

# UECM1703 Introduction to Scientific Computing

## Topic 2: Arrays Manipulation

*Lecturer: Dr Liew How Hui (liewhh@utar.edu.my)*  
*October 2024*

Programming with Topic 1's basic data types and Topic 3's imperative programming is not convenient and efficient for numeric array data. For example, creating a sequence of sine function data for the range 0 to 4 using basic data types and imperative programming would be the commands shown below.

```
# Performing calculation with basic data types and imperative programming
from math import pi, sin
xvals = [i*0.1 for i in range(int(4*pi/0.1))]
yvals = [sin(x) for x in xvals]

# Plotting
import matplotlib.pyplot as plt
plt.plot(xvals, yvals)
plt.show()
```

The Numpy module provides conceptually simpler Numpy array data types with various arithmetic and powerful array processing functions.

```
# Performing calculation with Numpy
import numpy as np      # Must memorise
xvals = np.arange(0, 4*np.pi, 0.1)
yvals = np.sin(xvals)
```

CO1: perform vector and matrix operation using computer software
--

### References:

\* <https://lectures.scientific-python.org/intro/numpy/index.html>

### 1. Input-Output and File Types

An **input** refers to anything that computer “gets” data and store into computer memory. So “keyboard” is an input, “mouse” is an input, “tablet” is an input, a “computer file” on our desktop is an input, etc.

An **output** refers to anything that computer “displays” or “stores” data from computer memory. So a “computer monitor” is an output, a “printer” is an output, a “computer file” on our desktop is an output, etc.

To illustrate a simple input-output between “keyboard” and “screen” (is “teletype terminal” a more precise term?), we will play with the following Python commands:

```
width = 70
pre = "\n"*2 + ""*width + "\n" + " "*(width-2) + ""*\n"
post = ""*\n" + " "*(width-2) + ""*\n" + ""*width
greet = "\n\nEnter your name: "
name = input(greet); print(pre+("Hello "+name).center(width-2)+post)
```

The command input in line 5 reads from “keyboard”, displays what you type on “screen” and stores the string into the variable name.

However, there is a major problem with “keyboard” and “screen” --- the data which is keyed in and displayed on “screen” will disappeared once we turn off the computer. A “computer file” is something that

will remain in computer even after we have turned off a computer and is hence the best choice for storing and retrieving data related to scientific computing.

Similar to humans speaking many different languages, computer files also “storing” many different file formats. All file formats can be categorised into two file types --- *text* file type and *binary* file type. The basic operations associated with a file are “open”, “read”, “write” and “close”. The “save” function is the same as opening file for writing and after writing data into it, close it.

```
Open text file for reading:    fp=open("f", "rt")
Read text file:              x=fp.readlines()
Open text file for writing:    fp=open("g", "wt")
Write text file:             fp.writelines(x)
Open binary file for reading: fp=open("f", "rb")
Read binary file:            M=fp.read()
Open binary file for writing:  fp=open("f", "wb")
Write binary file:           fp.write(x)
Close file:                  fp.close()
```

#### **Purpose of this section:**

- \* Know the commands to use when reading text data
- \* Combine with the string manipulation methods such as `.split()`, array indexing, `int()`, `float()`, etc. we can extract information from text data.

## **2. Numpy Array Data Types**

**Array = multi-index, homogeneous** (elements are of the **same data type**) data structure.

Python’s Numpy array `numpy.ndarray` is an n-dimensional (nD) array object which is also known as *tensor* in the tensorflow machine learning package.

### **2.1. Numpy Array Construction: Creating Vectors, Matrices, ...**

#### **2.1.1. Constructing arrays with no particular pattern**

```
# 1D array = Vector
A1 = np.array([1.0, 3.5, 4.2, 2.3, 3.4, 1.5]) # All double
A2 = np.array([10,11,12,13,14,15])           # All integer
A3 = np.array([7, 19, 19, 18], dtype='double') # All double
A4 = np.array([True,False,True,False,True])   # All Boolean

# 2D arrays = Matrix
B1 = np.array([ [7,19], [19,18] ]) # brackets in brackets; integers
B2 = np.array([ [7,19],[19,18] ], dtype='double') # floating points
B12 = np.array([ [np.tan(np.pi/3), 3/np.sin(np.pi/4)],
                  [np.log(np.cos(np.pi/6)), 1+np.exp(1.5)] ])

# 3D arrays = ArrayTensor (e.g. coloured image)
C1 = np.array([[[1,2],[1,4],[5,1]], [[7,2],[9,3],[8,8]]])
#
#      C1[0,:,:]      C1[1,:,:] =
#      1  2          7  2
#      1  4          9  3
#      5  1          8  8
#
```

Note that Python’s ImageIO module has a class called `Image` which is a subclass of `np.array` to represent coloured images.

#### **Purpose of this section:**

- \* Memorise the command `np.array()` which allows you to create simple 1D or 2D arrays.
- \* They are not for general use. For general array construction for the application in scientific computing or financial mathematics, we need arrays with particular patterns.

### 2.1.2. Constructing arrays with particular patterns --- identity matrices, diagonal matrices, etc.

```
# Vector --- One dimensional (1D) array
A5 = np.zeros(10)          # 10 zeros of data type double
A6 = np.ones(10)           # 10 ones of data type double
A7 = np.full(10,100)       # 10 hundreds of data type integer
A8 = np.linspace(0,np.pi,num=51,endpoint=True)
A9 = np.arange(0,10,2)     # or np.r_[0:10:2]

# Matrix --- Two dimensional (2D) array
B3 = np.zeros((2,4))
B4 = np.ones((4,2))
B5 = np.full((3,5),100)
B6 = np.eye(4)             # 4 x 4 identity matrix
B7 = np.eye(3,2)           # 3 x 2 identity matrix
B8 = np.diag([5,7,-3,4])   # Create a diagonal matrix
B9 = np.diag(np.arange(6,2,-1))
# Or: B9 = np.zeros((4,4)); np.fill_diagonal(B9, np.arange(6,2,-1))
B10 = B9.diagonal() or np.diag(B9) # Get the diagonal of a matrix!
                                     # 13 12 11 10
B11 = np.vander([1,3,2,5]) # 33 32 31 30
                                     # 23 22 21 20
                                     # 53 52 51 50

# Three dimensional (3D) arrays
C3 = np.zeros((2,4,3))
C4 = np.ones((4,2,3))
C5 = np.full((3,5,3),100)
```

The command `np.linspace` is usually used in the creation of a 1-D array for the interval of a particular function  $f$ . In particular,

$$A8 = 0, \frac{2}{50}, \frac{4}{50}, \dots, \frac{49}{50}, \dots$$

Note that interval  $[0, \pi]$  is cut into 50 intervals with 51 points.

#### Constructing an array with random patterns:

```
# Random matrices
B16 = np.random.rand(3, 2)      # 3x2 random matrix uniform over [0,1)
B17 = np.random.random((3,2))  # 3x2 random matrix uniform over [0,1)
B18 = np.random.randn(3, 2)    # 3x2 random matrix Normal(0,1)

# Random three dimensional arrays
C13 = np.random.rand(3,2,4)     # ~ Uniform[0,1)
C14 = np.random.random((3,2,4)) # ~ Uniform[0,1)
C15 = np.random.randn(3,2,4)    # ~ Normal(0,1)
```

#### Purpose of this section:

- \* Memorise the all commands which allows you to create simple 1D or 2D arrays which specific patterns (zeros, ones, linspace, diagonals, etc.)
- \* Know how to construct arrays with  $[0,1)$ -uniform and standard normal distribution patterns.

### 2.1.3. Constructing arrays for 2D and 3D plotting (for Topic 4)

```
#
# Grid is for vectorised evaluations of n-D scalar/vector fields
#
# Two dimensional grids:
B15a, B15b = np.meshgrid([1,2,3],[2,5,7,9]) # return two 4x3 matrices
#   [1, 2, 3]           [2, 2, 2]
#   [1, 2, 3]           [5, 5, 5]
#   [1, 2, 3]           [7, 7, 7]
#   [1, 2, 3]           [9, 9, 9]
# Note: default indexing = 'xy' for computer graphics (Topic 4)

#
# A three dimensional grid
#
#   X x Y x Z = [x1,x2] x [y1,y2,y3] x [z1,z2]
#
XR, YR, ZR = [1,2], [3,5,7], [8,9] # R for range
XP, YP, ZP = np.meshgrid(XR, YR, ZR,indexing='ij') # P for grid points
C6          C7          C8
|           |           |
V           V           V
[[[1 1]      [[3 3]      [[8 9]
 [1 1]      [5 5]      [8 9]
 [1 1]]     [7 7]]     [8 9]]

[[2 2]      [[3 3]      [[8 9]
 [2 2]      [5 5]      [8 9]
 [2 2]]     [7 7]]     [8 9]]

#
# It can be used for the computation of the scalar field  $f(x,y,z) = x^2 + y^2 + z^2$ 
#
#   def f(X, Y, Z): return X**2 + Y**2 + Z**2
#   f = np.vectorize(f)
#   arr = f(XP, YP, ZP)
#
# Related: np.mgrid[1:3,3:8:2,8:10] # mgrid does not accept list
# Related: np.ogrid[1:3,3:8:2,8:10] # => (2,1,1), (1,3,1), (1,1,2)
```

---

## 2.2. Basic Array Attributes (Shape, Size)

Arrays are created with a particular shape and data type. The information or attributions associated with arrays can be obtained from the Numpy array.

- \* Get the dimension of an array: `A.ndim`.
- \* Get the shape of an array: `A.shape`.
- \* Get the number of elements in an array: `A.size`.
- \* Get the total number of bytes used: `A.nbytes`, it is defined as `A.size*A.itemsize`. For example, if there are  $n$  elements in `A` and all the elements are 64-bit floating numbers, then the total number of bytes used in `A` to store array data is  $8n$ .

There are a few operations which are related to the attributes of an array:

- \* Transpose: It works by reversing the 'indices' but for real-world application, it is used to transpose a matrix `A` to `A.T` (or `A.transpose()`), for example:

```
# 1D transpose: No change
[ a b c d ] ---> [ a b c d ]

# 2D transpose: Change a matrix to its transpose
[ a b c ]
[ d e f ]      [ a d g j ]
[ g h i ] --->  [ b e h k ]
[ j k l ]      [ c f i l ]

[ a ]
[ b ]      ---> [ a b c ]
[ c ]

* Change from  $n$ -D array to 1-D array: A.ravel(), A.flatten()

# 1D ravel/flatten: No change
[ a b c d ] ---> [ a b c d ]

# 2D ravel/flatten
[ a b c ]
[ d e f ]      ---> [ a b c d e f g h i ]
[ g h i ]

* Change an array to a compatible shape: A.reshape(( $d_1, d_2, \dots, d_r$ ))

# np.array([1,3,5,7,9,5]).reshape((2,3))
[ 1 3 ]
[ 1 3 5 7 9 5 ] ---> [ 5 7 ]
[ 9 5 ]

* Change an array to any shape: np.resize(A, ( $d_1, d_2, \dots, d_r$ ))

# A = np.array([1,3,5,7,9,5])
# B = np.resize(A, (2,2))
[ 1 3 ]
[ 1 3 5 7 9 5 ] ---> [ 5 7 ]
# C = np.resize(A, (3,5))
[ 1 3 5 7 9 ]
[ 1 3 5 7 9 5 ] ---> [ 5 1 3 5 7 ]
[ 9 5 1 3 5 ]
```

### Purpose of this section:

- \* Memorise the all commands which allows you to check the shape of array and to manipulate shapes: `A.shape`, `A.T`, `A.reshape()`, etc.

### 2.3. Numpy Array Formatting

**Get the current format:** `np.get_printoptions()`  
"Default values:" precision = 8, threshold = 1000, edgeitems = 3, linewidth = 75 characters per line, suppress = False, i.e. do not print small floating point values using scientific notation, nanstr = 'nan', infstr = 'inf', formatter = None (a dictionary of types to set the formatting options), sign = '-', floatmode = 'maxprec'.  
**Set the format:** `set_printoptions(args)`  
args are the keywords above. We usually need to set the linewidth for printing a nicer matrix.  
**Print the matrix:** `print(A)`

**Example.** Print the matrix

```
[ 1/2  1/3  1/4  1/5  1/6  1/7  1/8  1/9 ]
[ 1/8  1/9  1/8  1/7  1/6  1/5  1/4  1/3 ]
```

using 'printoptions'.

**Sample Solution 1:**

```
A = np.array([[1/2, 1/3, 1/4, 1/5, 1/6, 1/7, 1/8, 1/9],
              [1/8, 1/9, 1/8, 1/7, 1/6, 1/5, 1/4, 1/3]])
print(A)
# Temporarily set the options
with np.printoptions(precision=4, linewidth=100):
    print(A)
print(A)
```

**Sample Solution 2:**

```
A = np.array([[1/2, 1/3, 1/4, 1/5, 1/6, 1/7, 1/8, 1/9],
              [1/8, 1/9, 1/8, 1/7, 1/6, 1/5, 1/4, 1/3]])
print(A)
default_printoptions = np.get_printoptions()
# Permanently set the options
np.set_printoptions(precision=4, linewidth=100)
print(A)
np.set_printoptions(**default_printoptions)
print(A)
```

---

For an array with large variance, using a more complex 'printoptions' is helpful.

```
A = np.array([0.0123, 12e-7, 23e6])
print(A)          # All in Engineering notations
# Temporarily set the options
with np.printoptions(formatter={'float': '{: .4g}'.format}):
    print(A)      # Some in easier to read decimal format
```

**Purpose of this section:**

- \* Useful in writing Python script. E.g. `np.printoptions(precision=6, threshold=2000, linewidth=150)`. It is not required for final exam.
- \* Alternatively, we may use  
`import sys`  
`np.savetxt(sys.stdout, A, fmt="%.6f")`  
according to <https://stackoverflow.com/questions/9360103/how-to-print-a-numpy-array-without-brackets>.

## 2.4. Numpy Array Applications (Purpose: NOT For Exam, for knowledge)

- \* **Array of numbers** can be used to represent
    - time series, audio / sound signals (mostly 1D)
    - black-and-white images, grayscale images (2D)
    - Colour images (3D)
      - + RGB colour =  $M \times N \times 3$  with values in [0,1] or 0--255;
      - + RGBA colour =  $M \times N \times 4$  with values in [0,1] or 0--255 with A=transparency

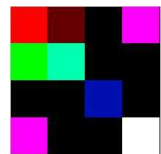
Note that out-of-range RGB(A) values are clipped.
  - \* **Array of characters** can be used to represent
    - ASCII / Unicode arts
  - \* **Array of Booleans** can be used for selection as to be discussed later.
- Python tools to visualise a matrix as an image:

```
matplotlib.pyplot.spy(Z) # 2D array only
matplotlib.pyplot.imshow(X, cmap=None, norm=None, *, aspect=None,
    interpolation=None, alpha=None, vmin=None, vmax=None, origin=None,
    extent=None, interpolation_stage=None, filternorm=True,
    filterrad=4.0, resample=None, url=None, data=None, **kwargs)
```

For grayscale images, use `cmap='gray'`, `vmin=0`, `vmax=255`.

**Example.** (Array of Numbers) A 4x4 pixel coloured image below can be represented as Numpy array:

```
P3
4 4
15
# https://paulbourke.net/dataformats/ppm/
255 0 0 100 0 0 0 0 0 255 0 255
0 255 0 0 255 175 0 0 0 0 0 0
0 0 0 0 0 0 0 15 175 0 0 0
255 0 255 0 0 0 0 0 0 255 255 255
```



The following is how we can read and display the 4x4 pixel image.

```
import numpy as np
# https://liaohaohui.github.io/UECM1703/test.ppm
fp = open("test.ppm", "r")
lines = []
while True:
    line = fp.readline()
    if not line: break # Break out of loop when no more lines
    # We skip any empty line and lines with are comment
    if len(line)>0 and line[0] != '#':
        lines.append(line.strip())
if lines[0][0]=="P" and int(lines[0][1:])==3: # P3=Coloured image
    W, H = lines[1].split()
    W = int(W)
    H = int(H)
    m = int(lines[2])
    if m == 255:
        imgarr = np.zeros((H,W,3), dtype='uint8')
    elif m == 65535:
        imgarr = np.zeros((H,W,3), dtype='uint16')
    image_contents = [int(c) for c in " ".join(lines[3:]).split()]
    for y in range(H):
        for x in range(W):
            imgarr[y,x,:] = image_contents[(y*3*W+3*x):(y*3*W+3*x+3)]
print(imgarr)

from PIL import Image
img = Image.fromarray(imgarr)
```

jgs



### 3. Array Mathematical Functions and Numpy Ufuncs

When we encounter an array below:

```
from math import sin
A = np.array([sin(1), sin(2), sin(3), sin(4), sin(5), sin(6)])
B = np.array([[sin(1), sin(2), sin(3)],
              [sin(4), sin(5), sin(6)]])
```

we would hope to **abbreviate** it as

```
fA = sin(np.array([1,2,3,4,5,6]))
fB = sin(np.array([[sin(1), sin(2), sin(3)],
                  [sin(4), sin(5), sin(6)]]))
```

However, the sine function from math module would **complain**.

Numpy provides two solutions:

- \* Build the commonly use mathematical functions in. For example, use `np.sin` instead of `sin`
- \* ‘Vectorise’ the function (using Numpy’s universal function framework with 1 input and 1 output):  
`arsin = np.vectorize(sin)`
- \* `np.frompyfunc(func, nin, nout, *[, identity])` Note that `nin` and `nout` are the number of inputs and number of outputs of the function `func` respectively. In this case, `arsin = np.frompyfunc(sin,1,1)`

**Example.** (Application: Plotting) By use the plotting functions `plt.plot` and `plt.show` to draw (a) Sine function; (b) Cosine function; and (c) floor function for the domain  $[-2i, 2i]$ .

**Solution:** The question does not say how small is the step size, we will just split  $[-2\pi, 2\pi]$  to 100 equal intervals.

```
import numpy as np, matplotlib.pyplot as plt
xr = np.linspace(-2*np.pi, 2*np.pi, 101) # x range
y1 = np.sin(xr)
y2 = np.cos(xr)
y3 = np.floor(xr)
plt.plot(xr, y1, xr, y2, xr, y3)
plt.show() # This is not needed in Spyder/Jupyter
```

If we have a computer to plot the graph, we can see that floor looks ugly, this is because `plt.plot` just join points and we need break the domain into more intervals to make the plot of floor function nice.

**Exercise.** Try and see if you can define a ‘vectorise’ function for

$$\operatorname{sinhc}(x) = \begin{cases} \frac{e^x - e^{-x}}{2x}, & x \neq 0, \\ 1, & x = 0. \end{cases}$$

so that you can use it to calculate `sinhc(np.linspace(-2*np.pi, 2*np.pi, 101))`.

□

**Example.** Consider the heart disease data from [https://bookdown.org/brianmachut/uofm\\_analytics\\_r\\_hw\\_sol\\_2/logreg.html](https://bookdown.org/brianmachut/uofm_analytics_r_hw_sol_2/logreg.html) by analysing the relation between  $X=\text{fast\_food\_spend}$  and  $Y=\text{heart\_disease}$ .

One mathematical model for fitting the data is called logistic regression model:

$$P(Y = 1|X = x) = \frac{1}{1 + \exp(-(-10.651330614 + 0.002199567x))}$$

By using the array processing knowledge, write a Python script to read heart\_data.csv and express the logistic regression model as Python function.

**Sample Python Script Solution:**

```
import numpy as np
# heart_data.csv can be downloaded from
# https://bookdown.org/brianmachut/uofm_analytics_r_hw_sol_2/logreg.html
data = np.loadtxt("heart_data.csv", delimiter=",",
                  skiprows=1, dtype=np.double)

col1 = data[:,0]
col3 = data[:,2]
from matplotlib.pyplot import plt
plt.plot(col3,col1,"*")
plt.show()

# Expressing logistic regression model as Python functions
def log_reg(x):
    return 1.0/(1.0+np.exp(-(-10.651330614 + 0.002199567*x)))
```

---

**Purpose of this section:**

- \* Recognise the different between a function (from a number to a number) and Numpy vectorised function (operate on the array with mapping elementwise)
- \* In option pricing, we will encounter call option and put option which are defined as  $\max(0, S-K)$  and  $\max(0, K-S)$  respectively. Here  $K$  is usually a constant and  $S$  can be an array (representing a range).

#### 4. Array Indexing: Sub-arrays, ... (Important)

Indexing is an important way to access the data in an  $n$ -D array. Indexing an array with an integer / integers will lead to the reduction of dimension by default. To keep the dimension, we either use a range  $m:n:s$  or use an 'extra' index called 'None' to keep the dimension. We need to note that indexing an array  $A$  on gives us a **view** of the array  $A$ .

- \* Return a "view" of  $A$  with a given shape  $(m_1, \dots, m_k)$ . Note that  $m_1 \times \dots \times m_k$  must be equal to  $A.size$ : `A.reshape((m1, ..., mk))`.
- \* Return a "view" of  $(m_1, m_2, \dots, m_k)$ -array  $A$  as a transpose with a shape  $(m_k, \dots, m_2, m_1)$ : `A.T` (alternatively, `np.transpose(A)`).

Making any changes to the **view** will be reflected on the original array. If we need a **copy** of the sub-array from  $A$ , we need to use the `.copy()` method or stacking commands:

- \* Return a "copy" of  $A$  with a specific type: `A.astype(sometype)`, here `sometype` can be 'double', 'bool', 'int8', etc.
- \* Return a new array by stacking existing array(s): `np.hstack` (stacking array horizontally) and `np.vstack` (stacking array vertically).

```
np.hstack((A_1,A_2,...,A_k)) : A_1 A_2 ... A_k
```

```
np.vstack((A_1,A_2,...,A_k)) : A_1
                             A_2
                             .
                             .
                             A_k
```

##### 4.1. Usual Indexing :n, m:n, m:, m:n:k, :

- \* In this section,  $m$  and  $n$  are assumed to be **non-negative**.
- \* Python's index starts from 0
- \* Python's ending index  $m:n$  will never reach  $n$

##### Example.

- \* `:` is similar to 'take all' (this depends on the shape of the array)
- \* `:12` is similar to `0,1,2,3,4,5,6,7,8,9,10,11` or `range(12)`
- \* `5:12` is similar to `5,6,7,8,9,10,11` or `range(5,12)`
- \* `2:12:3` is similar to `2,5,8,11` or `range(2,12,3)`
- \* `12:2:-2` is similar to `12,10,8,6,4` (2 will not be reached) or `range(12,2,-2)`

Without loss of generality, we consider a 2-D array  $A$ .

(I) `A[m-1, n-1]` : Indexing an element of  $A$  at  $(m, n)$  (dim=0)

(II) `A[m-1, :]` : Indexing  $A$  at  $m$ -row (dim=1)

(III) `A[:, n-1]` : Indexing  $A$  at  $n$ -column (dim=1)

(IV) `A[m1-1:m2, n1-1:n2]` :

Indexing a sub-array of  $A$  bounded by  $m1$ -row to  $m2$ -row and  $n1$ -column to  $n2$ -column

(V) `A[m1-1:m2:ms, n1-1:n2:ns]` :

Indexing a sub-array of  $A$  from  $m1$ -row to row- $m2$  by step  $ms$  and from  $n1$ -column to  $n2$ -column by step  $ns$ . When  $m1-1$  is ignored, it assumes 0 and when  $m2$  is ignored, it assumes the last row and when  $ms$  is ignored, it assumes 1. The situations are similar for  $n1-1$ ,  $ns$  and  $n2$ .

(VI) `A[[r1-1, ..., rk-1], :][:, [c1-1, ..., cl-1]]` :

Indexing a sub-array of  $A$  using rows  $r1, \dots, rk$ , columns  $c1, \dots, cl$ .

Numpy provides alternative indexing functions `take` and `put` to take and assign values to a slice of an array. They are less convenient to use, we will skip them:

- (a) `A.item(3,4)`
- (b) `A.take(indices=[3],axis=0)`
- (c) `A.take(indices=[4],axis=1)`
- (d) `A.take(range(3,7),axis=1).take(range(1,3),axis=0)`
- (d) `A.take([3,5,7],axis=1).take([1,3,5],axis=0)`

**Example.** (Final Exam Sept 2015, Q3(a)) The following vector is defined in Python

```
V = np.array([2,7,-3,5,0,14,-1,10,-6,8])
```

What will be displayed if the following variables B, C and D are printed.

- (i) `B = V[[1,3,4,5,6,9]]`

**Solution:** We first index the array V:

	0	1	2	3	4	5	6	7	8	9	<-- indices for V
V =	2	7	-3	5	0	14	-1	10	-6	8	
B =	[7	5	0	14	-1	8]					

- (ii) `C = V[[8,2,1,9]]`

**Solution:** `C = [-6 -3 7 8]`

- (iii) `D = np.array([V[[0,2,4]],V[[1,3,5]],V[[2,5,8]]])`

**Solution:** `D = [[2, -3, 0], [7, 5, 14], [-3, 14, -6]]`

**Example.** Consider the array generated with `A = np.arange(1,55,dtype='double').reshape(6,9).`

```
A =
  1  2  3  4  5  6  7  8  9
10 11 12 13 14 15 16 17 18
19 20 21 22 23 24 25 26 27
28 29 30 31 32 33 34 35 36
37 38 39 40 41 42 43 44 45
46 47 48 49 50 51 52 53 54
```

Write down the output of the following commands:

- (a) `print(A[3,4])`
- (b) `print(A[3,:])`
- (c) `print(A[:,4])`
- (d) `print(A[1:3,3:7])`
- (e) `print(A[1:6:2,:][:,3:8:2])`
- (f) `print(A[3,None])` (It is the same as `A[None,3]`)
- (g) `print(A[:,4,None])` or `print(A[:,4:5])`

Indexing (I)  
Indexing (II)  
Indexing (III)  
Indexing (IV)  
Indexing (V) & (VI)

**Example.** (Final Exam Sept 2015, Q1(a)) The following matrix is defined in Python:

```
M =
  6   9  12   4   3   0
  4   4  15   2   1   1
  2   1  18  -5   8   2
 -6  -4  21   1  -5   2
```

What will be displayed if the following variables A in (i) to C in (iii) are printed?

- (i) A = M[[0,2],:][:,[1,3]]
- (ii) B = M[:,[0,3,4,5]]
- (iii) C = M[1:3,:]

We have to be careful with the indexing in Numpy because it only create **views** and does not create **copies**.

**Example.** (View vs Copy of a sub-array) Study the following Python instructions and explain what is each output:

Assign the 'view'	Assign the 'copy'
<pre>import numpy as np A = np.array([[1,2,3],[4,5,6]]) B = A[:,0:2] B[0,0] = 8 print("A=",A) print("B=",B)</pre>	<pre>import numpy as np A = np.array([[1,2,3],[4,5,6]]) B = A[:,0:2].copy() B[0,0] = 8 print("A=",A) print("B=",B)</pre>
<pre>A = [ 8  2  3 ] [ 4  5  6 ]  B = [ 8  2 ] [ 4  5 ]</pre>	<pre>A = [ 1  2  3 ] [ 4  5  6 ]  B = [ 8  2 ] [ 4  5 ]</pre>

Summary of Numpy Indexing from [https://lectures.scientific-python.org/intro/numpy/array\\_object.html](https://lectures.scientific-python.org/intro/numpy/array_object.html)

```
>>> a[0, 3:5]
array([3, 4])

>>> a[4:, 4:]
array([[44, 55],
       [54, 55]])

>>> a[:, 2]
a([2, 12, 22, 32, 42, 52])

>>> a[2::2, ::2]
array([[20, 22, 24],
       [40, 42, 44]])
```

0	1	2	3	4	5
10	11	12	13	14	15
20	21	22	23	24	25
30	31	32	33	34	35
40	41	42	43	44	45
50	51	52	53	54	55

## 4.2. Negative Indexing: Index from End

Indexing array with the usual bounded index is often enough but Python provides “**Negative**” indexing which is used to index an array “from last element”. For example, `-1` refers to the last index, `-n` refers to the `n` last index. Note that going beyond index bound can cause error!

**Example.** Consider the following integer array:

```
A = 

|   |    |    |    |    |    |    |    |    |    |   |
|---|----|----|----|----|----|----|----|----|----|---|
| [ | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | ] |
| [ | 19 | 20 | 21 | 22 | 23 | 24 | 25 | 26 | 27 | ] |
| [ | 28 | 29 | 30 | 31 | 32 | 33 | 34 | 35 | 36 | ] |
| [ | 37 | 38 | 39 | 40 | 41 | 42 | 43 | 44 | 45 | ] |
| [ | 46 | 47 | 48 | 49 | 50 | 51 | 52 | 53 | 54 | ] |


```

What is the output of the following commands:

- (a) `print(A[-2,4])`
- (b) `print(A[-2,:])`
- (c) `print(A[-2,None])`
- (d) `print(A[:,-4])`
- (e) `print(A[:,-4:-5:-1])`
- (f) `print(A[1:-3,3:-4])`
- (g) `print(A[1:-1:2,:][:,2:-2:2])`

**Solution:**

- (a) 41 (second last row, fifth column)
- (b) `[37 38 39 40 41 42 43 44 45]` (**1D**) (second last row)
- (c) `[[37 38 39 40 41 42 43 44 45]]` (different from part (b) in terms of dimension, this is **2D**, i.e. a 1x9 ‘matrix’)
- (d) `[ 6, 15, 24, 33, 42, 51]` (fourth last column = sixth column, **1D**)
- (e) `[[ 6] [15] [24] [33] [42] [51]]` (**2D**)
- (f) `[[13 14] [22 23]]` (second row to the row before the last third row **and** fourth column to before the last fourth column)
- (g) `[[12 14 16] [30 32 34]]` (second and fourth rows **and** third, fifth and seventh columns)

**Purpose of this section and previous section:**

- \* You must memorise and understand `:`, `1:`, `:9`, `::2`, `::-1`, `9::-1`, `-1::-1`, `1:9:2`, etc. Just be careful that the last index will not be reached.

### 4.3. Usual Indexing vs For-Loops

Indexing will perform array operations internally similar to for-loops and therefore can be used to express for loops into array indexing patterns.

**Example.** (Combining Programming and Array Indexing in Solving Differential Equations Numerically) In applying finite difference approximation to the type 1 ODE-BVP:

$$\ddot{y}(x) + p(x)\dot{y}(x) + q(x)y(x) = r(x), \quad a < x < b, \quad y(a) = y_a, \quad y(b) = y_b$$

we obtain matrix

$$C = \begin{bmatrix} 1 - \frac{h}{2} p(x_1) & 0 & \frac{h}{2} p(x_1) & \cdots & 0 \\ h^2 q(x_1) - 2 & 1 + \frac{h}{2} p(x_1) & \cdots & \cdots & 0 \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ 0 & \cdots & 1 - \frac{h}{2} p(x_{n-1}) & \frac{h}{2} p(x_{n-1}) & 0 \\ 0 & \cdots & h^2 q(x_{n-1}) - 2 & 1 + \frac{h}{2} p(x_{n-1}) & 0 \\ 0 & 0 & 0 & 0 & 1 \end{bmatrix}, \quad x_i = a + i \cdot h, \quad h = \frac{b-a}{n}.$$

Write a Python script to generate the matrix C.

**Sample Solution 1:** By using for-loop, we have

```
C1 = np.eye(n+1)
x = a
h = (b-a)/n
for i in range(1,n):
    x = x+h
    C1[i,i-1] = 1.0-p(x)*h/2.0
    C1[i,i] = h**2*q(x)-2.0
    C1[i,i+1] = 1.0+p(x)*h/2.0
```

**Sample Solution 2:** By using array indexing, we have

```
C2 = np.eye(n+1)
idx = np.r_[1:n]
h = (b-a)/n
x = a + h*idx
C2[idx,idx-1] = 1.0-p(x)*h/2.0      # Using elementwise arithmetic: idx-1
C2[idx,idx] = h**2*q(x)-2.0
C2[idx,idx+1] = 1.0+p(x)*h/2.0
```

**Example.** (Combining Programming and Array Indexing in Technical Analysis of Financial Data) In the past, it is possible for us to download a lot of stock price data from Yahoo!Finance. However, Yahoo!Finance is providing less and less stock price data for data and is transforming itself to pay-per-use service. The stock price data of Telekom Malaysia (a listed company which provides the most expensive broadband service in South East Asia) below was downloaded a few years ago.

```
Date,Open,High,Low,Close,Volume,Adj Close
2016-12-30,6.06,6.09,5.81,5.95,5842300,5.95
2016-12-29,6.05,6.12,5.98,6.06,6777900,6.06
2016-12-28,5.96,6.06,5.96,6.03,2503700,6.03
2016-12-27,5.93,5.99,5.92,5.99,922400,5.99
2016-12-26,5.95,5.95,5.95,5.95,000,5.95
2016-12-23,5.91,5.97,5.91,5.95,838100,5.95
2016-12-22,5.97,5.97,5.92,5.95,1065000,5.95
2016-12-21,6.02,6.02,5.92,5.95,3405900,5.95
2016-12-20,5.95,6.07,5.94,5.98,3101400,5.98
2016-12-19,5.97,6.00,5.91,5.95,2006000,5.95
2016-12-16,5.90,5.96,5.89,5.95,3975600,5.95
2016-12-15,5.91,5.95,5.90,5.90,3987400,5.90
2016-12-14,5.96,6.00,5.93,5.95,5128000,5.95
2016-12-13,6.04,6.05,5.94,5.96,4111000,5.96
2016-12-12,6.03,6.03,6.03,6.03,000,6.03
```

```

2016-12-09,6.11,6.11,6.01,6.03,1573000,6.03
2016-12-08,6.16,6.20,6.11,6.11,3189800,6.11
2016-12-07,6.13,6.15,6.09,6.11,4564800,6.11
2016-12-06,6.12,6.15,6.09,6.12,2976600,6.12
2016-12-05,6.15,6.19,6.13,6.14,3303800,6.14
2016-12-02,6.15,6.22,6.09,6.13,2134000,6.13
2016-12-01,6.17,6.24,6.14,6.15,6188400,6.15

```

By using the closing price, write a Python program to calculate

- \* the price difference between the next day and today for December 2016;
- \* the three-day (moving) average for December 2016.

#### Sample Solution 1:

```

import numpy as np
dclose = np.array([6.15, 6.13, 6.14, 6.12, 6.11, 6.11, 6.03,
                   6.03, 5.96, 5.95, 5.9, 5.95, 5.95, 5.98,
                   5.95, 5.95, 5.95, 5.95, 5.99, 6.03, 6.06, 5.95])
price_diff = np.zeros(dclose.size-1)
moving3 = np.zeros(dclose.size-2)
for today in range(price_diff.size):
    next_day = today + 1
    price_diff[today] = dclose[next_day]-dclose[today]
for today in range(moving3.size):
    next_day = today + 1
    next_2day = today + 2
    moving3[today] = (dclose[today]+dclose[next_day]+dclose[next_2day])/3
print("Price difference between next day and today for December 2016: ")
print(price_diff)
print("3-D moving average for December 2016:", moving3)

```

#### Sample Solution 2:

```

import numpy as np
dclose = np.array([6.15, 6.13, 6.14, 6.12, 6.11, 6.11, 6.03,
                   6.03, 5.96, 5.95, 5.9, 5.95, 5.95, 5.98,
                   5.95, 5.95, 5.95, 5.95, 5.99, 6.03, 6.06, 5.95])
price_diff = np.diff(dclose)
idx = np.r_[: (dclose.size-2)]
moving3 = (dclose[idx]+dclose[idx+1]+dclose[idx+2])/3.0
print("Price difference between next day and today for December 2016: ")
print(price_diff)
print("3-D moving average for December 2016:", moving3)

```

You can also use the information from [https://rosettacode.org/wiki/Averages/Simple\\_moving\\_average](https://rosettacode.org/wiki/Averages/Simple_moving_average) to write a better Python script.

---

**Example.** (More Complex Array Indexing) Write a Python function `spiral(n)` to generate an  $n \times n$  clockwise spiral matrix using Python. For example, `spiral(5)` gives

```

[  0  1  2  3  4 ]
[ 15 16 17 18 5 ]
[ 14 23 24 19 6 ]
[ 13 22 21 20 7 ]
[ 12 11 10 9  8 ]

```

**Sample Solution:** It is inspired by [https://rosettacode.org/wiki/Spiral\\_matrix](https://rosettacode.org/wiki/Spiral_matrix)



```

def spiral(n,m=None):
    _n,_m = (n,m) if m is not None else (n,n)
    _nl,_ml=0,0
    dx,dy = 0,1          # Starting increments
    x,y = 0,0            # Starting location
    import numpy as np
    myarray = np.zeros((_n,_m),dtype='int')
    for i in range(_n*_m):
        myarray[x,y] = i
        nx,ny = x+dx, y+dy # (dx,dy) = direction to update array
        if _nl<=nx<_n and _ml<=ny<_m:
            x,y = nx,ny
        else:
            if dx==0 and dy==1:
                _nl+=1; dx,dy=1,0
            elif dx==1 and dy==0:
                _m-=1; dx,dy=0,-1
            elif dx==0 and dy==-1:
                _n-=1; dx,dy=-1,0
            elif dx== -1 and dy== 0:
                _ml+=1; dx,dy=0,1
            else:
                return None # Should not reach this state
            x,y = x+dx,y+dy
    return myarray

```

---

## 5. Arithmetic, Logical and Relational Operations

The basic arithmetic and logical operations for “numbers” such as +, −, x, /, power, not, and, or, equality, etc. are generalised to operate on “arrays”. as **elementwise arithmetic and logical operations**.

### 5.1. Elementwise Arithmetic Operations and Broadcasting

The basic arithmetic +, −, x, / and power for **two arrays of the same shape** A and B are just element-wise addition, subtraction, multiplication, division and power of numbers in the array as:

\* −A: elementwise negation

1-D example	2-D example
A = [ 1.2 1.3 1.4]	A = [ 3.2 -5.3 ]
−A = [−1.2 −1.3 −1.4]	[ −5.5 4.0 ]
	−A = [ −3.2 5.3 ]
	[ 5.5 −4.0 ]

\* A + B, A − B: elementwise addition and subtraction

1-D example	2-D example
A = [ 1.2 1.3 1.4]	A = [ 1 -2 ]
B = [ 2.1 2.3 3.4]	[ −3 4 ]
A+B =	B = [ 8 7 ]
[ 3.3 3.6 4.8]	[ −6 −5 ]
A−B =	A+B =
[−0.9 −1.0 −2.0]	[ 9 5 ]
	[ −9 −1 ]
	A−B =
	[ 7 −9 ]
	[ 3 9 ]

\* A \* B, A / B, A \*\* B: elementwise multiplication, division and ‘power’

1-D example	2-D example
A = [ 1.3 1.2 1.4]	A = [ 8 -7 ]
B = [ 2 3 4]	[ −6 5 ]
A*B =	B = [ 4 2 ]
[ 2.6 3.6 5.6]	[ −3 −4 ]
A/B =	A*B =
[0.65 0.4 0.35]	[ 32 −14 ]
A**abs(B) =	[ 18 −20 ]
[1.69 1.728 3.8416]	A/B =
	[ 2. −3.5 ]
	[ 2. −1.25 ]
	A**abs(B) =
	[ 4096 49 ]
	[ −216 625 ]

Together with the ufunc, the array arithmetic allows us to handle computations like  $\sin(x^2 + x + 2)$ :

```
np.sin(x**2 + x + 2)
```

**Example.** Write a Python script to plot the functions

$$y_1 = \frac{x^3}{2} + 3x^2 - 1, \quad y_2 = 2\sin x, \quad y_3 = \sin(2x)$$

in one diagram for the range  $-\pi \leq x \leq \pi$ .

**Sample Solution:**

```
import numpy as np, matplotlib.pyplot as plt
x = np.arange(-np.pi, np.pi, 0.0001)
y1 = x*x*x/2 + 3*x*x - 1
y2 = 2*np.sin(x)
y3 = np.sin(2*x)
plt.plot(x,np.vstack((y1,y2,y3)).T); plt.show()
```

Numpy's elementwise arithmetic can work on arrays with **compatible shapes** through **broadcasting** (see <https://lectures.scientific-python.org/intro/numpy/operations.html#broadcasting>). For example, shape (3,4) and shape (4,) are compatible but not shape (3,4) against shape (2,) or shape (3,). Consider

- \* `A = np.arange(1,13).reshape(3,4)`
- \* `B = np.array([5,4,8])` (Shape = (3,))
- \* `C1 = np.array([9,4,8,7])` (Shape = (4,)), `C2 = np.array([9,4])` (Shape = (2,))
- \* `A + C1` is OK but `A + B` and `A + C2` are **not OK**. However, reshaping B to (3,1) will make it compatible with A.

By expanding A, B, C1 and C2, we can see:

$$\begin{aligned}
 A + C1 &= \begin{bmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \end{bmatrix} + \begin{bmatrix} 9 & 4 & 8 & 7 \end{bmatrix} \\
 &= \begin{bmatrix} 1+9 & 2+4 & 3+8 & 4+7 \\ 5+9 & 6+4 & 7+8 & 8+7 \\ 9+9 & 10+4 & 11+8 & 12+7 \end{bmatrix}
 \end{aligned}$$

$$\begin{aligned}
 A + C2 &: \begin{bmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \end{bmatrix} + \begin{bmatrix} 9 & 4 \end{bmatrix} \quad ??? \\
 A + B &: \begin{bmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \end{bmatrix} + \begin{bmatrix} 5 & 4 & 8 \end{bmatrix} \quad ???
 \end{aligned}$$

$$A + B.reshape((3,1)) : \begin{bmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \end{bmatrix} + \begin{bmatrix} 5 \\ 4 \\ 8 \end{bmatrix}$$

**Example.** Write down two Python commands which allows us to transform the left matrix *A* to the right matrix *B* using index operations and array arithmetic:

$$A = \begin{bmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \end{bmatrix} \quad \text{--->} \quad \begin{bmatrix} 1 & 2 & 3 & 4 \\ 0 & -4 & -8 & -12 \\ 0 & -8 & -16 & -24 \end{bmatrix} = B$$
**Sample Solution:**

```
A[1,:] = A[1,:] - 5*A[0,:]
A[2,:] = A[2,:] - 9*A[0,:]
```

**Purpose of this section:**

- \* Just take care not to confuse elementwise matrix multiplication `*` with matrix multiplication `@` and also try to practise with the arithmetic broadcasting operations. The array indexing and array arithmetic is used a lot in scientific computing as in the above example.

## 5.2. Logical Operations

“Boolean arrays” arise when we “compare” number arrays. The logical operations for Boolean arrays are similar to the arithmetic operations for numeric arrays. They are just the generalisation of logical operations from Boolean values (True, False) to Boolean arrays (arrays of True and/or False) of compatible arrays)

Let C and D be Boolean array of the same shape, the element-wise negation, conjunction and disjunction for the Boolean array are:

- \* Check and make sure that `C.dtype` and `D.dtype` are `bool`.
- \* `~C`: elementwise negation
- \* `C & D`: elementwise conjunction
- \* `C | D`: elementwise disjunction

In Numpy, the logical operations also work on **two arrays of the compatible shape**. For example, Shape (2,3) and shape (3,) are compatible but not shape (2,)

- \* `C = np.array([[True, False, True], [False, True, False]])` (Shape = 2x3)
- \* `D1 = np.array([True, False])` (Shape = 2)
- \* `D2 = np.array([False, False, True])` (Shape = 3)
- \* `C & D2` is OK but `C & D1` is **not OK**

Note that Python allows us to use `~`, `&` and `|` to denote `~`, `&` and `|` respectively. However, it is not recommended to prevent confusion because Boolean will be converted to integers when other arithmetic operations are involved.

**Example.** (Final Exam Sept 2015, Q2(a)(iv)) What will display if the following commands are executed?

`np.array([not False, False, not True]) & np.array([True]*3)` (2 marks)

**Solution:** `[True False False] & [True True True] = [True False False]`

**Example.** Find the index of the ‘True’ value for `[False, True, False, False, True]`.

`np.where([False, True, False, False, True])` \_\_\_\_\_

### Purpose of this section:

- \* There is not a lot to say about Boolean arrays. They will be obtained when we employ a predicate on an array A (e.g. `A<=100`) or when we employ array relational operations (e.g. `A<B`).

## 5.3. Relational Operations

Let A and B be arrays of compatible shape. The ordering or real numbers allows us to compare numbers by the relational operations `==`, `!=`, `<`, `<=`, `>` and `>=`.

**Example.** (Final Exam Sept 2013, Q1(c)) Given that `x = np.array([1,3,4,2,5,0,-3])` and `y = np.array([6,3,2,4,1,0,6])`, list the results of the following commands (i) to (iii):

(i) `x - 2*(y>3)`

**Solution:** Let T denote True and F denote False. The calculation is as follows.  
`= x - 2*[T F F T F F T]`  
`= x - [2 0 0 2 0 0 2]`  
`= [-1 3 4 0 5 0 -5]`

(ii) `(x!=0) & (y==0)`

**Solution:** Let T denote True and F denote False. The calculation is as follows.  
`[T T T T T F T] & [F F F F F T F]`  
`= [F F F F F F F]`

(iii) `(x==y) | (y<x)`

**Solution:**  
`= [F T F F F T F] | [F F T F T F F]`  
`= [F T T F T T F]`

#### 5.4. Fancy Indexing with Boolean Array (and List)

The “Boolean” array for an array  $A$  generated with the use of relational operations (or more general predicates) can be used as a kind of **fancy indexing** called **Boolean indexing** for  $A$ .

This kind of indexing is widely used in statistics, image processing, signal processing, etc. because it allows us to **select** the array data of interest.

**Example.** Consider the 2-D array

$$A = \begin{bmatrix} -1 & 2 & 1 & -3 \\ 2 & -4 & -4 & 0 \\ 0 & 0 & -1 & -2 \end{bmatrix}$$

Write the Python commands to

1. list all the values in  $A$  which are **non-negative**.
2. replace the negative values in  $A$  by  $-10$ .

**Sample Solution:**

1. We select those array elements which are 0:

$$\begin{array}{cccc} [-1 \geq 0 & 2 \geq 0 & 1 \geq 0 & -3 \geq 0] & [F & T & T & F] & [ & 2 & 1 & ] \\ [2 \geq 0 & -4 \geq 0 & -4 \geq 0 & 0 \geq 0] & \rightarrow [T & F & F & T] & \rightarrow [2 & & & 0] \\ [0 \geq 0 & 0 \geq 0 & -1 \geq 0 & -2 \geq 0] & [T & T & F & F] & [0 & 0 & & ] \\ & & & \rightarrow [2 & 1 & 2 & 0 & 0 & 0] \end{array}$$

Answer:  $A[A \geq 0]$  # or  $A[\sim(A < 0)]$

2.  $A[A < 0] = -10$  Here's how it works for assignment:

$$\begin{array}{cccc} [-1, & 2, & 1, & -3] & [T, & F, & F, & T] & [-10, & 2, & 1, & -10] \\ [2, & -4, & -4, & 0] & [F, & T, & T, & F] & \rightarrow [2, & -10, & -10, & 0] \\ [0, & 0, & -1, & -2] & [F, & F, & T, & T] & [0, & 0, & -10, & -10] \end{array}$$

**Example.** (Final Exam Sept 2013, Q1(b) with modification) Write a Python script to perform the following actions:

- \* Generate a 2-by-3 array of random numbers using the `rand` command and,
- \* Move through the array, element by element, and set any value that is less than 0.2 to 0 and any value that is greater than or equal to 0.2 to 1.

**Solution:**

Open up a notepad (or Spyder), type in the following text and then save it:

```
import numpy as np
A = np.random.rand(2,3) # A 2x3 array of random numbers
A[A<0.2] = 0
A[A>=0.2] = 1
```

Run the above script a few times and explain what do you observe? \_\_\_\_\_

**Example.** Extract from the array  $B = \text{np.array}([3, 4, 6, 10, 24, 89, 45, 43, 46, 99, 100])$  those numbers

- \* which are not divisible by 3;
- \* which are divisible by 5;
- \* which are divisible by 3 and 5;
- \* which are divisible by 3 and set them to 42.

**Sample Solution:**

```
def is_divisible_by(n): return lambda x: x % n == 0
```

```
udiv3 = np.frompyfunc(is_divisible_by(3),1,1)
print("Not divisible by 3 =>", B[~udiv3(B).astype('bool')])
udiv5 = np.frompyfunc(is_divisible_by(5),1,1)
print("Divisible by 5 =>", B[udiv5(B).astype('bool')])
print("Divisible by 3 and 5 =>", B[udiv3(B).astype('bool') &
```

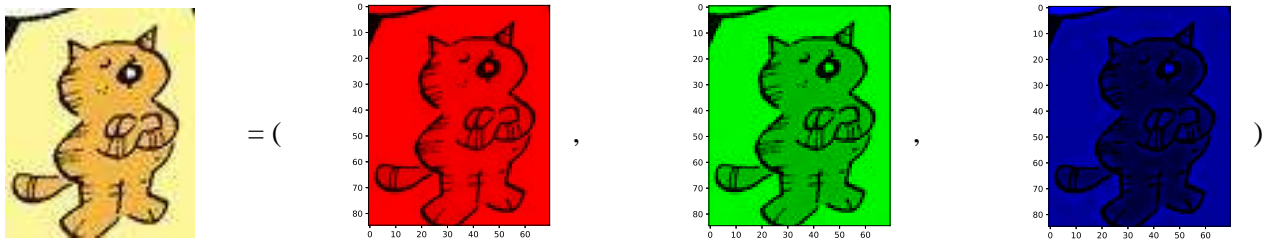
```

udiv5(B).astype('bool')])
B[udiv3(B).astype('bool')] = 42

```

**Example.** (Simple Image Processing) We can regard an array of Booleans,  $M$ , of the same shape as a number array  $A$  like a new layer above the array  $A$ , called a **mask**.

Consider a heathcliff image  $A$  below:



The Boolean indexing can be used to extract the red, green and blue components of a coloured image.

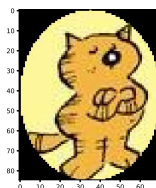
```

from PIL import Image
import numpy as np
import matplotlib.pyplot as plt
# https://liaohaohui.github.io/UECM1703/heathcliff2.jpg
f = Image.open("heathcliff2.jpg")
orig = np.array(f)
fig, (ax0, ax1, ax2) = plt.subplots(nrows=1, ncols=3)
ax0.hist(orig[:, :, 0])
ax0.set_title("Red")
ax1.hist(orig[:, :, 1])
ax1.set_title("Green")
ax2.hist(orig[:, :, 2])
ax2.set_title("Blue")
fig.tight_layout()
plt.show()

fig, (ax0, ax1, ax2) = plt.subplots(nrows=1, ncols=3)
arr = orig.copy()
arr[:, :, 1:] = 0 # red
img = Image.fromarray(arr)
ax0.imshow(img)
arr = orig.copy()
arr[:, :, [0, 2]] = 0 # green
img = Image.fromarray(arr)
ax1.imshow(img)
arr = orig.copy()
arr[:, :, :2] = 0 # blue
img = Image.fromarray(arr)
ax2.imshow(img)
fig.tight_layout()
plt.show()

```

We can also use the Boolean indexing to mask part of the image. For example, we can use it to create a 'elliptic frame' (in black) as follows.



using Boolean indexing (and array grid broadcasting)

```

arr = orig.copy()
# Using np.ogrid allows array indexing and broadcasting for calculation
x, y = np.ogrid[:arr.shape[0],:arr.shape[1]]
centre = np.array(arr.shape)/2
mask_ellip = (x-centre[0])**2/centre[0]**2+(y-centre[1])**2/centre[1]**2>1.0
arr[mask_ellip,:] = 0
plt.imshow(arr)
plt.savefig("heathcliff2_ell.eps")
plt.show()

```

In the practical, try to convert the coloured image 3-D array to gray colour 2-D array using the formula below:

$$gray = 0.2989 \text{ red} + 0.5870 \text{ green} + 0.1140 \text{ blue}$$

The Boolean indexing can also be used in thresholding to select regions the boundaries of a character.

Summary of Numpy Fancy Indexing from [https://lectures.scientific-python.org/intro/numpy/array\\_object.html](https://lectures.scientific-python.org/intro/numpy/array_object.html)

```

>>> a[(0,1,2,3,4), (1,2,3,4,5)]
array([1, 12, 23, 34, 45])

>>> a[3:, [0,2,5]]
array([[30, 32, 35],
       [40, 42, 45],
       [50, 52, 55]])

>>> mask = np.array([1,0,1,0,0,1], dtype=bool)
>>> a[mask, 2]
array([2, 22, 52])

```

0	1	2	3	4	5
10	11	12	13	14	15
20	21	22	23	24	25
30	31	32	33	34	35
40	41	42	43	44	45
50	51	52	53	54	55

Note that in the above example, `a[(0,1,2,3,4), (1,2,3,4,5)]` is the same as `a[[0,1,2,3,4], [1,2,3,4,5]]` and both means `np.array([a[0,1], a[1,2], a[2,3], a[3,4], a[4,5]])`.

## 6. Array Reduction Operations

When we want to work on the elements of arrays along some **axis** or multiple axes, we can regard as **reducing** the array data to some values. We will explore some classes of reduction operations below.

### ‘Reduction’ Operations for Filtering and Construction

- \* `np.choose(J, A)` picks `A[J[i]]` into `J`. It complements `np.compress()`, `np.select()`, `np.extract()`, etc.
- \* `np.putmask(A, mask, values)`: works similar to Boolean masking. Closely related to `np.take()`, `np.place()`, `np.put()`, `np.copyto()`.  
`np.putmask(A, A<0, 0)` # Same as `A[A<0] = 0`
- \* `np.correlate(x, y)` correlates two 1-D arrays

$$z_j = \sum_{i=\max(j-M, 0)}^{\min(j, K)} x_i y_{j+i}, \quad j = 0, \dots, K + M$$

- \* `np.convolve(x, y)` convolves two 1-D arrays

$$z_j = \sum_{i=\max(j-M, 0)}^{\min(j, K)} x_i y_{j-i}, \quad j = 0, \dots, \max(K, M)$$

Here  $x$  and  $y$  are two 1-D arrays with  $K = x.size-1$  and  $M = y.size-1$ . The last two items are used in **signal processing**.

### Ordering ‘Reduction’ Operations

- \* `np.max`, `np.min`: returns the largest value and the smallest value
- \* `ptp`: return the range of values, i.e. the difference of maximum and minimum
- \* `np.argmax`, `np.argmin`: returns the index of the largest value and the smallest value
- \* `sort`: return a sorted copy of an array
- \* `np.argsort`: return indices of sorted array
- \* `searchsorted`: find indices where elements should be inserted to maintain order

### Statistical ‘Reduction’ Operations

- \* `np.sum(X)`: It is used for summation. When  $X$  is the data  $x_1, x_2, \dots, x_n$ , the sum returns

$$x_1 + x_2 + \dots + x_n.$$

Together with array mathematical functions (Ufuncs), the for loop from Topic 1:

$$f(1) + f(2) + f(3) + \dots + f(n)$$

can be written as

$$f(\text{np.r_[1:(n+1)]}).\text{sum}() \quad \# \text{ or } \text{np.sum}(f(\text{np.r_[1:(n+1)]}))$$

Related: `np.cumsum`, `np.prod`, `np.cumprod`, ...

- \* `np.mean(X)`: The mean is  $\bar{X} = \frac{x_1 + \dots + x_n}{n}$ .  
`np.average(X, weights=W)` generalises mean and allows weighted mean.
- \* `np.median(X)`: Find the median of data  $X$ .
- \* `np.var(X)` (and `np.std(X) = \sqrt{\text{var}(X)}`): By default, it is the **population variance** (and standard deviation)

$$\text{Var}[X] = \frac{(x_1 - \bar{X})^2 + \dots + (x_n - \bar{X})^2}{n}.$$

Note that for **sample variance** (and sample population), the  $n$  needs to be changed to  $n-1$  (set `ddof=1`).

- \* `np.cov(X)`: Compute the covariance matrix of data in  $X$  based on the mathematical formulation:

$$\text{Cov}[X] = E[(X - E[X])(X - E[X])^H].$$



## 6.1. Ordering and Statistical ‘Reduction’ Operations (for Statistics)

In this section, we first look at the ordering reduction operations (min, max, ptp, ...) and the statistical reduction operations (sum, prod, mean, var, std, cumsum, ...) for a simple 2D array. We then look at the past year questions which uses reduction operations combining with other array operations in the earlier sections.

**Example.** (General and Statistical ‘Reduction’ Operations) Consider the array

```
M = [[ 6,  9, 12,  4,  3,  0],
      [ 4,  4, 15,  2,  1,  1],
      [ 2,  1, 18, -5,  8,  2],
      [-6, -4, 21,  1, -5,  2]]
```

Let us investigate the reduction operations min, max, range, sum, prod, cumsum, cumprod, mean, var (population variance), std (population standard deviation), etc. along the whole array, along the row and along the column.

**Solution:** Let’s investigate how the axis work with the various given reduction operations.

```
M = np.array([[ 6,  9, 12,  4,  3,  0],
              [ 4,  4, 15,  2,  1,  1],
              [ 2,  1, 18, -5,  8,  2],
              [-6, -4, 21,  1, -5,  2]])

### Reduction operations along the whole array
M.min()          # -6
M.max()          # 21
M.ptp()          # 27
M.sum()          # 96 (all numbers add)
M.prod()         # 0 (all numbers multiply)
M.cumsum()       # 6, 6+9, 6+9+12, ...
M.cumprod()      # 6, 6*9, 6*9*12, ...
M.mean()        # 4.0
M.var()         # 46.25
M.std()         # 6.800735254367722
#
# NOTE: There is a 'nan' version for the above commands which
# skips nan, e.g. np.nanmean(x), np.nanvar(x), etc.
#

### Reduction operations along the rows (axis = 1)
M.min(axis=1)    # [ 0,  1, -5, -6]
M.max(axis=1)    # [12, 15, 18, 21]
M.ptp(axis=1)
M.sum(axis=1)    # It will sum along the row return 1-D array
M.prod(axis=1, keepdims=1) # use keepdims=1 if we want 2-D array
M.cumsum(axis=1) # [6,6+9,...], [4,4+4,...], [2,2+1,...], ...
M.cumprod(axis=1) # [6,6*9,...], [4,4*4,...], [2,2*1,...], ...
M.mean(axis=1)
M.var(axis=1)    # for sample variance, use ddof=1
M.std(axis=1)

### Reduction operations along the columns (axis = 0)
M.min(axis=0)
M.max(axis=0)
M.ptp(axis=0)
M.sum(axis=0, keepdims=1)
M.prod(axis=0)
M.cumsum(axis=0)
M.cumprod(axis=0)
M.mean(axis=0)
M.var(axis=0)
M.std(axis=0)
```

**Example.** (Final Exam Sept 2015, Q1(b)(i)) Write down and explain the values of C for the following commands

```
import numpy as np
A = np.array([4,6,8],dtype='double')
B = np.array([2,0,4])
C = np.sum(A/B)
```

**Solution:**  $C = \text{sum}([4/2 \ 6/0 \ 8/4]) = \text{sum}([2 \ \text{Inf} \ 2]) = \text{Inf}$

**Lessons learned:** Be careful about the division by zero. We may get infinity.

**Example.** (Final Exam Oct 2018, Q1(b)) The dot product of two vectors  $\mathbf{x} = (x_1, x_2, \dots, x_n)$  and  $\mathbf{y} = (y_1, y_2, \dots, y_n)$ ,  $\mathbf{x} \cdot \mathbf{y}$ , is defined as

$$\mathbf{x} \cdot \mathbf{y} = x_1 y_1 + x_2 y_2 + \dots + x_n y_n.$$

The angle between two arrays is defined by the following relation

$$\cos = \frac{\mathbf{x} \cdot \mathbf{y}}{\sqrt{\mathbf{x} \cdot \mathbf{x}} \sqrt{\mathbf{y} \cdot \mathbf{y}}}.$$

Implement a **Python function** theta to calculate the angle (in **degree**) if you are given two arrays  $\mathbf{a}=[a_1, a_2, a_3, a_4]$  and  $\mathbf{b}=[b_1, b_2, b_3, b_4]$ . You **must** write down the proper import statements. If you use the Numpy module, you must prefix the Numpy functions with “np.” or marks will be heavily deducted. Use scientific calculator to find the return value of the Python command `theta([1,2,3,4],[2,1,3,4])` to 4 decimal places.

**Sample Solution:** In 2018, Numpy is not as advanced as now and the lecture has not mastered `np.dot()`:

```
# Appropriate import when writing scripts 1 mark
import numpy as np
from math import degrees, acos, sqrt
# Able to define a function and return a value: 1.5 marks
# Correct translation of mathematical formula to Python: 2.5 marks
def theta(x, y):
    # We are using the single-line for loop from Topic 1
    # The size of x and y may be different, so we need to
    # check it in real-world programming but it is fine
    # to assume x and y the same size in exam
    num = sum(x[i]*y[i] for i in range(x.size))
    sxx = sqrt(sum(x[i]*x[i] for i in range(x.size)))
    syy = sqrt(sum(y[i]*y[i] for i in range(y.size)))
    return degrees(acos(num/sxx/syy))
```

```
print(theta(np.array([1,2,3,4]),np.array([2,1,3,4])))
```

The question also test the understanding of manual calculation and the use of calculator:

$$\begin{aligned} &= \cos^{-1} \frac{2+2+9+16}{\sqrt{1^2+2^2+3^2+4^2}} \times \frac{1}{\sqrt{2^2+1^2+3^2+4^2}} & [1 \text{ mark}] \\ &= \cos^{-1} \frac{29}{30} = 0.258922 \times \frac{180^\circ}{1} = 14.8351^\circ & [2+1=3 \text{ marks}] \end{aligned}$$

**Sample Solution:** Since 2020, Numpy becomes advanced with the introduction of @ for `np.matmul`, a simple answer is given below.

```
# Appropriate import when writing scripts 1 mark
import numpy as np
def theta(x, y):
    return np.degrees(np.arccos(x@y/np.sqrt(x@x)/np.sqrt(y@y)))

print(theta(np.array([1,2,3,4]),np.array([2,1,3,4])))
```

**Example.** (Final Exam Oct 2018, Q1(a), CO1) The output of the Python commands below

```
>>> import numpy as np
>>> A = np.arange(1,36).astype('float').reshape(5,7)
>>> print(A)
```

is

```
[[ 1.  2.  3.  4.  5.  6.  7.]
 [ 8.  9. 10. 11. 12. 13. 14.]
 [15. 16. 17. 18. 19. 20. 21.]
 [22. 23. 24. 25. 26. 27. 28.]
 [29. 30. 31. 32. 33. 34. 35.]]
```

Use the above information to write down the output to the following Python commands for item (i) to item (iv).

(i) `print(A[2,:])`

**Solution:** `[ 15. 16. 17. 18. 19. 20. 21.]`

(ii) `print(A[1:4,3:5])`

**Solution:**

```
[[ 11.  12.]
 [ 18.  19.]
 [ 25.  26.]]
```

(iii) `print(A[2:4,4:6].mean())`

**Solution:** `A[2:4,4:6]` `[[ 19., 20.], [ 26., 27.]]`  

$$\frac{19+20+26+27}{4} = 23.0$$

(iv) `print(A[:,2]>10)`

**Solution:** `[False False True True True]`

(v) Write down the Python command to **count** the number of elements in A who are larger than 20.

**Solution:** `(A>20).sum()`

**Example.**

(a) Write down the Python command to subtract the each column of a matrix A by the mean of the data of each column vector.

`A - A.mean(axis=0)`

(b) The 'shortest' Python command to subtract the each row of a matrix A by the mean of the data of each row vector is probably

`(A.T - A.mean(1)).T`

Do you know other slightly longer Python command which achieves the same outcome for A?

`A - A.mean(axis=1, keepdims=True)`

`A - A.mean(1).reshape(-1,1)`

**Example.** (Final Exam Sept 2019, Q1)

(a) Given

```
A = [[ 4  9  8  0  2  9 ]
      [ 3  1  9  1  2  8 ]
      [ 7  2  2  3  7  8 ]
      [ 3  5  0  7  9  7 ]
      [ 9  4  6  7  9  8 ]]
```

Use the above information to **execute** the following Python commands for item (i) to item (iv) and write down the output of the execution.

(i) `print(A[:,1])`

**Solution:** `[9 1 2 5 4]`

(ii) `print(A[1:4,[1,2,3]])`

**Solution:**

```
[[1 9 1]
 [2 2 3]
 [5 0 7]]
```

(iii) `print(A[A<5].sum())`

**Solution:**  $(4 + 0 + 2) + (3 + 1 + 1 + 2) + (2 + 2 + 3) + (3 + 0) + 4 = 27$

(iv) `print(A[:4,:3].sum(axis=0))`

**Solution:**  $[4 + 3 + 7 + 3, 9 + 1 + 2 + 5, 8 + 9 + 2 + 0] = [17, 17, 19]$

(v) Write down the Python command which gives the mean of rows in A after **execution**.

**Solution:** `print(A.mean(axis=1))`

(vi) Write down the warning message that the command

`print(A[1,:]/A[0,:].astype(np.float64))` will raise when it is executed.

**Solution:** Since  $A[0,3]$  is zero, a division by zero error will be produced.

(b) Use Numpy array operations such as `np.arange`, etc. to write a computer program in no more than 3 lines and without using any semicolon to print the following output:

```
1      1      1      1
2      4      8     16
3      9     27     81
4     16     64    256
5     25    125   625
6     36    216  1296
7     49    343  2401
8     64    512  4096
9     81    729  6561
10    100   1000 10000
```

**Sample Solution:**

```
coll = np.arange(1,11).reshape(10,1)           # [1.5 marks]
B = np.hstack((coll, coll**2, coll**3, coll**4)) # [2 marks]
print(B)                                         # [0.5 mark ]
```

**Example.** (Write program script using Numpy array) The following function is used to generate a moving sequence (with a particular window size, by default 4) for a one-dimensional array a.

```
def rolling(a, window=4):
    n = a.size
    newarray = np.zeros((n-window+1,window))
    for i in range(n-window+1):
        newarray[i,:] = a[i:i+window]
    return newarray
```

When the one-dimensional array is  $\mathbf{x} = [x_1, x_2, x_3, x_4, \dots, x_n]$ , the return moving sequence of `rolling(x)` with a window size 4 is

$$[[x_1, x_2, x_3, x_4], [x_2, x_3, x_4, x_5], \dots, [x_{n-3}, x_{n-2}, x_{n-1}, x_n]].$$

(i) If `a = np.array([1.1, 1.34, 1.17, 1.06, 1.06, 0.94])`, write down the output of Python command `rolling(a)`.

**Solution:**

```
array([[1.1 , 1.34, 1.17, 1.06],
       [1.34, 1.17, 1.06, 1.06],
       [1.17, 1.06, 1.06, 0.94]])
```

(ii) Define a Python function `moving_average` to calculate moving average of  $\mathbf{x}$  with a window of 4 which returns the following array:

$$\left[ \frac{x_1 + x_2 + x_3 + x_4}{4}, \frac{x_2 + x_3 + x_4 + x_5}{4}, \dots, \frac{x_{n-3} + x_{n-2} + x_{n-1} + x_n}{4} \right]$$

based on the moving sequence `rolling(x)`. Write down the output of the Python command `print(moving_average(a))` where a is given in part (i).

**Solution:** By using Numpy array method, we have

```
def moving_average(x,w=4): return rolling(x,w).mean(axis=1)
```

Alternatively, a less elegant method is to be used for loop:

```
def moving_average(x,w=4):
    retval = np.zeros(x.size-w+1)
    data = rolling(x,w)
    for i in range(retval.size):
        retval[i] = np.mean(data[i])
    return retval
```

The output of `moving_average(a)` is `[1.1675, 1.1575, 1.0575]`

- (iii) Explain how to calculate moving variance of `a` in part (i) with a window of 4.

**Solution:** `rolling(a).var(axis=1)`

**Example.** (Final Exam Oct 2018, Q2(b), CO3) The following function from a program script is used to generate a moving sequence for a one-dimensional array

```
def rolling(a, window=4):
    n = a.size
    newarray = np.zeros((n-window+1,window))
    for i in range(n-window+1):
        newarray[i,:] = a[i:i+window]
    return newarray
```

When the one-dimensional array is  $\mathbf{x} = [x_1, x_2, x_3, x_4, \dots, x_n]$ , the return moving sequence of `rolling(x)` is

$$[[x_1, x_2, x_3, x_4], [x_2, x_3, x_4, x_5], \dots, [x_{n-3}, x_{n-2}, x_{n-1}, x_n]].$$

- (i) If `a = np.array([0.95, 0.87, 0.87, 0.98, 1.04, 1.08])`, write down the output of `rolling(a)`.

**Solution:**

```
array([[0.95, 0.87, 0.87, 0.98],
       [0.87, 0.87, 0.98, 1.04],
       [0.87, 0.98, 1.04, 1.08]])
```

- (ii) Define a Python function `moving_average` to calculate moving average of  $\mathbf{x}$  with a window of 4 which returns the following array:

$$\left[ \frac{x_1 + x_2 + x_3 + x_4}{4}, \frac{x_2 + x_3 + x_4 + x_5}{4}, \dots, \frac{x_{n-3} + x_{n-2} + x_{n-1} + x_n}{4} \right]$$

based on the moving sequence `rolling(x)`. Write down the output of the Python command `print(moving_average(a))` where `a` is given in part (i).

**Sample Solution:** By using Numpy array method, we have

```
def moving_average(x,w=4):
    return rolling(x,w).mean(axis=1)
```

Alternatively, a less elegant method is to use the for loop:

```
def moving_average(x,w=4):
    retval = np.zeros(x.size-w+1)
    data = rolling(x,w)
    for i in range(retval.size):
        retval[i] = np.mean(data[i])
    return retval
```

The output of `moving_average(a)` is `[0.9175, 0.94, 0.9925]`.

- (iii) Explain how to calculate moving variance of `a` in part (i) with a window of 4.

**Solution:** `rolling(a).var(axis=1)`

The following example is taken from Scipy just to illustrate how real-world Python program is written using Numpy.

**Example.** ((Final Exam Oct 2023, Magic Square using Integer Array)) Using numpy integer array, a magic square can be made simpler. The Python function to check if an object is a magic square can be simplified as follows.

```

def is_magic_square(arr):
    sums_from_every_row = arr.sum(axis=1)
    sums_from_every_col = arr.sum(axis=0)
    #Two diagonals
    diag1 = np.diag(arr).sum()      # Using reduction 'sum'
    diag2 = np.diag(arr[::-1]).sum()
    return diag1==diag2 and \
        np.all(sums_from_every_row == diag1) and \
        np.all(sums_from_every_col == diag1)

m=[[7, 12, 1, 14], [2, 13, 8, 11], [16, 3, 10, 5], [9, 6, 15, 4]]
print(is_magic_square(np.array(m)))
print(is_magic_square(np.array([[2, 7, 6], [9, 5, 1], [4, 3, 8]])))
print(is_magic_square(np.array([[2, 7, 6], [9, 5, 1], [4, 3, 7]])))

```

The generation of magic square algorithm from Topic 1 can be simplified as follows.

```

def magic_sqr_method1(n):
    if n % 2 == 0: return None # Only works with odd n
    magic_square = np.zeros((n,n))
    cnt, i, j = 1, 0, n//2
    while cnt <= n**2:
        magic_square[i,j] = cnt
        cnt += 1
        newi, newj = (i-1)%n, (j+1)%n
        if magic_square[newi,newj]:
            i += 1
        else:
            i, j = newi, newj
    return magic_square

```

---

**Example.** (scipy.stats.gmean) The geometric mean implementation in Scipy is listed below.

```

def gmean(a, axis=0, dtype=None, weights=None):
    r"""Compute the weighted geometric mean along the specified axis.

```

The weighted geometric mean of the array :math:`a\_i` associated to weights :math:`w\_i` is:

$$\exp\left(\frac{\sum_{i=1}^n w_i \ln a_i}{\sum_{i=1}^n w_i}\right)$$

and, with equal weights, it gives:

$$\sqrt[n]{\prod_{i=1}^n a_i}.$$

Parameters

-----

**a** : array\_like

Input array or object that can be converted to an array.

**axis** : int or None, optional

Axis along which the geometric mean is computed. Default is 0.

If None, compute over the whole array 'a'.

**dtype** : dtype, optional

Type to which the input arrays are cast before the calculation is performed.

**weights** : array\_like, optional

The 'weights' array must be broadcastable to the same shape as 'a'.

Default is None, which gives each value a weight of 1.0.

Returns

```

-----
gmean : ndarray
    See 'dtype' parameter above.

See Also
-----
numpy.mean : Arithmetic average
numpy.average : Weighted average
hmean : Harmonic mean

References
-----
.. [1] "Weighted Geometric Mean", *Wikipedia*,
    https://en.wikipedia.org/wiki/Weighted_geometric_mean.
.. [2] Grossman, J., Grossman, M., Katz, R., "Averages: A New Approach",
    Archimedes Foundation, 1983

Examples
-----
>>> from scipy.stats import gmean
>>> gmean([1, 4])
2.0
>>> gmean([1, 2, 3, 4, 5, 6, 7])
3.3800151591412964
>>> gmean([1, 4, 7], weights=[3, 1, 3])
2.80668351922014

"""

a = np.asarray(a, dtype=dtype)

if weights is not None:
    weights = np.asarray(weights, dtype=dtype)

with np.errstate(divide='ignore'):
    log_a = np.log(a)

return np.exp(np.average(log_a, axis=axis, weights=weights))

```

Note the programming techniques used:

- \* Function with default values
  - \* Defining 'help documentation' for a function using `""" ... """`.
  - \* If statement
  - \* Array mathematical functions and reduction operations.
-

## 7. Geometric Operations

In this section, we introduce operations which are related to geometry such as length (and the generalisation, norm), angle, matrix product (related to linear transformations), etc. They can be expressed as the reduction operations mentioned earlier.

### 7.1. Geometric Operations for Vectors

The **length** of a vector

$$|x| = \sqrt{x_1^2 + x_2^2 + \cdots + x_n^2}$$

can be obtained using `scipy.linalg.norm(x[, ord, axis, keepdims])`.

The **angle** between a vector  $x$  and a vector  $y$  (in radian) is given by the **dot-product**:

$$x \cdot y = x_1y_1 + x_2y_2 + \cdots + x_ny_n.$$

The Numpy operations which allow us to perform dot product on  $x$  and  $y$  include `x@y`, `np.matmul(x, y)`, `np.dot(x, y)` and `np.vdot(x, y)`.

**Example.** (Geometric Operations on Vectors) Find lengths and angle the vectors  $x$  and  $y$ :

```
x = np.array([3, -1, 2, -4])
y = np.array([5, 7, 3, 1])
```

**Sample Solution:**

```
import numpy as np
from scipy import linalg
xlength = linalg.norm(x) # earlier: np.sqrt(x@x)
ylength = linalg.norm(y) # earlier: np.sqrt(y@y)
angle_x_y = np.arccos((x@y)/xlength/ylength)
```

**Example.** (Final Exam Oct 2018, Q1(b)) The dot product of two vectors  $\mathbf{x} = (x_1, x_2, \dots, x_n)$  and  $\mathbf{y} = (y_1, y_2, \dots, y_n)$ ,  $\mathbf{x} \cdot \mathbf{y}$ , is defined as

$$\mathbf{x} \cdot \mathbf{y} = x_1y_1 + x_2y_2 + \cdots + x_ny_n = |\mathbf{x}||\mathbf{y}|\cos \theta.$$

The angle  $\theta$  between two arrays is defined by the following relation

$$\cos \theta = \frac{\mathbf{x} \cdot \mathbf{y}}{\|\mathbf{x}\| \|\mathbf{y}\|}.$$

Implement a **Python function** `theta` to calculate the angle  $\theta$  (in **degree**) if you are given two arrays  $\mathbf{a}=[a_1, a_2, a_3, a_4]$  and  $\mathbf{b}=[b_1, b_2, b_3, b_4]$ . You **must** write down the proper import statements. If you use the Numpy module, you must prefix the Numpy functions with “`np.`” or marks will be heavily deducted. Use scientific calculator to find the return value of the Python command `theta([1, 2, 3, 4], [2, 1, 3, 4])` to 4 decimal places.

Try to work out the answer using dot product and vector norms.

**Sample Solution:**

```
import numpy as np
from scipy import linalg # [1 mark]
def theta(x, y): # [1 mark]
    num = np.array(x).dot(y) # [1 mark]
    den = linalg.norm(x)*linalg.norm(y) # [1 mark]
    return np.degrees(np.arccos(num/den)) # [1 mark]
```



## 7.2. Operations for Matrices

Apart from the elementwise arithmetic mentioned earlier, the following are some operations specific to matrices:

- \* `A@B` or `np.matmul(A, B)`: Matrix product of two arrays.
- \* `np.dot(A, B)`: Dot product  $z[I, J, j] = \sum_k A[I, k]B[J, k, j]$  of A and B. It is equivalent to matrix multiplication.
- \* `np.linalg.matrix_power(A, n)`: Raise a square matrix to the (integer) power n.
- \* `np.kron(A, B)`: Kronecker product of two arrays, giving  $[a_{ij...k}B]$ .

**Example.** (Linear Algebra Operations on 2-D arrays) Consider the matrices

$$A = \begin{bmatrix} 1 & 2 \\ 3 & 4 \\ 5 & 6 \end{bmatrix} \quad B = \begin{bmatrix} 7 & 8 \\ 8 & 7 \end{bmatrix} \quad C = \begin{bmatrix} 3 & -2 \end{bmatrix}$$

Find the matrix product  $AB$ , the ‘matrix product’  $BC$ , the matrix power  $B^4$  and the Kronecker product  $A \otimes B$  using Python.

**Sample Solution:**

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

A @ B or np.matmul(A, B) or np.dot(A, B)
#      [23, 22]
# A x B = [53, 52]
#      [83, 82]

# Note: Python will cleverly regard C as column matrix
# when B (2x2 matrix) is multiplied to C
B @ C # [ 5, 10]

B @ B @ B @ B or np.linalg.matrix_power(B, 4)
#      [ 25313  25312 ]
# B^4 = B x B x B x B = [      ]
#      [ 25312  25313 ]

np.kron(A, B)
#      [ 7,  8, 14, 16]
#      [ 8,  7, 16, 14]
# Kronecker product A (x) B = [ 1B, 2B ] = [21, 24, 28, 32]
#      [ 3B, 4B ] = [24, 21, 32, 28]
#      [ 5B, 6B ] = [35, 40, 42, 48]
#      [40, 35, 48, 42]
```

**Example.** Explain which of the Python/Numpy instruction is most appropriate achieved the following results.

- (a) Generate the multiplication tables for 1 to 9 using Python’s linear algebra operations.

**Sample Solution:** We can use the elementwise arithmetic to achieve this:

```
np.r_[1:10].reshape((9,-1)) * np.r_[1:10] # or
np.kron(np.r_[1:10].reshape((9,-1)), np.r_[1:10])
```

- (b) Let  $A = \begin{bmatrix} 2 & 1 \\ 1 & 2 \end{bmatrix}$ . Use A to generate a matrix like this

$$\begin{bmatrix} 6 & 8 & 10 & 3 & 4 & 5 \\ 10 & 8 & 6 & 5 & 4 & 3 \\ 3 & 4 & 5 & 6 & 8 & 10 \\ 5 & 4 & 3 & 10 & 8 & 6 \end{bmatrix}$$

**Sample Solution:** There are many answers to this but requires us to observe that

$$\begin{bmatrix} 2 \begin{bmatrix} 3 & 4 & 5 \\ 5 & 4 & 3 \\ 3 & 4 & 5 \\ 5 & 4 & 3 \end{bmatrix} \\ 1 \begin{bmatrix} 3 & 4 & 5 \\ 5 & 4 & 3 \\ 3 & 4 & 5 \\ 5 & 4 & 3 \end{bmatrix} \end{bmatrix}$$

```
A = np.array([[2,1],[1,2]])
B = np.array([[3,4,5],[5,4,3]])
np.kron(A,B)
```

**Example.** (Final Exam Sept 2019, Q2) Given the matrix

$$M = \begin{bmatrix} 1 & 2 \\ 3 & 4 \\ 5 & 6 \end{bmatrix}$$

(a) **Execute** the following Python commands for item (i) to item (v) and write down the output of the execution.

(i) `print(M * M)`

**Solution:**

```
1 4
9 16
25 36
```

(ii) `print(M @ M.T)`

**Solution:**

```
5 11 17
11 25 39
17 39 61
```

(iii) `print(M[[2,1,0,1,2],:][:,[1,0,0,1]])`

**Solution:**

<code>M[[2,1,0,1,2],:]</code>		<code>M[[2,1,0,1,2],:][:,[1,0,0,1]]</code>
<pre>[ 5  6 ]</pre>		<pre>[ 6  5  5  6 ]</pre>
<pre>[ 3  4 ]</pre>		<pre>[ 4  3  3  4 ]</pre>
<pre>[ 1  2 ]</pre>	----->	<pre>[ 2  1  1  2 ]</pre>
<pre>[ 3  4 ]</pre>		<pre>[ 4  3  3  4 ]</pre>
<pre>[ 5  6 ]</pre>		<pre>[ 6  5  5  6 ]</pre>

(iv) `print(M[:2,:]==M[[2,1],:])`

**Solution:**

<pre>[ 1  2 ]</pre>	<pre>[ 5  6 ]</pre>	<pre>[ False False ]</pre>
<pre>[ 3  4 ]</pre>	<pre>[ 3  4 ]</pre>	<pre>[ True  True ]</pre>

(v) `print((M<3)|(M>4))`

**Solution:**

<pre>[ 1&lt;3  2&lt;3 ]</pre>	<pre>[ 1&gt;4  2&gt;4 ]</pre>	<pre>[ True  True ]</pre>
<pre>[ 3&lt;3  4&lt;3 ]</pre>	<pre>[ 3&gt;4  4&gt;4 ]</pre>	<pre>[ False False ]</pre>
<pre>[ 5&lt;3  6&lt;3 ]</pre>	<pre>[ 5&gt;4  6&gt;4 ]</pre>	<pre>[ True  True ]</pre>

(b) Write a Python program with no more than 3 lines to produce the following matrices from M:

$$M\_1 = \begin{bmatrix} -2.5 & -1.5 \\ -0.5 & 0.5 \\ 1.5 & 2.5 \end{bmatrix} \quad M\_2 = \begin{bmatrix} -2 & -2 \\ 0 & 0 \\ 2 & 2 \end{bmatrix} \quad M\_3 = \begin{bmatrix} -0.5 & 0.5 \\ -0.5 & 0.5 \\ -0.5 & 0.5 \end{bmatrix}$$

by using the Numpy vector operation in the Python computer software. Note that  $M\_1$  is  $M$  subtracted by the mean of all values in  $M$ ,  $M\_2$  is a matrix such that each column in  $M$  being subtracted by the mean of corresponding column,  $M\_3$  is a matrix such that each row in  $M$  being subtracted by the mean of corresponding row. Note that your program must work when  $M$  is changed to an arbitrary  $m \times n$  matrix.

**Sample Solution:**

```
M1 = M - M.mean()           # [1 mark ]
M2 = M - M.mean(axis=0))    # [2 marks]
M3 = M - M.mean(axis=1,keepdims=True) # [2 marks]
```

## 8. Linear Algebra Solvers (To be covered in Topic 5)

In science and engineering, we often encounter the equations involving matrices called the **linear system** or the **linear algebra problem**:

$$AX = B \quad (\text{LS})$$

where  $A$  is an  $m \times n$  matrix,  $X$  is an  $n \times k$  matrix and  $B$  is an  $m \times k$  matrix.  $X$  is unknown whereas  $A$  and  $B$  need to be given.

### 8.1. When $m = n$ and $A$ is invertible

(LS) can be solved using

```
from scipy import linalg
X = linalg.solve(A, B) # linalg.inv(A) @ B is not recommended
```

The `linalg.solve` should be able to solve (LS) with  $m = n \ll 10^4$  using the Gaussian elimination method (and Cholesky method when the matrix  $A$  is positive definite). For any larger matrix, we may need the sparse matrix solvers.

**Example.** Write down the Python script to solve the following problem:

$$\begin{bmatrix} -2 & 11 \\ 17 & -19 \end{bmatrix} \begin{bmatrix} x_{11} & x_{12} \\ x_{21} & x_{22} \end{bmatrix} = \begin{bmatrix} 19 & 1 \\ 3 & 2 \end{bmatrix}$$

**Sample Solution:**

```
from scipy import linalg # Mentioned earlier
A = np.array([[ -2, 11], [17, -19]])
B = np.array([[19, 1], [3, 2]])
X = linalg.solve(A, B)
print("X=", X)
```

The solution is

$$X = \begin{bmatrix} 2.64429530 & 0.27516779 \\ 2.20805369 & 0.14093960 \end{bmatrix}$$

---

**Example.** (Final Exam Sept 2014, Q4(a)) Given the linear system

$$3x_1 + 7x_2 - 2x_3 + 3x_4 - x_5 = 37$$

$$4x_1 + 3x_5 = 40$$

$$5x_3 - 4x_4 + x_5 = 12$$

$$2x_1 + 9x_3 + 4x_4 + 3x_5 = 14$$

$$5x_4 + 8x_5 = 20$$

Write a Python script to solve the linear system.

**Sample Solution:**

```
import numpy as np
from scipy import linalg
A = np.array([[3, 7, -2, 3, -1],
              [4, 0, 0, 0, 3],
              [0, 0, 5, -4, 1],
              [2, 0, 9, 4, 3],
              [0, 0, 0, 5, 8]])
x = linalg.solve(A, [37, 40, 12, 14, 20])
```

---

**Example.** (Final Exam Sept 2020 during MCO, Q1)

(a) Given that A stores the following matrix

```
[ 4 0 0 0 0 15 8 1 0 0 ]
[ 0 6 0 0 0 6 24 6 1 0 ]
[ 0 0 6 0 0 1 8 15 4 4 ]
[ 1 0 0 3 0 0 8 4 18 5 ]
[ 2 3 0 0 5 0 0 8 6 24 ]
[ 29 3 1 0 0 5 0 0 3 7 ]
[ 3 17 5 6 0 0 7 0 0 2 ]
[ 4 4 17 3 4 0 0 6 0 0 ]
[ 0 8 2 25 4 0 0 0 8 0 ]
[ 0 0 7 1 16 0 0 0 0 7 ]
```

- (i) Write down the output of the Python command `A[:, [3, 5, 2, 4]]`. Determine if it is the same as `A[[3, 5, 2, 4]]` and explain the difference.

**Solution:**

```
[ [ 0 15 0 0 ]
  [ 0 6 0 0 ]
  [ 0 1 6 0 ]
  [ 3 0 0 0 ]
  [ 0 0 0 5 ]
  [ 0 5 1 0 ]
  [ 6 0 5 0 ]
  [ 3 0 17 4 ]
  [ 25 0 2 4 ]
  [ 1 0 7 16 ] ]
```

`A[:, [3, 5, 2, 4]]` and `A[[3, 5, 2, 4]]` are different because the former picks the columns while the later pick the rows.

- (ii) Write the Python command to pick all the odd rows and even columns from A and write down the output of your command.

**Solution:** `A[::2, 1::2]` -> 

```
[ [ 0 0 0 24 1 ]
  [ 1 0 0 8 18 ]
  [ 29 1 0 0 3 ]
  [ 4 17 4 0 0 ]
  [ 0 7 16 0 0 ] ]
```

- (iii) Write the Python command to pick the intersection of the second, fifth, third columns and of the eighth, fifth and seventh rows in the given order and write down the output of your command.

**Solution:** `A[:, [1, 4, 2]][[7, 4, 6], :]` -> 

```
[ [ 4 4 17 ]
  [ 3 5 0 ]
  [ 17 0 5 ] ]
```

- (iv) Write the Python command to arrange the given matrix A into the following diagonally dominant form:

```
[ 15 8 1 0 0 4 0 0 0 0 ]
[ 6 24 6 1 0 0 6 0 0 0 ]
[ 1 8 15 4 4 0 0 6 0 0 ]
[ 0 8 4 18 5 1 0 0 3 0 ]
[ 0 0 8 6 24 2 3 0 0 5 ]
[ 5 0 0 3 7 29 3 1 0 0 ]
[ 0 7 0 0 2 3 17 5 6 0 ]
[ 0 0 6 0 0 4 4 17 3 4 ]
[ 0 0 0 8 0 0 8 2 25 4 ]
[ 0 0 0 0 7 0 0 7 1 16 ]
```

**Solution:** `A[:, [5, 6, 7, 8, 9, 0, 1, 2, 3, 4]]`

- (v) For an  $n \times n$  matrix A, it is said to be *diagonally dominant* if for each row the absolute value of the diagonal element is larger than the sum of the absolute value of the rest of the elements in the row:

$$|a_{ii}| > \sum_{j=1, j \neq i}^n |a_{ij}|, \quad i = 1, 2, \dots, n.$$

Write a Python function `is_diag_domin(A)` which determines whether the matrix  $A$  is diagonally dominant. The function will return True if the matrix  $A$  is diagonally dominant, False if the matrix  $A$  is not diagonally dominant, and None if the matrix is not square.

**Sample Solution:**

```
def is_diag_domin(A):
    N = A.shape[0]
    for i in range(N):
        S = sum(abs(A[i,j]) for j in range(N) if j != i)
        if abs(A[i,i]) <= S:
            print("i=", i)
            return False
    return True
```

```
#import q1
#print(is_diag_domin(q1.AA))
```

(b) Given that three  $3 \times 3$  matrices

$$P = \begin{bmatrix} 5 & 8 & 8 \\ 6 & -9 & -8 \\ 6 & -5 & 1 \end{bmatrix} \quad Q = \begin{bmatrix} 2 & 2 & -2 \\ 7 & 8 & -2 \\ 0 & 2 & 2 \end{bmatrix} \quad R = \begin{bmatrix} -2 & -8 & 8 \\ -8 & -5 & 8 \\ 6 & -9 & 4 \end{bmatrix}$$

(i) Write down the Python command to find the inverse matrix of  $Q$ ,  $Q^{-1}$ .

**Solution:** The Python command to find  $Q^{-1}$  is

```
np.linalg.inv(Q)
```

or

```
np.linalg.solve(Q, np.eye(Q.shape[0]))
```

The output is

$$\begin{bmatrix} -1.25 & 0.5 & -0.75 \\ 0.875 & -0.25 & 0.625 \\ -0.875 & 0.25 & -0.125 \end{bmatrix}$$

(ii) Write down the Python command to find matrix  $L$  if  $P^3 L Q = R$ . Write down the **matrix**  $L$ .

**Solution:**  $L = (P^3)^{-1} R Q^{-1}$

```
L = linalg.inv(P@P@P) @ R @ linalg.inv(Q)
```

```
L = linalg.solve(np.linalg.matrix_power(P,3), R)@linalg.inv(Q)
```

The matrix  $L$  is

$$\begin{bmatrix} 0.08603119 & -0.04214438 & 0.07957573 \\ 0.21360577 & -0.09753974 & 0.18129806 \\ -0.24895274 & 0.11235113 & -0.20818642 \end{bmatrix}$$

(iii) Suppose the  $3 \times 3$  matrices  $E, F, G, H$  satisfies

$$\begin{bmatrix} P & Q \\ Q & R \end{bmatrix}^{-1} = \begin{bmatrix} E & F \\ G & H \end{bmatrix}$$

First, find the matrix  $H$  by writing down the appropriate Python commands. Then, write down the appropriate Python command(s) to show that

$$(R - QP^{-1}Q)^{-1} = H.$$

**Solution:** After from `scipy import linalg`, the command

```
H = linalg.inv(R - Q@linalg.inv(P)@Q)
```

allows us to obtain

$$H = \begin{bmatrix} 0.26819736 & -0.15480007 & -0.07891041 \\ 0.69421553 & -0.3532685 & -0.42828374 \\ 1.00692917 & -0.48176952 & -0.51330189 \end{bmatrix}$$


---

## 8.2. When $m \neq n$ or $A$ is not invertible

Mathematicians have solved the general linear system (LS) with no restrictions (except that that they cannot be too large because computer memory is limited) on  $m$  and  $n$  (the price to pay is a longer computation time) using the SVD method or QR method leading to the following functions in Python:

```
from scipy import linalg
X = linalg.lstsq(A, B) # linalg.pinv(A) @ B is not recommended
```

Note that  $X$  may not be a solution but a ‘least square solution’ of the linear system (LS).

**Example.** Write down the Python script to solve the following problem:

$$\begin{bmatrix} -2 & 11 \end{bmatrix} \begin{bmatrix} x_{11} & x_{12} \\ x_{21} & x_{22} \end{bmatrix} = \begin{bmatrix} 19 & 1 \end{bmatrix}$$

**Sample Solution:**

```
A = np.array([[ -2, 11]])
B = np.array([[19, 1]])
X, _, Rank, Sing = linalg.lstsq(A, B)
print("X=", X) # Many solutions but only one return
```

---

**Example.** Solve the Least Square Problem:

$$\begin{bmatrix} -2 & 11 \\ 17 & -19 \\ 6 & 6 \end{bmatrix} \begin{bmatrix} x_{11} & x_{12} \\ x_{21} & x_{22} \end{bmatrix} = \begin{bmatrix} 19 & 1 \\ 3 & 2 \\ 6 & 7 \end{bmatrix}$$

Note: Be careful, `linalg.solve` will not work.

**Sample Solution:**

```
A = np.array([[ -2, 11], [17, -19], [6, 6]])
B = np.array([[19, 1], [3, 2], [6, 7]])
X, Err, Rank, Sing = linalg.lstsq(A, B)
print("X=", X)
print("Residue=", Err)
```

---

### 8.3. Special (Dense) Matrices and Sparse Matrices

For some linear system with special square matrices such as the **Toeplitz matrix**:

$$\begin{bmatrix} a_1 & b_1 & b_2 & \cdots & b_{n-1} & b_n \\ a_2 & a_1 & b_1 & \cdots & b_{n-2} & b_{n-1} \\ a_3 & a_2 & a_0 & \cdots & . & . \\ . & . & . & . & . & . \\ . & . & . & . & a_0 & b_1 \\ a_n & a_{n-1} & a_{n-2} & \cdots & a_1 & a_0 \end{bmatrix}$$

It can be generated using

```
linalg.toeplitz([a1, a2, a3, ..., an], [b0, b1, b2, ..., bm])
```

Mathematicians have developed special algorithms to speed up the solution of linear system with special matrices.

```
* linalg.solve_toeplitz(c_or_cr, b, check_finite=True)
```

```
* Other special cases are ignored.
```

**Example.** Construct a Toeplitz matrix from the 1-D arrays  $a=[2,3,4,5]$  and  $b=[500,6,7,8,9,10]$ .

**Sample Solution:**

```
>>> print(linalg.toeplitz([2,3,4,5],[500,6,7,8,9,10]))
[2, 6, 7, 8, 9, 10]
[3, 2, 6, 7, 8, 9]
[4, 3, 2, 6, 7, 8]
[5, 4, 3, 2, 6, 7]
```

---

**Example.** (Toeplitz System) Write a script using `linalg.toeplitz` to solve the linear system:

$$\begin{bmatrix} 1 & -1 & -2 & -3 \\ 3 & 1 & -1 & -2 \\ 6 & 3 & 1 & -1 \\ 10 & 6 & 3 & 1 \end{bmatrix} y = \begin{bmatrix} 1 \\ 2 \\ 2 \\ 5 \end{bmatrix}$$

**Sample Solution:** When the Toeplitz matrix `linalg.toeplitz(c, r)` is characterised by  $c$  and  $r$ , we can write down the following commands.

```
c = np.array([1,3,6,10]) # first column of left matrix
r = np.array([1,-1,-2,-3]) # first row of left matrix
b = np.array([1,2,2,5]) # right column matrix
x = linalg.solve_toeplitz((c, r), b)
```

---

Some square matrices like the tridiagonal matrices have a lot zeros and sparse matrix is a good representation and special solvers can be applied.

**Example.** (Sparse Matrix Solver)

```
import numpy as np
from scipy import sparse
N = 10
idx = np.r_[0:N]
v1 = 3*idx**2 +(idx/2)
v2 = -(6*idx**2 - 1)
v3 = 3*idx**2 -(idx/2)
A = sparse.spdiags(np.vstack((v1,v2,v3)),(-1,0,1),N,N).tocsc()
B = np.r_[N:0:-1]
X = sparse.linalg.spsolve(A, B)
```

---

Refer [https://lectures.scientific-python.org/advanced/scipy\\_sparse/index.html](https://lectures.scientific-python.org/advanced/scipy_sparse/index.html) to for more Scipy functions related sparse matrix.

## 9. Eigenvalue Problems and Matrix Functions

**Eigenvalues** are important in science and engineering because they are linked with **resonance frequencies, characteristic functions, etc.** The eigenvalue problem  $Ax = \lambda x$  has the following matrix form:

$$AX = \lambda X \quad (\text{EP})$$

Here  $A$  is an  $n \times n$  matrix. (EP) can be solved using:

```
from scipy import linalg
eigenvalues, eigenvectors = linalg.eig(A)
```

returning the eigenvalues (the diagonals of  $\Lambda$ ) and the normalised right eigenvectors as the columns of  $X$  for the square array  $A$ . For a special case where  $A$  is a Hermitian or symmetric, `linalg.eigh(A)` has a faster algorithm.

A (right) generalised matrix eigenvalue problem has the form:

$$AX = \lambda BX \quad (\text{GEP})$$

It can be solved using

```
eigenvalues, eigenvectors = linalg.eig(A,B).
```

**Example.** Write down the Python script to solve the following eigenvalue problem:

$$\begin{bmatrix} -2 & 11 \\ 17 & -19 \end{bmatrix} v = \lambda v$$

**Sample Solution:**

```
from scipy import linalg
A = np.array([-2,11],[17,-19])
lambdas, eigenvectors = linalg.eig(A)
print("/\=", lambdas)
print("X=", eigenvectors)
```

### 9.1. Linear Matrix Equations

There are a few matrix equations from linear control theory, signal processing, filtering, model reduction, image restoration, decoupling techniques for ordinary and partial differential equations below and the respective solvers in Python are listed.

Sylvester equations:

$$AX + BX = C$$

are solved with `linalg.solve_sylvester(A, B, C)` using the Bartels-Stewart algorithm.

A continuous-time algebraic Riccati equation (CARE):

$$XA + A^H X - XBR^{-1}B^H X + Q = 0$$

is `linalg.solve_continuous_are(A, B, Q, R[, E, S, ...])` in Python.

A discrete-time algebraic Riccati equation (DARE):

$$A^H X A - X - (A^H X B)(R + B^H X B)^{-1}(B^H X A) + Q = 0$$

is `linalg.solve_discrete_are(A, B, Q, R[, E, S, balanced])` in Python.

A continuous-time Lyapunov equation:

$$AX + XA^H = -Q$$

is `linalg.solve_continuous_lyapunov(A, -Q)` in Python.

A discrete-time Lyapunov equation:

$$AXA^H - X + Q = 0$$

is `linalg.solve_discrete_lyapunov(A, -Q[, method])` in Python.



## 9.2. Matrix Functions and Matrix Equations

A more general **nonlinear matrix problem** has the form:

$$f(X) = 0 \quad (\text{ME})$$

where  $X$  and  $0$  are  $n \times n$  square matrices. There is no simple / unified solution technique to this problem. A special case of (ME) has a quadratic left hand side leading to a 'quadratic matrix equation':

$$AX^2 + BX + C = 0$$

where  $X, A, B, C, 0$  are all  $n \times n$  matrices. For example,

$$\begin{bmatrix} -2 & 11 \\ & \end{bmatrix} X^2 + \begin{bmatrix} 19 & 1 \\ & \end{bmatrix} X + \begin{bmatrix} 0 & 13 \\ -13 & 13 \end{bmatrix} = \begin{bmatrix} 0 & 0 \\ 0 & 0 \end{bmatrix}$$

where  $X$  is a  $2 \times 2$  matrix.

The  $f(X)$  in (ME) can be a **matrix function** defined by the Taylor series for matrix of the form

$$f(X) = \sum_{k=0}^{\infty} \frac{f^{(k)}(0)}{k!} X^k.$$

Python has the exponential, logarithm, trigonometric and hyperbolic matrix functions such as `linalg.expm`, `linalg.logm`, `linalg.sinm`, `linalg.cosm`, `linalg.tanm`, `linalg.sinhm`, `linalg.coshm` and `linalg.tanhm`.

**Example.** (Final Exam Sept 2021 during MCO, Q1(d)) Given a  $2 \times 2$  matrix

$$A = \begin{bmatrix} 1 & -0.1 \\ 0.1 & 1 \end{bmatrix}.$$

Let  $X$  be a  $2 \times 2$  matrix with entries  $x_{ij}$ ,  $i, j = 1, 2$ . You are investigating the difference between the matrix exponential function

$$\exp^{[m]}(X) = I_2 + X + \frac{1}{2!} X^2 + \frac{1}{3!} X^3 + \frac{1}{4!} X^4 + \cdots + \frac{1}{k!} X^k + \cdots$$

and the elementwise exponential function

$$\exp(X) = \begin{bmatrix} e^{x_{11}} & e^{x_{12}} \\ e^{x_{21}} & e^{x_{22}} \end{bmatrix}$$

- (i) Write down the Python commands for calculating  $\exp^{[m]}(A)$  and  $\exp$ . Run the Python commands and write down the output of the commands. Then, write down the difference  $\exp^{[m]}(A) - \exp(A)$ .

**Solution:** The Python commands are respectively

```
* exp[m](A): linalg.expm(A)
* exp(A): np.exp(A)
```

The outputs are respectively

$$\begin{bmatrix} 2.70470174 & -0.27137536 \\ 0.27137536 & 2.70470174 \end{bmatrix} \quad \begin{bmatrix} 2.71828183 & 0.90483742 \\ 1.10517092 & 2.71828183 \end{bmatrix}$$

and the difference is

$$\begin{bmatrix} -0.01358009 & -1.17621278 \\ -0.83379556 & -0.01358009 \end{bmatrix}$$

- (ii) Write down the Python command to find the difference

$$\exp^{[m]}(A) - I_2 - A - \frac{1}{2!} A^2 - \frac{1}{3!} A^3$$

and write down the difference.

**Solution:** The Python command to find the difference is

$$\text{linalg.expm}(A) - \text{np.eye}(2) - A - 0.5*A*A - 1/6*A*A*A$$

and the output is

$$\begin{bmatrix} 0.04803508 & -0.02154203 \\ 0.02154203 & 0.04803508 \end{bmatrix}$$

## 10. Inline Functions, Anonymous Functions

Python does not have inline functions. Another implementation of Python called PyPy will inline functions automatically.

The lambda notion for function is the **anonymous function** in Python:

```
lambda x: an_expression_of x ...
```

It is usually used when we don't need to give an operation a function name.

**Example.** Sorting list strings reversely based on the characters and the number.

```
import re
table_data = ["vlan1", "usb0", "eth1", "vlan4", "vlan20"]
sorted(table_data, key=lambda v:
    [re.findall(r'([a-z]+)', v), -int(re.findall(r'(\d+)', v)[0])],
    reverse=True)
```

---

### Purpose of Section 8 and Section 9:

- \* We are trying to use examples to illustrate how to transform certain scientific problem to arrays and then to use the right numerical methods to get a reasonable solution/model to the scientific problem.

### Purpose of (this) Section 10:

- \* Inline functions is for improve speed but Python is only fast when the underlying C or C++ implementations are used. Anonymous functions are used when we want to do perform a data transformation for some computer algorithms (e.g. sorted).

## 11. Practical Topic: Arrays and Built-in Functions

Work through the examples in Sections 2.1 to 2.3, 3, 4.1 to 4.2, 5.1 to 5.4 and practical2\_array.py.

If time permits, try to construct statistical tables using Numpy array and built-in functions or Scipy statistical functions. For example, write down the Python program using array command(s) for constructing standard normal distribution cumulative table (less than Z) like the one shown in [https://en.wikipedia.org/wiki/Standard\\_normal\\_table](https://en.wikipedia.org/wiki/Standard_normal_table).

To understand the popularity of Python in scientific computing, the C++ program normtbl2.cc below using boost and armadillo linear algebra library is listed below for comparison.

```
#include <iostream>
#include <iomanip>
#include <string>
#include <boost/math/distributions.hpp>
#include <armadillo>
using namespace std;

const int prec = 5;
const int cw = prec + 3; // column width for printing table
const char nl[2] = "\n";
const string sepleft{" | "};
const string septop(cw-4, ' '); // 0.xy occupies 4 places

int main() {
    const double xs=0.0, xe=3.9, inc=0.1;
    const int N=int(round((xe-xs)/0.1))+1, M=10;
    arma::mat s_table(N,M); // statistiscal table
    cout << s_table.n_rows << " x " << s_table.n_cols << nl;

    // Print Top Table Header
    cout << string(5, ' '); // left label column
    for(int i=0; i<M; i++)
        cout << septop << fixed << setprecision(2) << 0.01*i;
    cout << nl << string(4, '-') << '+' << string(M*cw, '-') << nl;
    boost::math::normal_distribution<> normd;
    for(int i=0; i<N; i++) {
        for(int j=0; j<M; j++) {
            s_table(i,j) = cdf(normd, 0.1*i + 0.01*j);
        }
    }
    // https://stackoverflow.com/questions/63357023/print-
    // double-data-type-with-a-precision-upto-4-decimal-places-armadillo
    for(int row=0; row<N; row++){
        cout << setprecision(1) << row*0.1 << sepleft;
        cout.precision(prec);
        cout.setf(ios::fixed);
        s_table.row(row).raw_print(cout);
    }
}
```

## 12. Practical Topic: Advanced Array and Matrix

According to [https://en.wikipedia.org/wiki/Financial\\_engineering](https://en.wikipedia.org/wiki/Financial_engineering), **financial engineering** is a multidisciplinary field involving the application of mathematical finance and computational finance in the practice of finance. It is complex because it is trying to design financial products, in particular, complex financial derivatives are very difficult priced.

In this section, we explore the simplest financial derivative called the **European options**. Two typical European options are **European call options** and **European put options**. The former has a payoff

$$\text{payoff}(S, K) = \max(S_T - K, 0)$$

and the latter has a payoff

$$\text{payoff}(S, K) = \max(K - S_T, 0)$$

where  $S_T$  is the price of the underlying asset at time  $T$  and the constant  $K$  is the strike price.

Note that a **European Option** is just a contract between a seller and a buyer so that the buyer has the **right to exercise the option ONLY at the expiration date**. There is a contract between a seller and a buyer so that the buyer has the **right to exercise the option at ANY time up to expiration date** and it is called an **American Option**.

### 12.1. Practical Exercise

For a strike price  $K = 50$  and  $S_T$  for the range 40 to 60, plot the diagram for  $\text{payoff}(S, K)$  against  $S_T$  using the technique mentioned earlier.

## 12.2. Binomial Options Pricing Model (BOPM) with a Single-Period

The **binomial tree model** (see [https://en.wikipedia.org/wiki/Binomial\\_options\\_pricing\\_model](https://en.wikipedia.org/wiki/Binomial_options_pricing_model)) is a kind of computational method for the valuation of options using a "discrete-time" (lattice based) model of the varying price over time of the underlying financial instrument.

Binomial Options Pricing			
Time	Underlying Asset	Option	Replicate
$t = 0$	$S$	$C$	$S + B$
$t = T$	$S_d \ S_u$	$C_d \ C_u$	$e^{DT} S_d + e^{rT} B = C_d \quad e^{DT} S_u + e^{rT} B = C_u$

where  $C_u = \text{payoff}(S_u, K)$ ,  $C_d = \text{payoff}(S_d, K)$ ,  $D$  is the dividend yield rate of the underlying asset.

Solving the replicate, gives

$$B = e^{-DT} \left( \frac{C_u - C_d}{S_u - S_d} \right), \quad C = e^{-DT} \left( \frac{S_u C_d - S_d C_u}{S_u - S_d} \right).$$

and after rearranging, we have

$$C = S + B = e^{-rT} \left[ \left( \frac{e^{(r-D)T} - d}{u - d} \right) C_u + \left( \frac{u - e^{(r-D)T}}{u - d} \right) C_d \right]$$

where  $S_u = uS$  and  $S_d = dS$  and we can let

$$p = \frac{e^{(r-D)T} - d}{u - d}$$

the option price  $C$  becomes

$$C = e^{-rT} [pC_u + (1-p)C_d] \quad (\text{OP})$$

Note that in Cox-Ross-Rubinstein binomial tree model, additional assumptions are made:  $u = e^{\bar{h}}$  and  $d = e^{-\bar{h}}$ .

**Example.** Assuming the risk-free interest rate is 0.25 (annually compounding). For a non-dividend paying stock with a current stock price of 50. It is known that the price at the end of 12 months will either be 100 or 25, determine the value of the European call option to buy the stock at a strike 50.

**Sample Solution:**

```
import numpy as np
def payoff(S, K):
    return np.max([S-K, 0])

T = 1          # 12 months = 1 year
S0 = 50
Su = 100
Sd = 25
K = 50
D = 0          # zero-dividend
r = np.log(1 + 0.25) # convert to force of interest
u = Su / S0
d = Sd / S0
p = (np.exp(r*T) - d) / (u - d)
C = np.exp(-r*T) * (p*payoff(Su, K) + (1-p)*payoff(Sd, K))
print(f"The value of the European call option is {C}")
```

### 12.3. Multi-Period Binomial Options Pricing Model

Let  $S_u = uS_0$  and  $S_d = dS_0$ . The multi-period BOPM with a period  $h$  such that  $nh = T$  has the following table.

Multi-Period Binomial Options Pricing			
Time	Underlying Asset	Option	Option Replicate
$t = 0$	$S_0$	$C_0$	$S_0 + B$
$t = h$	$S_d \ S_u$	$C_d \ C_u$	$e^{Dh} S_d + e^{rh} B = C_d \quad e^{Dh} S_u + e^{rh} B = C_u$
$t = 2h$	$S_{dd} \ S_{ud} \ S_{uu}$	$C_{dd} \ C_{ud} \ C_{uu}$	...
$t = nh = T$	$S_{dd\dots d} \dots\dots S_{uu\dots u}$	$C_{dd\dots d} \dots\dots C_{uu\dots u}$	...

where  $S_{d\dots d u\dots u} = d^{i-j} u^j$  where  $i = 1, 2, \dots, n$ .

**Example.** Consider a non-dividend-paying stock with a current stock price of 41. Suppose the risk-free force of interest rate is 8%, calculate the value of a 1-year European put option with a strike price 43 by using the two-period binomial tree with  $u = 1.2870$  and  $d = 0.8419$ .

**Sample Solution:** By searching a bit from Internet, I found <https://github.com/Brucewuzhang/Financial-Engineering-and-risk-management-I-/>. The Jupyter notebook has something useful but the implementation is for Cox-Ross-Rubinstein binomial tree model

```
def binomial_tree(T,S0,u,d,n):
    tree = np.zeros((n+1,n+1))
    h = T/n
    for i in range(n+1):
        for j in range(i+1):
            tree[i,j]=(d**(i-j))*(u**j)*S0
    return tree
```

```
# Given
T = 1          # 1-year
n = 2          # 2-period model
S0 = 41
D = 0          # non-dividend-paying
r = 0.08
K = 43
u = 1.2870
d = 0.8419
S_tree = binomial_tree(T,S0,u,d,n)
print(S_tree)
```

The output is as follows

```
[[41.          0.          0.          ]
 [34.5179      52.767      0.          ]
 [29.06062001  44.4245373  67.911129   ]]
```

To determine the price of the option, we need to calculate the payoff but **the following will not work:**

```
np.max(K - S_tree, 0)          # wrong
```

The correct answer that give us the correct payoff is

```
def payoff_put(S):
    return np.max([K-S, 0])
```

```
f = np.vectorize(payoff_put)    # Vectorise the payoff
```

For European put option, we ONLY need to calculate the payoff at time  $T$  because this is when the contract is exercised:

```
f(S_tree[n,:])
```

The pricing can be calculated using (OP) by going backwards which is beyond this course.