([[False, False, True], [False, False, True]])
A[A > 80]              # array([82., 84.])
A[A > 80] = 80         # Masked operations are possible!
```

```
# 3. Fancy indexing
```

```
A[(1, 0, 0), (1, 0, 2)] # array([18., 3., 82.])
                        # Get items at (1, 1), (0, 0), (0, 2)
```

# NumPy: A Matrix Library in Python

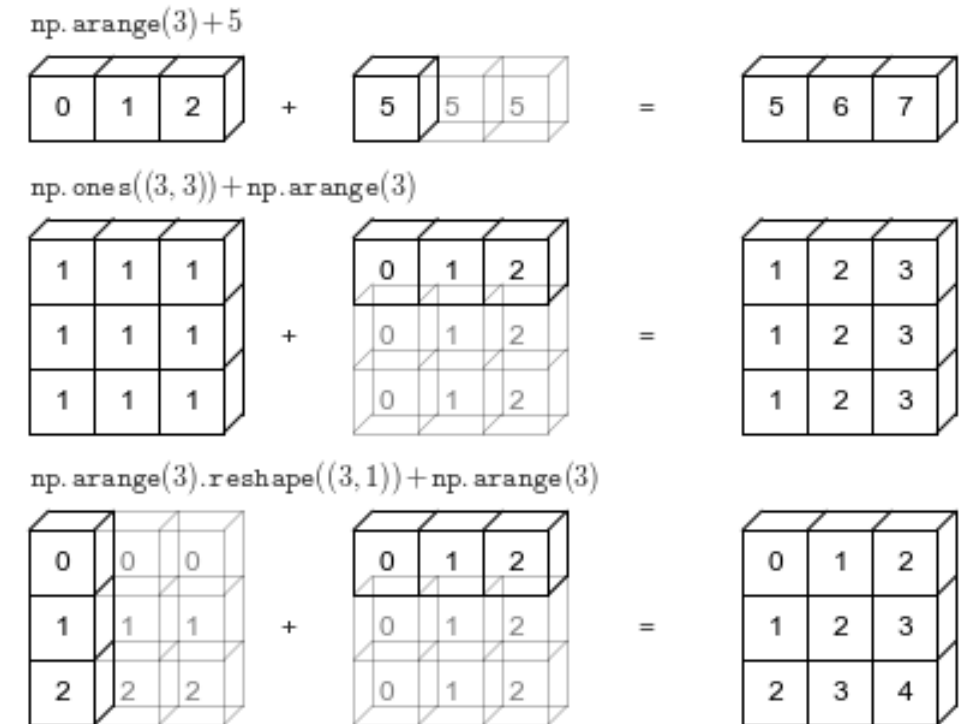
- Usage example) Arithmetic operations (vs. list and tuple)

```
import numpy as np
A = np.array([[1, 2], [3, 4]])
B = np.array([[5, 6], [7, 8]])

# 1. Element-wise arithmetic operations
print(A + B)           # np.add(A, B)
print(A - B)           # np.subtract(A, B)
print(A * B)           # np.multiply(A, B)
print(A / B)           # np.divide(A, B)

# 2. Matrix operations
print(np.matmul(A, B)) # Matrix multiplication!
print(A.T)             # A.transpose()
print(np.linalg.matrix_rank(A)) # 2, full rank
print(np.linalg.det(A)) # -2, non-zero determinant
print(np.linalg.inv(A)) # Matrix inverse
print(np.linalg.pinv(A)) # Matrix pseudo-inverse

# 3. Broadcasting
print(A + 1)           # [[2, 3], [4, 5]]
print(A + [0, -1])     # [[1, 1], [3, 3]]
print(A + [[1], [-1]]) # [[2, 3], [2, 3]]
```



# Matrix

- Example) Line fitting from two points, (1, 4) and (4, 2)

- Line representation:  $y = wx + b$
- A system of equations:  $w \cdot 1 + b = 4$

$$w \cdot 4 + b = 2$$

$$\begin{array}{c} \downarrow \\ \begin{bmatrix} 1 & 1 \\ 4 & 1 \end{bmatrix} \begin{bmatrix} w \\ b \end{bmatrix} = \begin{bmatrix} 4 \\ 2 \end{bmatrix} \\ (A\mathbf{x} = \mathbf{b}) \end{array} \longrightarrow \mathbf{x} = A^{-1}\mathbf{b}$$

- Example) Line fitting from two points, (1, 4) and (4, 2), using [NumPy](#)

```
import numpy as np
```

```
A = np.array([[1., 1.], [4., 1.]])
```

```
b = np.array([[4.], [2.]])
```

```
A_inv = np.linalg.inv(A)
```

```
print(A_inv * b)           # [[-1.33333333  1.33333333]
                           # [ 2.66666667 -0.66666667]] cf. broadcast
print(np.matmul(A_inv, b)) # [[-0.66666667]
                           # [ 4.66666667]]
```

# Matrix

- Example) Line fitting from two points, (1,4) and (4,2), using [NumPy](#)

```
import numpy as np
```

```
A = np.array([[1., 1.], [4., 1.]])
```

```
b = np.array([4., 2.])
```

```
A_inv = np.linalg.inv(A)
```

```
print(np.matmul(A_inv, b)) # [[-0.66666667]
                          # [ 4.66666667]]
```

- Example) Line fitting from more than two points such as (1,4), (4,2), and (7,1) using [NumPy](#)

–  $A = \begin{bmatrix} 1 & 1 \\ 4 & 1 \\ 7 & 1 \end{bmatrix}$  and  $\mathbf{b} = \begin{bmatrix} 4 \\ 2 \\ 1 \end{bmatrix}$

```
import numpy as np
```

```
A = np.array([[1., 1.], [4., 1.], [7., 1.]])
```

```
b = np.array([4., 2., 1.])
```

```
A_inv = np.linalg.pinv(A)
```

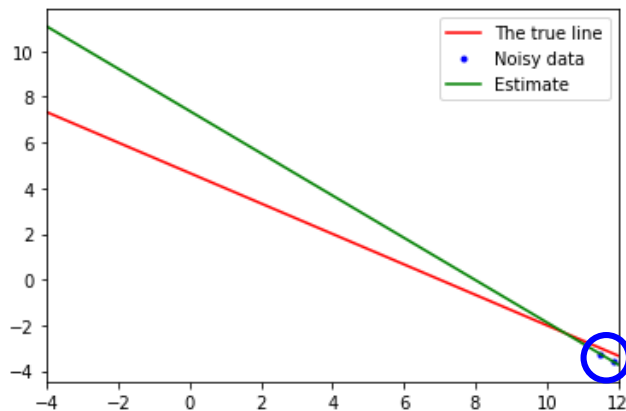
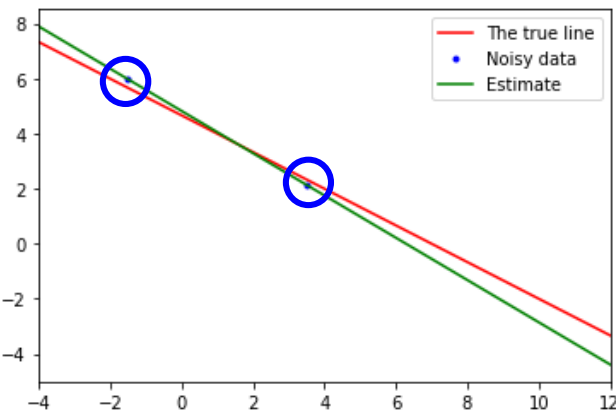
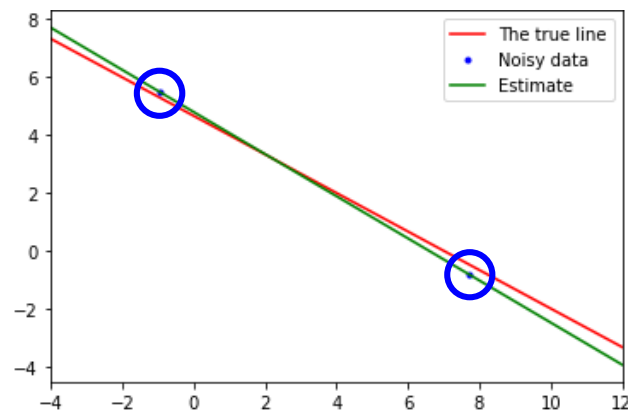
```
print(np.matmul(A_inv, b)) # [[-0.5]
                          # [ 4.33333333]]
```

● (7,1)

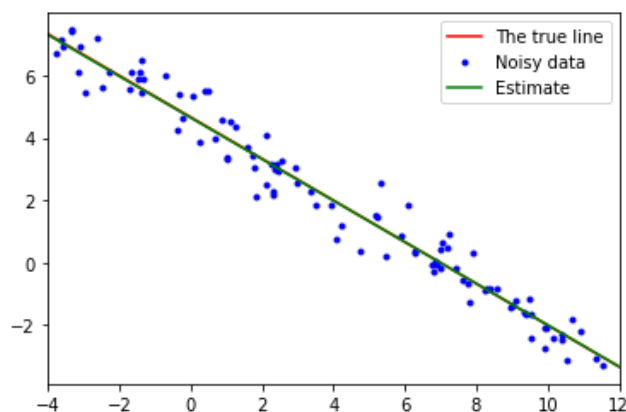
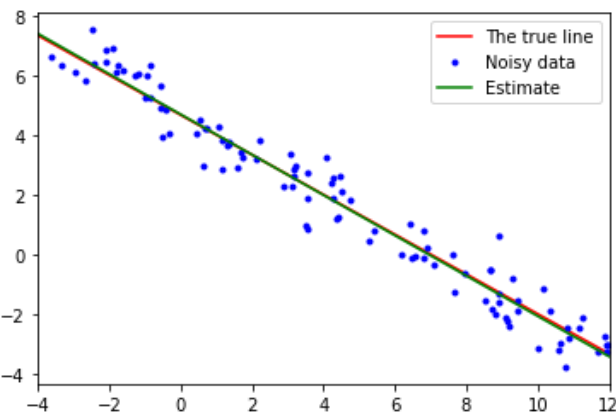
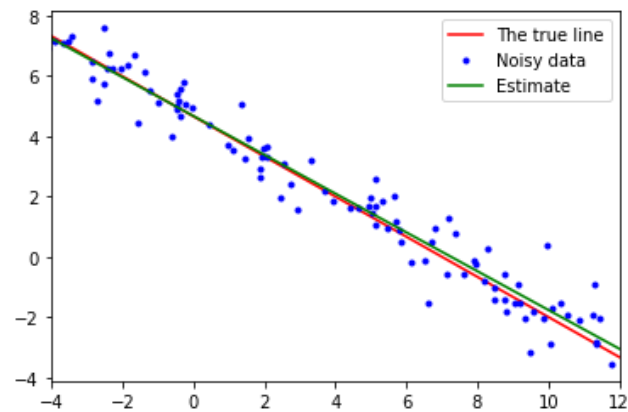
# Matrix

- Example) Line fitting with noisy but more data (1/2)

- data\_num = 2



- data\_num = 100



# Matrix

- Example) Line fitting with noisy but more data (2/2)

```
import numpy as np
import matplotlib.pyplot as plt

true_line = lambda x: -2/3*x + 14/3
data_range = np.array([-4, 12])
data_num = 100
noise_std = 0.5

# Generate the true data
x = np.random.uniform(data_range[0], data_range[1], size=data_num)
y = true_line(x) # y = -2/3*x + 10/3

# Add Gaussian noise
xn = x + np.random.normal(scale=noise_std, size=x.shape)
yn = y + np.random.normal(scale=noise_std, size=y.shape)

# Solve the system of equations
A = np.vstack((xn, np.ones(xn.shape))).T
b = yn
line = np.matmul(np.linalg.pinv(A), b)

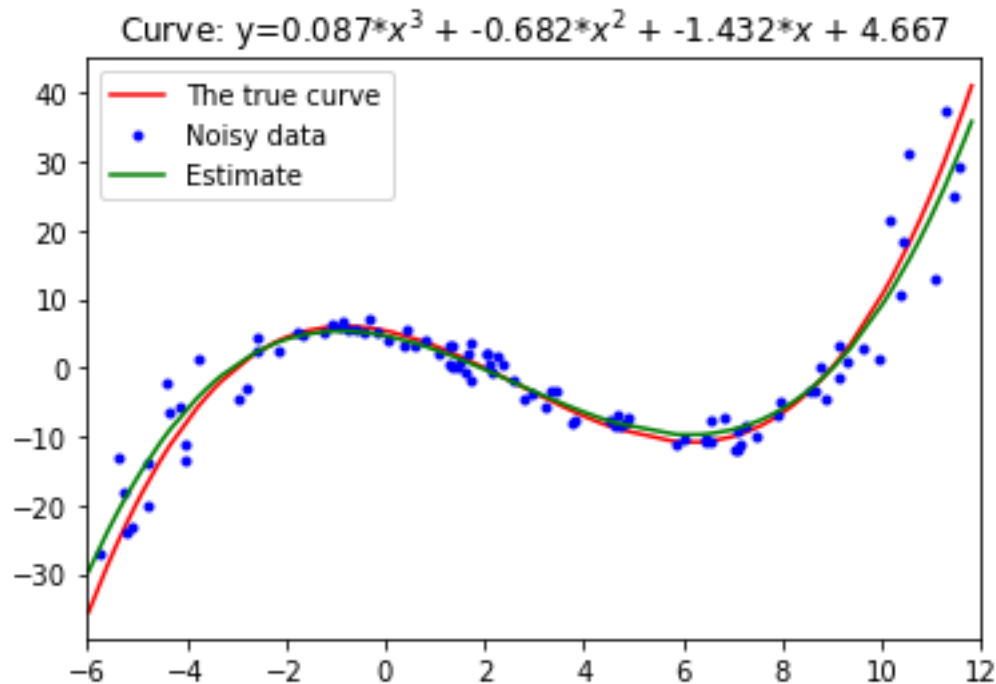
# Plot the data and result
plt.title(f'Line: y={line[0]:.3f}*x + {line[1]:.3f} ')
plt.plot(data_range, true_line(data_range), 'r-', label='The true line')
plt.plot(xn, yn, 'b.', label='Noisy data')
plt.plot(data_range, line[0]*data_range + line[1], 'g-', label='Estimate')
plt.xlim(data_range)
plt.legend()
plt.show()
```

$$\begin{bmatrix} x_1 & x_2 & \cdots & x_n \\ 1 & 1 & \cdots & 1 \end{bmatrix} \xrightarrow{\text{transpose}} \begin{bmatrix} x_1 & 1 \\ x_2 & 1 \\ \vdots & \vdots \\ x_n & 1 \end{bmatrix} \begin{bmatrix} w \\ b \end{bmatrix} = \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{bmatrix}$$

(Ax = b)

# Matrix

- Example) Curve fitting (1/2)
  - Curve representation:  $y = ax^3 + bx^2 + cx + d$  (a 3rd-order polynomial equation)
    - e.g.  $y = 0.1x^3 - 0.8x^2 - 1.5x + 5.4$



$$\begin{bmatrix} x_1^3 & x_1^2 & x_1 & 1 \\ x_2^3 & x_2^2 & x_2 & 1 \\ \vdots & \vdots & \vdots & \vdots \\ x_n^3 & x_n^2 & x_n & 1 \end{bmatrix} \begin{bmatrix} a \\ b \\ c \\ d \end{bmatrix} = \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{bmatrix}$$

$(A\mathbf{x} = \mathbf{b})$



# Matrix

- Example) Curve fitting (2/2)

```
import numpy as np
import matplotlib.pyplot as plt

true_curve = lambda x: 0.1*x**3 - 0.8*x**2 - 1.5*x + 5.4
data_range = (-6, 12)
data_num = 100
noise_std = 0.5

# Generate the true data
x = np.random.uniform(data_range[0], data_range[1], size=data_num)
y = true_curve(x)

# Add Gaussian noise
xn = x + np.random.normal(scale=noise_std, size=x.shape)
yn = y + np.random.normal(scale=noise_std, size=y.shape)

# Solve the system of equations
A = np.vstack((xn**3, xn**2, xn, np.ones(xn.shape))).T
b = yn
curve = np.matmul(np.linalg.pinv(A), b)

# Plot the data and result
plt.title(f'Curve: y={curve[0]:.3f}*$x^3$ + {curve[1]:.3f}*$x^2$ + {curve[2]:.3f}*$x$ + {curve[3]:.3f}')
xc = np.linspace(*data_range, 100)
plt.plot(xc, true_curve(xc), 'r-', label='The true curve')
plt.plot(xn, yn, 'b.', label='Noisy data')
plt.plot(xc, curve[0]*xc**3 + curve[1]*xc**2 + curve[2]*xc + curve[3], 'g-', label='Estimate')
plt.xlim(data_range)
plt.legend()
plt.show()
```

$$\begin{bmatrix} x_1^3 & x_1^2 & x_1 & 1 \\ x_2^3 & x_2^2 & x_2 & 1 \\ \vdots & \vdots & \vdots & \vdots \\ x_n^3 & x_n^2 & x_n & 1 \end{bmatrix} \begin{bmatrix} a \\ b \\ c \\ d \end{bmatrix} = \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{bmatrix}$$

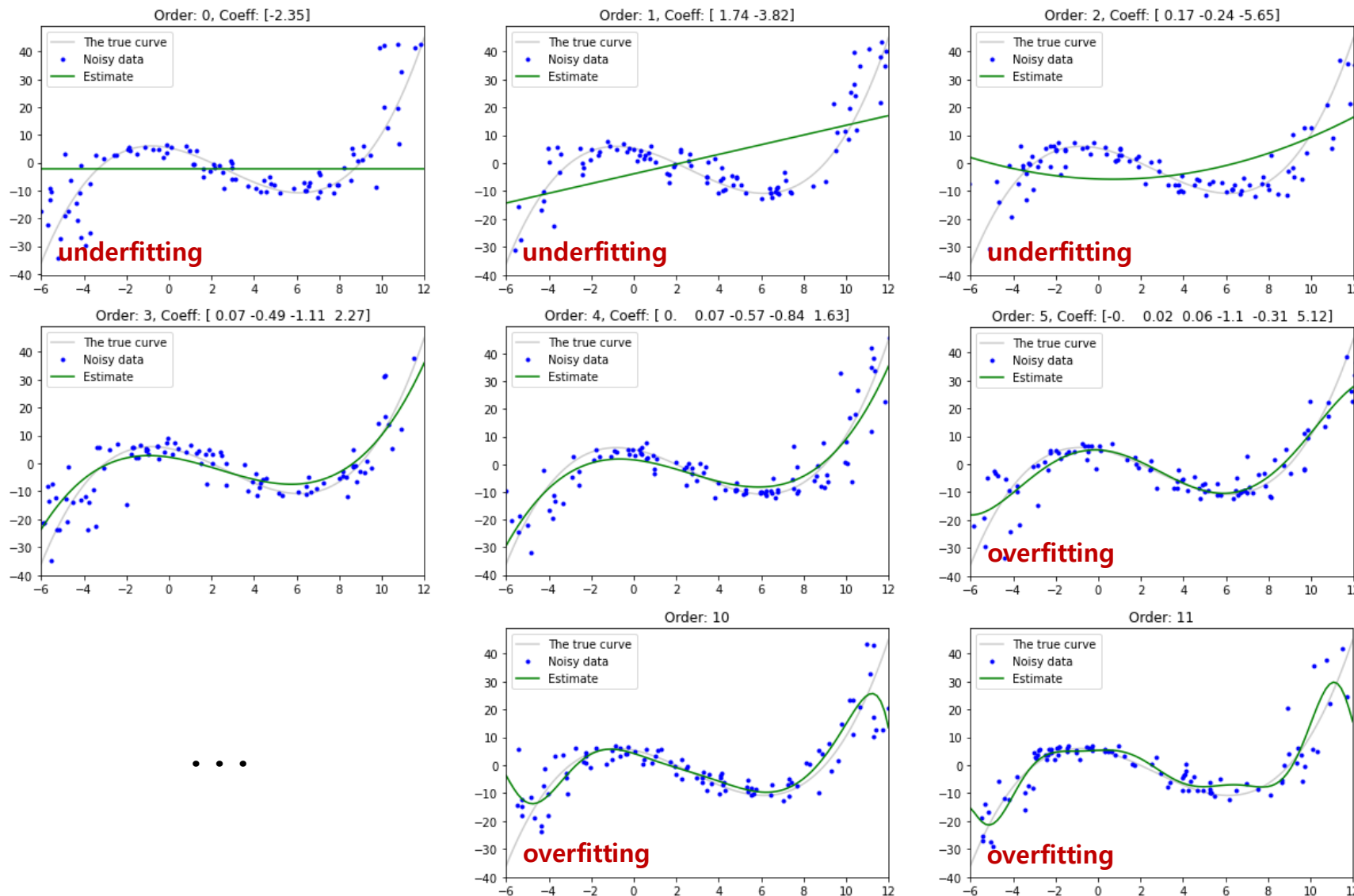
$(A\mathbf{x} = \mathbf{b})$

# Matrix

- Example) Curve fitting with model selection (1/2)

$$\begin{bmatrix} x_1^r & x_1^{r-1} & \cdots & x_1^2 & x_1 & 1 \\ x_2^r & x_2^{r-1} & \cdots & x_2^2 & x_2 & 1 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ x_n^r & x_n^{r-1} & \cdots & x_n^2 & x_n & 1 \end{bmatrix} \begin{bmatrix} c_r \\ c_{r-1} \\ \vdots \\ c_2 \\ c_1 \\ c_0 \end{bmatrix} = \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{bmatrix}$$

( $Ax = b$ )



```
import numpy as np
import matplotlib.pyplot as plt
```

```
def buildA(order, xs):
    A = np.empty((0, len(xs)))
    for i in range(order + 1):
        A = np.vstack((xs**i, A))
    return A.T
```

```
true_coeff = [0.1, -0.8, -1.5, 5.4]
poly_order = 3 # Try other integer (>= 0)
data_range = (-6, 12)
data_num = 100
noise_std = 1
```

```
# Generate the true data
x = np.random.uniform(data_range[0], data_range[1], size=data_num)
y = np.matmul(buildA(len(true_coeff) - 1, x), true_coeff)
```

```
# Add Gaussian noise
xn = x + np.random.normal(scale=noise_std, size=x.shape)
yn = y + np.random.normal(scale=noise_std, size=y.shape)
```

```
# Solve the system of equations
A = buildA(poly_order, xn)
b = yn
coeff = np.matmul(np.linalg.pinv(A), b)
```

```
# Plot the data and result
plt.title(f'Order: {poly_order}, Coeff: ' + np.array2string(coeff, precision=2, suppress_small=True))
xc = np.linspace(*data_range, 100)
plt.plot(xc, np.matmul(buildA(len(true_coeff) - 1, xc), true_coeff), 'k-', label='The true curve', alpha=0.2)
plt.plot(xn, yn, 'b.', label='Noisy data')
plt.plot(xc, np.matmul(buildA(poly_order, xc), coeff), 'g-', label='Estimate')
plt.xlim(data_range)
plt.legend()
plt.show()
```

$$\begin{bmatrix} x_1^r & x_1^{r-1} & \cdots & x_1^2 & x_1 & 1 \\ x_2^r & x_2^{r-1} & \cdots & x_2^2 & x_2 & 1 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ x_n^r & x_n^{r-1} & \cdots & x_n^2 & x_n & 1 \end{bmatrix} \begin{bmatrix} c_r \\ c_{r-1} \\ \vdots \\ c_2 \\ c_1 \\ c_0 \end{bmatrix} = \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{bmatrix}$$

(Ax=b)

# Matrix

- Example) Line fitting from two points, (1, 4) and (1, 2)

- Line representation:  $y = wx + b$
- A system of equations:  $w \cdot 1 + b = 4$

$$w \cdot 1 + b = 2$$



$$\begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix} \begin{bmatrix} w \\ b \end{bmatrix} = \begin{bmatrix} 4 \\ 2 \end{bmatrix}$$

$$(A\mathbf{x} = \mathbf{b})$$

$$\longrightarrow \mathbf{x} = A^{-1}\mathbf{b}$$

- Example) Line fitting from two points, (1, 4) and (1, 2), using [NumPy](#)

```
import numpy as np
```

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

```
b = np.array([[4.], [2.]])
```

```
A_inv = np.linalg.inv(A) # Error! (singular matrix)
```

```
print(np.matmul(A_inv, b))
```

# Matrix

- **Null space** (a.k.a. [kernel](#))
  - A set of vectors which map  $A$  ( $m$ -by- $n$  matrix) to the zero vector
$$N(A) = \{ \mathbf{v} \in K^n \mid A\mathbf{v} = \mathbf{0} \}$$
  - [Rank-nullity theorem](#):  $\text{rank}(A) + \text{nullity}(A) = n$
  - Application) Solving a **homogenous** system of linear equations,  $A\mathbf{x} = \mathbf{0}$
- Practice) Line fitting from two points,  $(1, 4)$  and  $(1, 2)$ , using [NumPy](#)
  - Line representation:  $ax + by + c = 0$
  - In a matrix form,  $\begin{bmatrix} 1 & 4 & 1 \\ 1 & 2 & 1 \end{bmatrix} \begin{bmatrix} a \\ b \\ c \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$

```
import numpy as np

A = np.array([[1., 4., 1.], [1., 2., 1.]])
x = scipy.linalg.null_space(A)
print(x / x[0]) # [[1., 0., -1.]], line: x - 1 = 0
```

How about  $(1, 4)$  and  $(4, 2)$ ?

# Summary

- [NumPy](#): N-dimensional array representation and [numerical analysis](#) (수치해석 in Korean)
  - `numpy.array` vs. `list/tuple`
    - Homogeneous data type
    - + Indexing/slicing and arithmetic operations
- ~~Linear algebra~~ [Vector](#)
  - Why? 1) Representing physical quantities / 2) Representing data and models
  - [Vector multiplication](#)
    - Dot product:  $\mathbf{a} \cdot \mathbf{b} = \|\mathbf{a}\| \|\mathbf{b}\| \cos \theta$ 
      - Application: [Cosine similarity](#)
    - Cross product:  $\mathbf{a} \times \mathbf{b} = \|\mathbf{a}\| \|\mathbf{b}\| \sin \theta \mathbf{n}$  ( $\mathbf{n}$ : the orthogonal vector of  $\mathbf{a}$  and  $\mathbf{b}$ ) defined in 3D spaces
      - Application: Physics in 3D, ..., [computational geometry](#)

# Summary

## ~~Linear algebra~~ Matrix

- Why? + 3) Representing geometric transformation / 4) **Solving a system of linear equations**
- Matrix operations
  - Matrix multiplication (not commutative)
  - Matrix inverse (square + full rank), pseudo-inverse
- Examples) **Line and curve fitting** (regression analysis; 회귀분석 in Korean)
  - What is a model and its parameters?
    - e.g. Line:  $y = wx + b$ ,  $ax + by + c = 0$
    - e.g. Curve:  $n$ th-order polynomial equations
      - Model selection: underfitting and overfitting
  - How to formulate the problem using matrices:  $A\mathbf{x} = \mathbf{b}$  (or  $A\mathbf{x} = \mathbf{0}$ )
  - How to solve the equations using matrices: Matrix (pseudo-)inverse (or finding null vectors)
  - Benefit from more (over-constrained) data
    - The estimation result becomes *robust* ( $\sim$  strong) to noise!