UECM1703 Introduction to Scientific Computing
Topic 2: Arrays Manipulation

Dr Liew How Hui

Oct 2021

Outline

Numpy Array

We usually use Python's 'list' to construct Numpy array:

- List is slow, i.e. doing calculations with a list can be 10 times or sometimes even 100 times slower than Numpy arrays.
- List does not have nice "array operations".
- Numpy array has nice syntax similar to (but different from) MATLAB, which makes it nice for machine learning prototyping.

Numpy Array (cont)

An (n-D) *array* is a "n-index" **homogeneous** data structure with elements of the same type, i.e. `a[i,j,...,k]` and in maths, the notation is usually $a_{ij\ldots k}$ with $n = 1$, a_i; $n = 2$, a_{ij} as special cases.

We can have:

- Array of Floating Point Numbers (Key in Scientific Computing)
- Array of Integers
- Array of Booleans
- Array of Strings (not use in Scientific Computing)

Numpy Array (cont)

To use Numpy array (as mentioned in Week 1):

- import numpy as np

Creating 1-D arrays:

- `np.array([1,2,3,4],dtype='double')`
- `np.arange(1,5)`, `np.arange(50,1,-2)`
- `np.r_[1:5]`, `np.r_[1:50:2]`
- `np.linspace(start,stop,num=50, endpoint=True,retstep=False,dtype=None)`
- Special functions: np.zero, np.one, to be introduced later

Numpy Array (cont)

Creating 2-D arrays:

- Using list:
  `np.array([[1,3],[4,5]],dtype=np.double)`
- Reshaping from 1-D array: np.arange(1, 10).reshape((3,3))
- Stacking from 1-D array: np.vstack(([1,3,4,2],[4,2,3,1]))
- Stacking from 2-D arrays: np.hstack(([[1,2],[3,4]], [[4,3],[2,1]])), np.vstack(([[1,2],[3,4]], [[4,3],[2,1]]))
- Special functions: np.zero, np.one, to be introduced later

Numpy Array (cont)

Ways of creating “special” 3-D arrays:

- Using list: `np.array([[[6,3],[9,8]], [[1,3],[4,5]]],dtype=np.double)`.
- Reshaping from 1-D: `np.r_[1:(2*3*4+1)].reshape((2,3,4))`
- Stacking from 2-D arrays:
- Load from an “image” file:

```
from PIL import Image
imgarr = np.array(Image.open("a_colour_image.png"))
imgarr.shape
```

Ways of creating general $n \geq 4$-D arrays are similar to 3-D cases but are rarely used.

Outline

Special matrices (cont)

Creating “special” *n*-D arrays:

- Array of zeros: `np.zeros((2,4))`
- Array of ones: `np.ones`
- Array of value `v`: `np.full((m1,···,mk),v)`

Note: Matrices are just 2-D arrays in Numpy.

Special matrices (cont)

np.zeros(10) gives a 1-D array:

```
array([0., 0., 0., 0., 0., 0., 0., 0., 0., 0.])
```

np.zeros((6,6)) gives a 2-D array:

```
array([[0., 0., 0., 0., 0., 0.],
       [0., 0., 0., 0., 0., 0.],
       [0., 0., 0., 0., 0., 0.],
       [0., 0., 0., 0., 0., 0.],
       [0., 0., 0., 0., 0., 0.],
       [0., 0., 0., 0., 0., 0.]])
```

Special matrices (cont)

np.ones(10) gives a 1-D array:

```
array([1., 1., 1., 1., 1., 1., 1., 1., 1., 1.])
```

np.ones((6,3)) gives a 2-D array:

```
array([[1., 1., 1.],
       [1., 1., 1.],
       [1., 1., 1.],
       [1., 1., 1.],
       [1., 1., 1.],
       [1., 1., 1.]])
```

Special matrices (cont)

np.full(10, -1.0) gives a 1-D array:

```
array([-1., -1., -1., -1., -1., -1., -1., -1., -1., -1.])
```

np.full((6,3), 100) gives a 2-D array:

```
array([[100, 100, 100],
       [100, 100, 100],
       [100, 100, 100],
       [100, 100, 100],
       [100, 100, 100],
       [100, 100, 100]])
```

Special matrices

Creating “special” 2-D arrays:

- $n \times n$ identity matrix`np.eye(n)`. E.g.

$$\texttt{np.eye(4)} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}, \ \texttt{np.eye(3,2)} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \\ 0 & 0 \end{bmatrix}$$

- `np.diag(D)` is used to construct an $n \times n$ matrix with diagonal elements from 1-D array `D`. E.g.

$$\texttt{np.diag(np.arange(5,8))} = \begin{bmatrix} 5 & 0 & 0 \\ 0 & 6 & 0 \\ 0 & 0 & 7 \end{bmatrix}$$

Special matrices (cont)

We can create special type of ‘shifted’ diagonal matrices:

```
[[1, 0, 0, 0, 0],        [[0, 1, 0, 0, 0],        [[0, 0, 0, 0, 0],
 [0, 3, 0, 0, 0],         [0, 0, 3, 0, 0],         [0, 0, 0, 0, 0],
 [0, 0, 5, 0, 0],         [0, 0, 0, 5, 0],         [8, 0, 0, 0, 0],
 [0, 0, 0, 7, 0],         [0, 0, 0, 0, 7],         [0, 5, 0, 0, 0],
 [0, 0, 0, 0, 9]]         [0, 0, 0, 0, 0]]         [0, 0, 2, 0, 0]]

np.diag(np.r_[1:10:2])  np.diag([1,3,5,7],1)   np.diag([8,5,2],-2)
```

Sorry, diagonal matrix does not allow us to create something like this:

```
array([[0, 0, 0, 0, 1],
       [0, 0, 0, 3, 0],
       [0, 0, 5, 0, 0],
       [0, 7, 0, 0, 0],
       [9, 0, 0, 0, 0]])
```

Special matrices (cont)

In maths, we will come across the following Vandermonde matrix in linear algebra:

$$\begin{bmatrix} 1 & x_1 & \dots & x_1^{n-1} \\ 1 & x_2 & \dots & x_2^{n-1} \\ \vdots & \vdots & \ddots & \vdots \\ 1 & x_n & \dots & x_n^{n-1} \end{bmatrix} \quad 4\times 4 \text{ example : } \begin{bmatrix} 1 & 3 & 3^2 & 3^3 \\ 1 & 4 & 4^2 & 4^3 \\ 1 & 5 & 5^2 & 5^3 \\ 1 & 6 & 6^2 & 6^3 \end{bmatrix}$$

The 'reversed' can be generated by np.vander([3,4,5,6])

- `np.vander(`$[x_1, \cdots, x_n]$`)` is used to construct the Vandermonde matrix arises in interpolation.

Special matrices (cont)

There is a special matrix from `scipy.linalg` which is very close to the diagonal matrix, i.e. block_diag

```
>>> from scipy import linalg
>>> linalg.block_diag(np.eye(2), np.array([[3,2],[5,4]]), np.ones((3,3)))

array([[1., 0., 0., 0., 0., 0., 0.],
       [0., 1., 0., 0., 0., 0., 0.],
       [0., 0., 3., 2., 0., 0., 0.],
       [0., 0., 5., 4., 0., 0., 0.],
       [0., 0., 0., 0., 1., 1., 1.],
       [0., 0., 0., 0., 1., 1., 1.],
       [0., 0., 0., 0., 1., 1., 1.]])
```

There is one related to “circular” pattern:

```
>>> linalg.circulant([2,3,4,5])

array([[2, 5, 4, 3],
       [3, 2, 5, 4],
       [4, 3, 2, 5],
       [5, 4, 3, 2]])
```

Special matrices (cont)

There is another one related to “shifting” pattern called Toeplitz matrix:

$$A = \begin{bmatrix} a_0 & a_{-1} & a_{-2} & \cdots & \cdots & a_{-(n-1)} \\ a_1 & a_0 & a_{-1} & \ddots & & \vdots \\ a_2 & a_1 & \ddots & \ddots & \ddots & \vdots \\ \vdots & \ddots & \ddots & \ddots & a_{-1} & a_{-2} \\ \vdots & & \ddots & a_1 & a_0 & a_{-1} \\ a_{n-1} & \cdots & \cdots & a_2 & a_1 & a_0 \end{bmatrix}$$

Special matrices (cont)

Toeplitz can be generated using linalg.toeplitz($[a_0, a_1, a_2, \cdots, a_{n-1}], [x, a_{-1}, a_{-2}, ..., a_{-n}, ...]$).

For example,

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

array([[2, 6, 7, 8, 9],
       [3, 2, 6, 7, 8],
       [4, 3, 2, 6, 7],
       [5, 4, 3, 2, 6]])
```

Special matrices (cont)

Other special matrices related to science and engineering's numerical problems are available from `scipy.linalg` module:

- `companion(a)`, `convolution_matrix(a, n[, mode])`, `dft(n[, scale])`, `fiedler(a)`, `hadamard(n[, dtype])`, `hankel(c[, r])`, `helmert(n[, full])`, `hilbert(n)`, `invhilbert(n[, exact])`, `leslie(f, s)`, `pascal(n[, kind, exact])`, `invpascal(n[, kind, exact])`, `toeplitz(c[, r])`, `tri(N[, M, k, dtype])`

Special matrices (cont)

Special matrices with random numbers are essential in computer simulation.

- An array of uniform random numbers between 0 and 1:

```
np.random.rand(m1,...,mk)
np.random.random((m1,...,mk))
```

- An array of normally distributed random numbers:

```
np.random.randn(d0,d1,...,dn)
```

- To prevent the random numbers to be different every time we use them, the seed should be set:

```
np.random.seed(some_integer)
```

Special matrices (cont)

Let us generate a 6×6 'uniform' $[0, 1)$ random matrix

```
>>> np.random.seed(202010)
>>> A = np.random.rand(6,6)
A = array([[0.2 , 0.42, 0.18, 0.28, 0.  , 0.19],
           [0.29, 0.49, 0.96, 0.31, 0.66, 0.79],
           [0.84, 0.77, 0.44, 0.39, 0.65, 0.91],
           [0.47, 0.15, 0.52, 0.1 , 0.97, 0.62],
           [0.43, 0.93, 0.33, 0.77, 0.87, 0.51],
           [0.48, 0.41, 0.42, 0.22, 0.03, 0.8 ]])
```

Using Topic 4's plt.hist, we can try to see if it is 'uniform' enough.

Special matrices (cont)

Let us generate a 6×6 'uniform' $[0, 1)$ random matrix

```
>>> np.random.seed(202010)
>>> B = np.random.randn(6,6)
B = array([[-0.36, -1.35, -0.57, -0.84, -0.11, -1.85],
           [-0.05,  0.11,  1.14,  0.64,  0.47,  0.58],
           [-2.02, -1.17,  0.66,  0.25, -1.18, -0.09],
           [-0.96,  0.05,  0.1 ,  0.38,  0.72, -0.11],
           [ 1.14, -0.73,  0.03,  1.11, -2.51, -0.64],
           [-1.41, -0.38,  0.12,  1.56, -0.77, -0.13]])
```

Using Topic 4's plt.hist, we can try to see if it is 'normal' enough.

Sparse matrices

A sparse matrix is a kind of special structure to store a matrix with many zeros. It is usually used in numerical partial differential equation (PDE, which is an elective for AM year 3 students) problems.

Another application of sparse matrix is in network / graph problems.

The support for sparse matrix is available in scipy.sparse module.

Outline

Size, Reshape

The 'size' of an array is called **shape** in Numpy. So if we can use the following command to check the 'size':

```
A.shape
```

`A.reshape(6)` can be used to reshape a 2D array with 3 rows, 2 columns to a 1D array with 6 elements and vice versa. In general, it can be used to reshape to 'any' dimension as long as the number of elements are **compatible**. E.g. you cannot reshape a 1D array of 5 elements to 2D array with 6 elements.

Note that all *n*-D array `A` can be converted 1-D array using `A.ravel()` or `A.flatten()`.

Transpose

The transpose of an nD array $a_{i,j,\dots,k}$ is

$$[a_{i,j,\dots,k}]^T = [a_{k,\dots,j,i}]$$

For 1D array, the transpose is the same as itself:

```
>>> np.arange(1,10).T  # Alternate syntax: np.transpose

array([1, 2, 3, 4, 5, 6, 7, 8, 9])
```

For 2D array, the transpose of A is the same as matrix transpose A^T:

```
>>> np.arange(1,9).reshape((2,4)).T

array([[1, 2, 3, 4],        array([[1, 5],
       [5, 6, 7, 8]]) ->            [2, 6],
                                    [3, 7],
                                    [4, 8]])
```

Transpose (cont)

For 3D array, the transpose turns a $2 \times 3 \times 4$ 3D-matrix (two 3×4 matrix) to a $4 \times 3 \times 2$ 3D-matrix (four 3×2 matrix):

```
>>> np.transpose(np.arange(1,25).reshape((2,3,4)))

array([[[ 1,  2,  3,  4],           array([[[ 1, 13],
        [ 5,  6,  7,  8],                   [ 5, 17],
        [ 9, 10, 11, 12]],                  [ 9, 21]],
                                  ->
       [[13, 14, 15, 16],                  [[ 2, 14],
        [17, 18, 19, 20],                   [ 6, 18],
        [21, 22, 23, 24]]])                 [10, 22]],

                                           [[ 3, 15],
                                            [ 7, 19],
                                            [11, 23]],

                                           [[ 4, 16],
                                            [ 8, 20],
                                            [12, 24]]])
```

"Sub-array" Indexing / Slicing

Sub-array can be regarded as part of the original array. The usual way to take a sub-array is by indexing (compare it to rows and columns in Excel).

Consider an array

$$A = [a_{i,j,\ldots,k}]$$

Each of the i, j, k is called an index. An index is an integer or a sequence of integers to 'get' the elements of an array A.

Indexing (cont)

Numpy provides use a few ways to 'get' / 'view' the elements from an array.

- a number. E.g. 3
- a list of number. E.g. [1,2,3]
- a sequence pattern(?). E.g. 1:4 (like [1,2,3]), 1:6:2 (like [1,3,5]), 5::-1 (like [5,4,3,2,1,0]).
- special pattern. : is used to denote all elements

Indexing (cont)

Consider a 1-D array:

```
>>> a = np.arange(10,100,10)
>>> a[3]      # gives 40 because the index starts from 0
>>> a[3:10]   # gives 40 50 60 70 80 90 as expected
>>> a[3:10:2]  # gives 40 60 80 as expected
>>> a[:5]     # gives 10 20 30 40 as expected
>>> a[5:]     # gives 50 60 70 80 90 as expected
>>> a[9::-1]       # reverse the array as expected
>>> a[[3,5,2]]     # gives 40 60 30
>>> a[[2,2,2,2]]   # gives 30 30 30 30
>>> a[:]      # same as a as expected
```

Note that when the ‘index’ in the pattern `a:b:c` is beyond the bound, Python will return empty. Therefore,

```
>>> a[5:] == a[5:100000]
```

Of course we shouldn’t like this! It is confusing!

Indexing (cont)

Consider a 2-D array `A`.

- Indexing an element of `A` at `(m,n)`: `A[m-1,n-1]`
- Indexing `A` at `m`-row: `A[m-1,:]` or `A[m-1]` (both are 1D array); for 2D, use `A[[m-1]]`
- Indexing `A` at `n`-column: `A[:,n-1]` (1D), `A[:,[n-1]]` (2D)
- Indexing using rows & columns: `A[m1-1:m2,n1-1:n2]`, `A[m1-1:m2:ms,n1-1:n2:ns]`.
- Indexing a sub-array of `A` using rows `r1`, $\cdots$, `rk`, columns `c1`, $\cdots$, `cl`: `A[[r1-1,...,rk-1],:][:,[c1-1,...,cl-1]]`

Indexing (cont)

Conside a 2-D array:

```
A=np.arange(1,55,dtype=np.double).reshape(6,9)
```

A =

	0	1	2	3	4	5	6	7	8
0	1	2	3	4	5	6	7	8	9
1	10	11	12	13	14	15	16	17	18
2	19	20	21	22	23	24	25	26	27
3	28	29	30	31	32	33	34	35	36
4	37	38	39	40	41	42	43	44	45
5	46	47	48	49	50	51	52	53	54

Study the following “indexing”/“slicing” commands:

```
(a) A[3,4]          (b) A[3,:]             (c) A[:,4]
(d) A[1:3,3:7]      (e) A[1:6:2,:][:,3:8:2]
```

Try finding the answer without the help of Python.

Indexing (cont)

By now, we can see that the ‘index’ can be used for the following purpose:

- Obtain sub-array(s): E.g. cutting a “rectangular” piece of large image.
- Arranging matrix to special form: E.g.

$$A = \begin{bmatrix} 1 & 1 & 3 \\ 5 & 2 & 2 \\ 3 & 8 & 3 \end{bmatrix} \Rightarrow \texttt{A[[1,2,0],:]} = \begin{bmatrix} 5 & 2 & 2 \\ 3 & 8 & 3 \\ 1 & 1 & 3 \end{bmatrix}$$

The right-hand matrix is “diagonally dominant”.

Indexing (cont)

Example:

```
array([[0, 0, 0, 0, 1],          array([[  1,   3,   9,  27],
       [0, 0, 0, 3, 0],                 [  1,   4,  16,  64],
       [0, 0, 5, 0, 0],                 [  1,   5,  25, 125],
       [0, 7, 0, 0, 0],                 [  1,   6,  36, 216]])
       [9, 0, 0, 0, 0]])
```

With index, we can generate the above matrices mentioned earlier (e.g. in Slide 15).

```
np.diag(np.r_[1:10:2])[:,4::-1]
np.vander([3,4,5,6])[:,3::-1]
```

Note that the following also works:

```
np.diag(np.r_[1:10:2])[:,5::-1]
np.vander([3,4,5,6])[:,len([3,4,5,6])::-1]
```

Indexing (cont)

Numpy provides an alternative indexing function `take` (or `put`) to **take** (or **assign** values to) a “slice” of elements from an array `A`. To get an element from an item, the function `item` is used instead:

```
(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)
```

Indexing (cont)

A 3-D array is used in the representation of a coloured image. Python's PIL.Image.open or imageio.imread could be used to load images to 3-D arrays.

```
import numpy as np, matplotlib.pylab as plt
from PIL import Image
im = Image.open("emoji1.jpg")   # https://getemoji.com/
imarr = np.array(im)   # shape = (5,5,3)
fig=plt.figure(figsize=(8, 8))
mycmaps = [plt.cm.Reds_r,plt.cm.Greens_r,plt.cm.Blues_r]
nrows, ncols = 1, 4
for i in range(3):
    fig.add_subplot(nrows, ncols, i+1)
    plt.imshow(imarr[:,:,i],interpolation='None',cmap=mycmaps[i])
fig.add_subplot(nrows, ncols, 4)
plt.imshow(imarr)
plt.show()
```

Indexing: 3D Example (cont)

`imarr.shape` $\Rightarrow$
Red: `imarr[:,:,0])`

$$\Rightarrow \begin{bmatrix}
14 & 4 & 0 & 0 & 6 & 19 & 29 & 27 & 29 & 14 & 15 & 7 & 8 & 19 & 11 \\
10 & 255 & 255 & 255 & 255 & 247 & 243 & 241 & 224 & 234 & 231 & 249 & 255 & 255 & 15 \\
14 & 255 & 255 & 255 & 241 & 247 & 246 & 249 & 252 & 236 & 230 & 251 & 248 & 255 & 3 \\
7 & 255 & 255 & 233 & 240 & 253 & 214 & 232 & 251 & 240 & 240 & 245 & 230 & 227 & 0 \\
3 & 255 & 238 & 236 & 227 & 156 & 116 & 138 & 237 & 209 & 141 & 123 & 218 & 221 & 0 \\
19 & 255 & 231 & 247 & 170 & 132 & 136 & 117 & 225 & 135 & 114 & 110 & 151 & 203 & 24 \\
53 & 252 & 252 & 255 & 228 & 180 & 176 & 207 & 243 & 195 & 161 & 158 & 228 & 255 & 16 \\
84 & 255 & 255 & 255 & 255 & 255 & 255 & 255 & 253 & 255 & 255 & 255 & 255 & 255 & 20 \\
85 & 255 & 247 & 255 & 255 & 255 & 255 & 255 & 251 & 255 & 255 & 255 & 255 & 225 & 33 \\
52 & 255 & 245 & 249 & 240 & 243 & 254 & 229 & 255 & 245 & 244 & 240 & 243 & 220 & 30 \\
16 & 255 & 241 & 243 & 241 & 241 & 243 & 203 & 145 & 177 & 227 & 247 & 246 & 224 & 20 \\
6 & 255 & 249 & 244 & 247 & 251 & 252 & 226 & 235 & 238 & 246 & 246 & 245 & 233 & 5 \\
7 & 255 & 255 & 251 & 245 & 249 & 252 & 255 & 250 & 250 & 253 & 248 & 247 & 244 & 0 \\
6 & 255 & 255 & 255 & 255 & 253 & 244 & 253 & 252 & 240 & 235 & 246 & 255 & 255 & 12 \\
1 & 254 & 249 & 252 & 255 & 255 & 252 & 242 & 227 & 238 & 252 & 255 & 255 & 255 & 8 \\
0 & 251 & 240 & 245 & 254 & 253 & 255 & 255 & 255 & 255 & 251 & 255 & 255 & 255 & 7 \\
0 & 0 & 0 & 0 & 0 & 3 & 6 & 6 & 4 & 3 & 1 & 1 & 1 & 1 & 3
\end{bmatrix}$$

Green: `imarr[:,:,1])`

Blue: `imarr[:,:,2])`

Ref: `https://en.wikipedia.org/wiki/RGB_color_model`

Indexing (cont)

Apart from getting 'sub-array', indexing can be used to **changed** part of the array. For example, one way of creating the following matrix

$$\begin{bmatrix} 1 & 1.5 & 2 & 2.5 \\ 1 & 1.5 & 2 & 2.5 \\ 1 & 1.5 & 2 & 2.5 \end{bmatrix}$$

is

```
A = np.zeros((3,4))
A[:,0] = 1      # A.put([0,4,8],1)
A[:,1] = 1.5    # A.put([1,5,9],1.5)
A[:,2] = 2      # A.put([2,6,10],2)
A[:,3] = 2.5    # A.put([3,7,11],2.5)
```

Indexing: Beware of Array View

The index usually gives the 'view' (not a 'copy') of an array:

```
import numpy as np
A = np.array([[1,2,3],[4,5,6]])
B = A[:,0:2]
B[0,0] = 8   # We only change an element in B
print("A=",A)
print("B=",B)
```

So if we refer to a subarray B of A, then when we change B, A will also be changed:

$$A = \begin{bmatrix} 8 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix}, \quad B = \begin{bmatrix} 8 & 2 \\ 4 & 5 \end{bmatrix}$$

To get a copy, use the following instead:

```
B = A[:,0:2].copy()
```

Indexing, etc.: Summary

The basic methods (operations) associated with Numpy array `A`:

- Information about an array:
  - `A.ndim`: the dimension of `A`;
  - `A.shape`: $(m_1, m_2, \cdots, m_k)$, k=dim `A`
  - `A.size`: number of elements $m_1 \times m_2 \times \cdots \times m_k$
  - `A.nbytes`: the total number of bytes used, i.e. `A.size*A.itemsize`. E.g. 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 $8n$.
- "Negative" indexing: Count backwards. Not very nice for scientific computing.

Indexing, etc.: Summary (cont)

1. Getting a “view” of the array `A`.
   - Indexing usually returns a “view” of `A`.
   - Transpose view: `A.T`, `np.transpose(A)`.
2. Creating a “copy” of an array `A`.
   - Return a duplicate of `A`: `A.copy()`.
   - 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` and `np.vstack`.

For a 2D matrix A, using index to **take** elements from A can be seen as rearranging ‘square’ portion of the ‘image’ A; while using index to **assign** values to A can be seen as changind ‘colours’ in the rectangular region in ‘image’ A.

Example: Sept 2014 FE, Q2(b)

Find the 4×4 matrix M after the following Python code is run:

```
for i in range(4):
    for j in range(i,4):
        M[i,j] = i+j
        M[j,i] = M[i,j]-2
```

Remark: Dr Goh will discuss more about for loop in Topic 3.

The code is the same as

```
for i,j in [(0,0), (0,1), (0,2), (0,3),
                   (1,1), (1,2), (1,3),
                          (2,2), (2,3),
                                 (3,3)]:
    M[i,j] = i+j; M[j,i] = M[i,j]-2
```

Example: Sept 2014 FE, Q2(b) cont

Expanding on the Python code, we obtain

```
M[0,0] = 0+0; M[0,0] = M[0,0]-2
M[0,1] = 0+1; M[1,0] = M[0,1]-2
M[0,2] = 0+2; M[2,0] = M[0,2]-2
M[0,3] = 0+3; M[3,0] = M[0,3]-2
M[1,1] = 1+1; M[1,1] = M[1,1]-2
M[1,2] = 1+2; M[2,1] = M[1,2]-2
M[1,3] = 1+3; M[3,1] = M[1,3]-2
M[2,2] = 2+2; M[2,2] = M[2,2]-2
M[2,3] = 2+3; M[3,2] = M[2,3]-2
M[3,3] = 3+3; M[3,3] = M[3,3]-2
```

The values in the matrix M is

```
[[-2.  1.  2.  3.]
 [-1.  0.  3.  4.]
 [ 0.  1.  2.  5.]
 [ 1.  2.  3.  4.]]
```

Final Exam Oct 2020 Q1(a)

Given that `A` stores the following matrix

$$\begin{bmatrix}
4 & 0 & 0 & 0 & 0 & 15 & 8 & 1 & 0 & 0 \\
0 & 6 & 0 & 0 & 0 & 6 & 24 & 6 & 1 & 0 \\
0 & 0 & 6 & 0 & 0 & 1 & 8 & 15 & 4 & 4 \\
1 & 0 & 0 & 3 & 0 & 0 & 8 & 4 & 18 & 5 \\
2 & 3 & 0 & 0 & 5 & 0 & 0 & 8 & 6 & 24 \\
29 & 3 & 1 & 0 & 0 & 5 & 0 & 0 & 3 & 7 \\
3 & 17 & 5 & 6 & 0 & 0 & 7 & 0 & 0 & 2 \\
4 & 4 & 17 & 3 & 4 & 0 & 0 & 6 & 0 & 0 \\
0 & 8 & 2 & 25 & 4 & 0 & 0 & 0 & 8 & 0 \\
0 & 0 & 7 & 1 & 16 & 0 & 0 & 0 & 0 & 7
\end{bmatrix}.$$

Final Exam Oct 2020 Q1(a) cont

(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. (1 mark)

Q: Why is the matrix so large?

A: During Movement Control Order (MCO), the final exam is open book. So the matrix is large to prevent easy copy and paste into a Python program and the answers could be easily obtain.

To answer: Note that `A[:,[3,5,2,4]]` means take the 4th, 6th, 3rd and 5th columns.

`A[[3,5,2,4]]` is the same as `A[[3,5,2,4],:]` which means take the 4th, 6th, 3rd and 5th rows.

Final Exam Oct 2020 Q1(a) cont

Answer

Take the 4th, 6th, 3rd and 5th columns from `A`, we get

```
[[ 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]]
```

. . . . [0.8 mark]

`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. . . . . [0.2 mark]

Final Exam Oct 2020 Q1(a) cont

(ii) Write the Python command to pick all the odd rows and even columns from **A** and write down the output of your command. (1 mark)

Answer

Following what we mentioned earlier: 1::2 means 1,3,5,... and ::2 (if empty it will be default, for left most is 0) means 0, 2,4,...

`A[::2,1::2]` ....[0.7 mark]

```
[[ 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]]
```

Final Exam Oct 2020 Q1(a) cont

(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. (1 mark)

Answer

`A[:,[1,4,2]][[7,4,6],:]` ....[0.7 mark]

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

....[0.3 mark]

Final Exam Oct 2020 Q1(a) cont

(iv) Write the Python command to arrange the given matrix $\mathbf{A}$ into the following diagonally dominant form:

$$\begin{bmatrix} 15 & 8 & 1 & 0 & 0 & 4 & 0 & 0 & 0 & 0 \\ 6 & 24 & 6 & 1 & 0 & 0 & 6 & 0 & 0 & 0 \\ 1 & 8 & 15 & 4 & 4 & 0 & 0 & 6 & 0 & 0 \\ 0 & 8 & 4 & 18 & 5 & 1 & 0 & 0 & 3 & 0 \\ 0 & 0 & 8 & 6 & 24 & 2 & 3 & 0 & 0 & 5 \\ 5 & 0 & 0 & 3 & 7 & 29 & 3 & 1 & 0 & 0 \\ 0 & 7 & 0 & 0 & 2 & 3 & 17 & 5 & 6 & 0 \\ 0 & 0 & 6 & 0 & 0 & 4 & 4 & 17 & 3 & 4 \\ 0 & 0 & 0 & 8 & 0 & 0 & 8 & 2 & 25 & 4 \\ 0 & 0 & 0 & 0 & 7 & 0 & 0 & 7 & 1 & 16 \end{bmatrix}$$

(0.5 mark)

Answer

`A[:,[5,6,7,8,9,0,1,2,3,4]]` ....[0.5 mark]

Final Exam Oct 2020 Q1(a) cont

(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:

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

Write a Python function `is_diag_domin(A)` which determines whether the matrix A is diagonally dorminant. The function with return True if the matrix A is diagonally dorminant, False if the matrix A is not diagonally dorminant, and None if the matrix is not square. (1 mark)

Final Exam Oct 2020 Q1(a) cont

To understand what the question is saying, let us look at a simple example:

$$\begin{bmatrix} -20 & 1 & 2 \\ 3 & 41 & -4 \\ -5 & 6 & -52 \end{bmatrix}$$

This is diagonally dominant because

- $|-20| = 20 > |1| + |2| = 3$, i.e. $|a_{11}| > |a_{12}| + |a_{13}|$
- $|41| = 41 > |3| + |-4| = 7$, i.e. $|a_{22}| > |a_{21}| + |a_{23}|$
- $|-52| = 52 > |-5| + |6| = 11$, i.e. $|a_{33}| > |a_{31}| + |a_{32}|$

are **all true**

Final Exam Oct 2020 Q1(a) cont

Let us look at a simple example which is **not** diagonally dominant :

$$\begin{bmatrix} -20 & 1 & 2 \\ 3 & 41 & -49 \\ -5 & 6 & -52 \end{bmatrix}$$

We learn from logic: If “not all true”, then ‘at least one is false’!

- $|-20| = 20 > |1| + |2| = 3$, i.e. $|a_{11}| > |a_{12}| + |a_{13}|$
- $|41| = 41 \not> |3| + |-49| = 52$, i.e. $|a_{22}| \not> |a_{21}| + |a_{23}|$
- we don’t need to care about the rest since we have already found one row which does not satisfied the inequality.

Final Exam Oct 2020 Q1(a) cont

Based on logic, we can write something like this:

```
N = A.shape[1]    # Assume matrix A is NxN
is_dd = True      # Initialise the variable is_dd
for row in range(N):
    i = row
    if A[i,i] > A[i,0] + ... + A[i,i-1] +
                A[i,i+1] + ... + A[i,N-1] then
        is_dd = is_dd and True
    else:
        is_dd = False
```

Of course the above won't work because direct translation from maths to Python programming is impossible.

Final Exam Oct 2020 Q1(a) cont

Answer

A sample implementation of the script (other equivalent answers will be accepted):

```
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))
```

. . . . [1 mark]

Outline

Elementwise Operations

Elementwise operations:

- `A+B`, `A-B`
- `A*B`, `A/B`
- `A+=B`, `A-=B`, `A*=B`, `A/=B`
- `A==B`, `A~=B`
- `A<B`, `A<=B`, `A>B`, `A>=B`
- `f(A)`

They work for n-D arrays, i.e. both A and B need to be n-D arrays. In particular, they are valid for $n = 2$ and $n = 1$, i.e. matrices and 1-D arrays.

Elementwise Operations (cont)

Consider

$$A = \begin{bmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 9 & 9 & 9 \end{bmatrix}, \quad B = \begin{bmatrix} 9 & 8 & 7 & 6 \\ 5 & 4 & 3 & 2 \\ 1 & 1 & 1 & 1 \end{bmatrix}$$

We can generate them using the Python commands:

- `A = np.vstack((np.r_[1:9].reshape((2,4)),[9]*4))`
- `B = np.vstack((np.r_[9:1:-1].reshape((2,4)),[1]*4))`

Now, we can explore the elementwise operations:

```
A + B =
array([[10, 10, 10, 10],
       [10, 10, 10, 10],
       [10, 10, 10, 10]])
```

```
A - B =
array([[-8, -6, -4, -2],
       [ 0,  2,  4,  6],
       [ 8,  8,  8,  8]])
```

Elementwise Operations (cont)

Elementwise multiplication and elementwise division: The former is different from matrix multiplication, the later is defined but matrix does not has a proper division operation.

```
>>> A * B
array([[ 9, 16, 21, 24],
       [25, 24, 21, 16],
       [ 9,  9,  9,  9]])

>>> A / B
array([[0.11111111, 0.25      , 0.42857143, 0.66666667],
       [1.        , 1.5       , 2.33333333, 4.        ],
       [9.        , 9.        , 9.        , 9.        ]])
```

Elementwise Operations (cont)

The following syntax are taken from C language:

- `A+=B`: `A = A+B`
- `A-=B`: `A = A-B`
- `A*=B`: `A = A*B`
- `A/=B`: `A = A/B`

The elementwise relational operators compare elements by elements just like the elementwise arithmetic operations:

- `A==B`, `A!=B`, `A<B`, `A<=B`, `A>B`, `A>=B`

Consider $A = \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}$ and $B = \begin{bmatrix} 5 & 4 \\ 3 & 2 \end{bmatrix}$.

```
>>> A == B, A != B, A < B, A >= B
[[False, False],  [[ True,  True],  [[ True,  True],  [[False, False],
 [ True, False]]   [False,  True]]   [False, False]]   [ True,  True]]
```

Elementwise Operations (cont)

The elementwise 'function' operation `f(A)` is usually encountered when we want to plot a graph of f.

Consider $x = [0, 0.1, 0.2, ..., 6.5]$. If want to plot $\sin x_i$ at each value x_i of x. In 'array' form:

$$y = [\sin(0),\ \sin(0.1),\ \sin(0.2), ...,\ \sin(6.5)].$$

The corresponding Python commands are

```
>>> x = np.arange(0,6.6,0.1)
>>> y = np.sin(x)
```

In general, for n-D array, the definition is

$$f(A) = [f(a_{i,j,...,k})]$$

Elementwise Operations (cont)

Why do we need elementwise operations?
We need them in

- numerical methods:
  - finding the difference $A - B$ between exact matrix A and estimated matrix B.
  - update a matrix A to $A + U$ in an iteration: `A = A+U` or `A += U`.
- statistics and data analysis: E.g. shifting the columns in a matrix to centre at the mean.

Matrix Arithmetic

The operations we commonly use are the **scalar multiplication** and the **matrix multiplication**:

$$cA = [ca_{ij}], \quad AB = \left[\sum_{j=1}^{n} a_{ij}b_{jk}\right].$$

E.g. $A_1 = \begin{bmatrix} 0 & 1 & 2 & 3 \\ 4 & 5 & 6 & 7 \end{bmatrix}$, $B_1 = \begin{bmatrix} 0 & 1 \\ 2 & 3 \\ 4 & 5 \\ 6 & 7 \end{bmatrix}$,

$A_2 = \begin{bmatrix} 8 & 9 & 10 & 11 \\ 12 & 13 & 14 & 15 \end{bmatrix}$, $B_2 = \begin{bmatrix} 8 & 9 \\ 10 & 11 \\ 12 & 13 \\ 14 & 15 \end{bmatrix}$.

Matrix Arithmetic (cont)

The matrix multiplication $A_1 B_1$, $A_2 B_2$ are

$$A_1 B_1 = \begin{bmatrix} 28 & 34 \\ 76 & 98 \end{bmatrix}, \quad A_2 B_2 = \begin{bmatrix} 428 & 466 \\ 604 & 658 \end{bmatrix}$$

The matrix multiplication in Numpy is handle by the command `np.matmul(A, B)` or `A @ B`.

Note that if A and B are 1-D, then `A @ B` works as dot product $a_1 b_1 + a_2 b_2 + ... + a_n b_n$.

When A and B are 2-D (and 1-D), the following n-D array arithmetic operation is the same in function:

- `np.dot(A, B)`: Dot product of two arrays, giving $z[I, J, j] = \sum_k A[I, k] B[J, k, j]$.

But for 3-D and above, they are different.

Matrix Arithmetic (cont)

In Python, we can use the following commands to do the calculation for the A_1B_1, A_2B_2 multiplication example.

```
A1 = np.r_[0:8].reshape((2,4))
B1 = np.r_[0:8].reshape((4,2))
A2 = np.r_[8:16].reshape((2,4))
B2 = np.r_[8:16].reshape((4,2))
Product1 = A1 @ B1  # Same as np.matmul(A1,B1)
Product2 = A2 @ B2  # Same as np.matmul(A2,B2)
```

Amazingly, the np.matmul can be used to 'stack' matrix multiplications.

```
A = np.r_[0:16].reshape((2,2,4))  # Two 2x4 matrices
B = np.r_[0:16].reshape((2,4,2))  # Two 4x2 matrices
Product = A @ B                   # Two 2x2 matrices
```

More Arithmetics

There are many other arithmetic operations which are beyond this subject:

- `np.linalg.multi_dot(arrays)`: Compute the dot product of two or more arrays in a single function call, while automatically selecting the fastest evaluation order.
- `np.inner(a, b)`: Inner product of two compatible arrays with outocme $\sum_{i=0}^{K} A_{i_1,i_2,\ldots,i_{I-1},i} B_{j_1,j_2,\ldots,j_{J-1},i}$.
- `np.outer(a, b[, out])`: Compute the outer product of two vectors, i.e. $[x_i y_j]$.
- `np.tensordot(a, b[, axes])`: Compute tensor dot product along specified axes for arrays $\geq$ 1-D.

More Arithmetics (cont)

- `np.einsum(subscripts, *operands[, out, dtype, ...])`: Evaluates the Einstein summation convention on the operands.
- `np.einsum_path(subscripts, *operands[, optimize])`: Evaluates the lowest cost contraction order for an einsum expression by considering the creation of intermediate arrays.
- `np.kron(A,B)`: Kronecker product of two arrays, giving $[a_{ij\cdots k}B]$.

np.kron is the only slightly easier to understand array operation which we have some simple examples.

Example: Kronecker Product

Let the Kronecker product of two matrices A and B denote as kron(A,B) or $A \otimes B$.

$$A = \begin{bmatrix} a & b & c \\ d & e & f \end{bmatrix}, \quad A \otimes B = \begin{bmatrix} aB & bB & cB \\ dB & eB & fB \end{bmatrix}$$

If B is a 2×2 matrix with all 1's, then

$$A \otimes B = \begin{bmatrix} a & a & b & b & c & c \\ a & a & b & b & c & c \\ d & d & e & e & f & f \\ d & d & e & e & f & f \end{bmatrix}$$

Note that $A \otimes B \neq B \otimes A$ in general.

Example: Kronecker Product (cont)

Q: What can we do with Kronecker Product?

Answer 1: We can use it to generate the results of multiplication tables for 2 to 9 (or any integers).

```
>>> np.kron(np.r_[2:10].reshape((-1,1)), np.r_[2:10])
array([[ 2,  3,  4,  5,  6,  7,  8,  9],
       [ 4,  6,  8, 10, 12, 14, 16, 18],
       [ 6,  9, 12, 15, 18, 21, 24, 27],
       [ 8, 12, 16, 20, 24, 28, 32, 36],
       [10, 15, 20, 25, 30, 35, 40, 45],
       [12, 18, 24, 30, 36, 42, 48, 54],
       [14, 21, 28, 35, 42, 49, 56, 63],
       [16, 24, 32, 40, 48, 56, 64, 72],
       [18, 27, 36, 45, 54, 63, 72, 81]])
```

The -1 in reshape is used to ask Python to count how many elements are there in the 1-D array, which is convenient.

Example: Kronecker Product (cont)

Answer 2: It can be used the case where there are obvious block patterns in a matrix such as

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

The following command can be used to generate it:

```
>>> np.kron(np.array([[2,1],[1,2]]),
            np.array([[3,4,5],[5,4,3]]))
```

Matrix Arithmetics (cont)

Let us end the matrix arithmetics with two scaling techniques we may encounter in data analysis:

- Min-max scaling
- Standard scaling / standardisation

Consider the following matrix

$$A = \begin{bmatrix} -5 & -4 & -3 \\ -2 & -1 & 0 \\ 1 & 2 & 3 \\ 3 & 6 & 9 \end{bmatrix}$$

For min-max scaling, it transforms the each column of A to

$$(\text{column } i - \text{min of column } i)/(\text{range of column } i).$$

Matrix Arithmetics (cont)

If you still remember the concept of ‘range’ from statistics, then you will be able to write down:

	col 1	col 2	col 3
max	3	6	9
min	-5	-4	-3
range = max-min	8	10	12

Performing min-max scaling is similar to

$$\left(\begin{bmatrix} -5 & -4 & -3 \\ -2 & -1 & 0 \\ 1 & 2 & 3 \\ 3 & 6 & 9 \end{bmatrix} - \begin{bmatrix} -5 & -4 & -3 \\ -5 & -4 & -3 \\ -5 & -4 & -3 \\ -5 & -4 & -3 \end{bmatrix}\right)./\begin{bmatrix} 8 & 10 & 12 \\ 8 & 10 & 12 \\ 8 & 10 & 12 \\ 8 & 10 & 12 \end{bmatrix}$$

Matrix Arithmetics (cont)

Note the ./ stands for ‘elementwise division’.
The final result of min-max scaling is

$$\begin{bmatrix} 0. & 0. & 0. \\ 0.375 & 0.3 & 0.25 \\ 0.75 & 0.6 & 0.5 \\ 1. & 1. & 1. \end{bmatrix}$$

It can be obtained using what we have learned:

```
A = np.vstack((np.r_[-5:4].reshape((3,3)), [3,6,9]))
AColumnMin = np.ones((4,1)) @ np.array([-5,-4,-3]).reshape((1,3))
ARange = np.ones((4,1)) @ np.array([8,10,12]).reshape((1,3))
scaleA = (A - AColumnMin) / ARange
```

Matrix Arithmetics (cont)

It would be too painful if we were to write like this!

There is a simplify form as follows:

```
scaleA = (A - [-5,-4,-3]) / [8, 10, 12]
```

But how can this be???

A is 4×3 matrix while $[-5, -4, -3]$ is 1-D array with 3 elements?

Answer: Numpy will treat A as **three** 4×1 matrix when [-5,-4,-3] has **three** elements.

This won't work when we try A - [1,2,3,4].

Matrix Arithmetics (cont)

Question: If we want to scale the rows instead of the columns? What can we do?
We can use the tedious method:

$$\left(\begin{bmatrix} -5 & -4 & -3 \\ -2 & -1 & 0 \\ 1 & 2 & 3 \\ 3 & 6 & 9 \end{bmatrix} - \begin{bmatrix} -5 & -5 & -5 \\ -2 & -2 & -2 \\ 1 & 1 & 1 \\ 3 & 3 & 3 \end{bmatrix}\right) ./ \begin{bmatrix} 2 & 2 & 2 \\ 2 & 2 & 2 \\ 2 & 2 & 2 \\ 6 & 6 & 6 \end{bmatrix}$$

```
A = np.vstack((np.r_[-5:4].reshape((3,3)), [3,6,9]))
ARowMin = np.array([-5,-2,1,3]).reshape((4,1)) @ np.ones((1,3))
ARowRange = np.array([2,2,2,6]).reshape((4,1)) @ np.ones((1,3))
scaleARow = (A - ARowMin) / ARowRange
```

Matrix Arithmetics (cont)

Or we can use “transpose” to do it:

```
scaleARow = ((A.T - [-5,-2,1,3]) / [2,2,2,6]).T
```

Or we can transform [-5,-2,1,3] and [2,2,2,6] to 1×4 matrices. The Numpy will automatically match 3×4 to 1×4 and now turn A to **four** 3×1 row vector to subtract each element from the 1×4 column vector.

```
scaleARow = (A - np.array([-5,-2,1,3]).reshape((4,1))) / \
                 np.array([2,2,2,6]).reshape((4,1))
```

All of them gives us the final answer: $\begin{bmatrix} 0 & 0.5 & 1 \\ 0 & 0.5 & 1 \\ 0 & 0.5 & 1 \\ 0 & 0.5 & 1 \end{bmatrix}$

Outline

Functions for Arrays

It is sometimes confusing when there are many functions but we just have to bear with them:

- Elementwise function operations: mentioned earlier, works with any n-D array of any shape returning $[f(a_{i,j,\ldots,k})]$.
- Statistical functions: A.sum(), A.sum(axis=0), A.sum(axis=1), A.min(), A.min(0), A.min(1) A.max(), A.max(0), A.max(1), A.mean(), A.mean(0), A.mean(1), A.std(), A.std(0), etc.
- Matrix functions: It ONLY works with square matrices. They are available from scipy.linalg and to reduce confusion, they are marked at the end with 'm', e.g. expm(), sinm(), etc.

Statistical Functions

min, max, mean, std (population standard deviation by default, we can change it to sample standard deviation using ddof=1), var, diagonal, trace (sum of the diagonal), sum, cumsum, prod cumprod, etc. are some of the basic statistical functions.

For the matrix $A = \begin{bmatrix} -5 & -4 & -3 \\ -2 & -1 & 0 \\ 1 & 2 & 3 \\ 3 & 6 & 9 \end{bmatrix}$,

using statistical functions, the min-max scaling discussed earlier can be rewritten as

```
scaleARow = (A - A.min(0)) / (A.max(0) - A.min(0))
```

Statistical Functions (cont)

Standard scaling / standardisation:

$$\frac{\text{column } i - \text{mean of column } i}{\text{standard deviation of column } i}.$$

It is not difficult to derive the Python command as

```
>>> A = np.vstack((np.r_[-5:4].reshape((3,3)), [3,6,9]))
>>> stdA = (A - A.mean(0)) / A.std(0)
array([[-1.40213637, -1.28390102, -1.18321596],
       [-0.41239305, -0.47301616, -0.50709255],
       [ 0.57735027,  0.33786869,  0.16903085],
       [ 1.23717915,  1.41904849,  1.52127766]])
```

Statistical Functions (cont)

If you expand `(A - A.mean(0)) / A.std(0)`, you will find that Python is actually performing the elementwise operations below:

$$\left(\begin{bmatrix}-5 & -4 & -3\\ -2 & -1 & 0\\ 1 & 2 & 3\\ 3 & 6 & 9\end{bmatrix} - \begin{bmatrix}-0.75 & 0.75 & 2.25\\ -0.75 & 0.75 & 2.25\\ -0.75 & 0.75 & 2.25\\ -0.75 & 0.75 & 2.25\end{bmatrix}\right)./\begin{bmatrix}3.0311 & 3.6997 & 4.4371\\ 3.0311 & 3.6997 & 4.4371\\ 3.0311 & 3.6997 & 4.4371\\ 3.0311 & 3.6997 & 4.4371\end{bmatrix}$$

Matrix Functions

If we want to talk about matrix functions, we need to talk about the *n*th power of a square matrix:

$$A^0 = I_n, \quad A^1 = A, \quad A^n = \underbrace{AA\ldots A}_{n \text{ times}}, \quad n \geq 2.$$

Consider $A = \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}$.

$$A^2 = AA = \begin{bmatrix} 7 & 10 \\ 15 & 22 \end{bmatrix}, \quad A^3 = (AA)A = \begin{bmatrix} 37 & 54 \\ 81 & 118 \end{bmatrix}$$

`np.linalg.matrix_power(A, n)` is the Numpy function for calculating the power of a matrix.

Matrix Functions (cont)

Mathematicians generalise the Taylor series of a function f at $x = 0$

$$f(x) = \sum_{k=0}^{\infty} \frac{f^{(k)}(0)}{k!} x^k,$$

to obtain the **matrix function**:

$$f^{[m]}(A) = \sum_{k=0}^{\infty} \frac{f^{(k)}(0)}{k!} A^k.$$

The usual exponential, logarithm, trigonometric and hyperbolic functions can be generalised to become matrix functions (available in scipy.linalg).

Matrix Functions (cont)

Looking up in a Calculus book, we can find

$$exp(x) = \sum_{k=0}^{\infty} \frac{1}{k!} x^k,$$

So the matrix exponential function for an $n \times n$ matrix A is

$$\exp^{[m]}(A) = \sum_{k=0}^{\infty} \frac{1}{k!} A^k$$

When A is a zero matrix O, we obtain

$$\exp^{[m]}(O) = \frac{1}{0!} I_n + \frac{1}{1!} O + \frac{1}{2!} O + \ldots = I_n.$$

Matrix Functions (cont)

- expm(A) (Pade approximation), logm(A[, disp])
- cosm(A), sinm(A), tanm(A)
- coshm(A), sinhm(A), tanhm(A)
- signm(A[, disp]): Matrix sign function.
- sqrtm(A[, disp, blocksize])
- funm(A, func[, disp]): Evaluate a matrix function specified by a callable.
- expm_frechet(A, E[, method, compute_expm, ...]): Frechet derivative of the matrix exponential of A in the direction E.
- expm_cond(A[, check_finite])
- fractional_matrix_power(A, t)

Matrix Functions (cont)

Analytic matrix functions are related to eigenvalue problems which is beyong this course.

However, it is interesting to investigate the generalisation of the function $f(x) = \frac{1}{x}, \quad x \neq 0$ to the matrix function

$$f^{[m]}(A) = A^{-1}, \quad \det A \neq 0.$$

The A^{-1} is called the inverse matrix of $n \times n$ matrix A and can be used to solve the square matrix problem:

$$AX = B \tag{1}$$

where $X = A^{-1}B$ (when $\det A \neq 0$) and B are $n \times k$ matrices.

Linear Algebra Functions

Matrix functions are part of the larger category of "linear algebra functions" in Python.

According `https://docs.scipy.org/doc/scipy/reference/tutorial/linalg.html`, Scipy is normally built using the optimised LAPACK and BLAS libraries and it has very fast linear algebra capabilities and contains all the functions in `numpy.linalg`. Therefore, we will be using Scipy instead of Numpy if we want to solve the square matrix problem (1).

```
from scipy import linalg
```

Linear Algebra

A few linear algebra functions are also available in `np.linalg`, which are able to compute results for several matrices at once, if they are **stacked** into the same array. This is indicated in the documentation via input parameter specifications such as `A: (..., M, M) array_like`. This means that if for instance given an input array `A.shape == (N, M, M)`, it is interpreted as a “stack” of `N` matrices, each of size `M`-by-`M`.

Linear Algebra (cont)

Similar specification applies to return values, for instance the determinant takes a 'stack' of $m \times m$ matrices with the shape $k \times ... \times \ell \times m \times m$ and returns an array of the shape $k \times ... \times \ell$

Many linear algebra functions from np.linalg works on higher-dimensional arrays in a similar fashion: the last 1 or 2 dimensions of a multidimensional array are interpreted as vectors or matrices, as appropriate for each operation.

Linear Algebra (cont)

`scipy.linalg` linear algebra solvers and inverses:

- `linalg.solve(A, B)`: Solves $AX = B$.
- `linalg.inv(A)`: Compute the (multiplicative) inverse of a matrix A. It is the same as solving $AX = B$ with B being the identity matrix.
- `linalg.lstsq(A, B)`: Return the least-squares solution X to $\min_X \|AX = B\|_2$.
- `linalg.pinv(A)`: Compute the (Moore-Penrose) pseudo-inverse of a matrix.
- Solves special matrices: linalg.solve_circulant(C,B), linalg.solve_toeplitz(T,B), linalg.solve_triangular(U,B).

Linear Algebra (cont)

`scipy.linalg` linear algebra solvers (cont):

- linalg.solve_sylvester(A, B, q): solves for X to the Sylvester equation $AX + XB = Q$.
- linalg.solve_continuous_are(A, B, Q, R[, e, s, ...]): Solves the continuous-time algebraic Riccati equation (ARE) $XA + A^H X - XBR^{-1}B^H X + Q = 0$.
- linalg.solve_discrete_are(A, B, Q, R[, e, s, balanced]) Solves the discrete-time ARE.
- linalg.solve_continuous_lyapunov(A, Q): Solves the continuous Lyapunov equation $AX + XA^H = Q$.
- linalg.solve_discrete_lyapunov(A, Q[, method]) Solves the discrete Lyapunov equation $AXA^H - X + Q = 0$

Example: Sept 2014 FE, Q4(a)

Given the linear system

$$
\begin{aligned}
3x_1 + 7x_2 - 2x_3 + 3x_4 - x_5 &= 37 \\
4x_1 + 3x_5 &= 40 \\
5x_3 - 4x_4 + x_5 &= 12 \\
2x_1 + 9x_3 + 4x_4 + 3x_5 &= 14 \\
5x_4 + 8x_5 &= 20
\end{aligned}
$$

Write a Python command to solve the linear system.
(8 marks)

Example: Sept 2014 FE, Q4(a) cont

Sample Answer:

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

x =
array([ 23.67072111, -12.36837533, 32.57688966,
        33.16420504, -18.22762815])
```

Example: SPM Forecast Question

It is given that matrix M is a 2×2 matrix such that

$$M\begin{pmatrix} -2 & 1 \\ 1 & 3 \end{pmatrix} = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$$

Use Python to

1. find matrix M;
2. calculate the value of x and of y for the following simultaneous linear equations

$$\begin{aligned} -2x + y &= 10, \\ x + 3y &= 9. \end{aligned}$$

Example: SPM (cont)

Using SPM linear algebra, we can obtain easily obtain

$$M = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} \begin{pmatrix} -2 & 1 \\ 1 & 3 \end{pmatrix}^{-1} = \frac{1}{-6-1} \begin{bmatrix} 3 & -1 \\ -1 & -2 \end{bmatrix}$$

Using Python for part (1):

```
import numpy as np
A = np.array([[-2, 1], [1, 3]])
from scipy import linalg
M = linalg.inv(A)
```

The matrix M is

```
array([[-0.42857143,  0.14285714],
       [ 0.14285714,  0.28571429]])
```

Example: SPM (cont)

There are two methods for using Python in part (2):
Method 1:

```
X = M @ [10, 9]  # Answer: [-3, 4]
x = X[0]
y = X[1]
```

Method 2:

```
X = linalg.solve(A, [10,9])
x, y = X
```

Example: Oct 2018 FE, Q2(c)

Given a Python function `myst` as follows:

```
def myst(a):
    b = a.copy()
    n = a.size
    for i in range(int(n/2)):
        b[i],b[n-i-1] = b[n-i-1],b[i]
    return b
```

After you have imported the function `myst` and run `myst(np.array([1,2,3, 5,7,9]))`, what return value will you obtain? Explain in *one sentence* what the function `myst` does? (5 marks)

Example: Oct 2018 FE, Q2(c) cont

To answer the question without running it on a computer, one will need to understand what does the following command do:

```
b[i],b[n-i-1] = b[n-i-1],b[i]
```

In Python, we usually use the ‘comma’ to exchange values!!!

```
x = 3; y = 4
x, y = y, x
print("x =",x, "; y =", y)
```

Sample Answer:

- We will obtain [9,7,5, 3,2,1]
- `myst` reverses the Numpy array that we supply.

Final Exam Oct 2020, Q1(b)

Given that three 3×3 matrices $P = \begin{bmatrix} 5 & 8 & 8 \\ 6 & -9 & -8 \\ 6 & -5 & 1 \end{bmatrix}$,

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

(i) Write down the Python command to find the inverse matrix of Q, Q^{-1}. (0.5 mark)

Final Exam Oct 2020, Q1(b)

Answer (Assuming linalg is imported from scipy)

The Python command to find Q^{-1} is
`linalg.inv(Q)` or
`linalg.solve(Q,np.eye(Q.shape[0]))`.
The output is .... [0.5 mark]

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

Final Exam Oct 2020, Q1(b) cont

(ii) Write down the Python command to find matrix L if $P^3 LQ = R$. Write down the **matrix** L. (1 mark)

Answer

$L = (P^3)^{-1} RQ^{-1}$ ....[0.7 mark]

```
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

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

[0.3 mark]

Final Exam Oct 2020, Q1(b) cont

(iii) Suppose the 3×3 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

$$(R - QP^{-1}Q)^{-1} = H. \qquad \text{(1 mark)}$$

Final Exam Oct 2020, Q1(b) cont

Answer

`np.hstack((np.vstack((P,Q)),np.vstack((Q,R))))` is used to obtain

$$H = \begin{bmatrix} 0.26819736 & -0.15480007 & -0.07891041 \\ 0.69421553 & -0.3532685 & -0.42828374 \\ 1.00692917 & -0.48176952 & -0.51330189 \end{bmatrix} \qquad [0.6 \text{ mark}]$$

The Python command is

`linalg.inv(R - Q@linalg.inv(P)@Q)`. ....[0.4 mark]

In general,

$$\begin{bmatrix} \mathbf{A} & \mathbf{B} \\ \mathbf{C} & \mathbf{D} \end{bmatrix}^{-1} = \begin{bmatrix} \mathbf{A}^{-1} + \mathbf{A}^{-1}\mathbf{B}\left(\mathbf{D} - \mathbf{C}\mathbf{A}^{-1}\mathbf{B}\right)^{-1}\mathbf{C}\mathbf{A}^{-1} & -\mathbf{A}^{-1}\mathbf{B}\left(\mathbf{D} - \mathbf{C}\mathbf{A}^{-1}\mathbf{B}\right)^{-1} \\ -\left(\mathbf{D} - \mathbf{C}\mathbf{A}^{-1}\mathbf{B}\right)^{-1}\mathbf{C}\mathbf{A}^{-1} & \left(\mathbf{D} - \mathbf{C}\mathbf{A}^{-1}\mathbf{B}\right)^{-1} \end{bmatrix}$$

Outline

Relational Operations

As mentioned earlier, when A and B are arrays of same shape, we can extend the relational operations of numbers in an elementwise manner to arrays by the definitions below.

`A==B`	$[a_{i,j,\ldots,k} = b_{i,j,\ldots,k}]$	`A!=B`	$[a_{i,j,\ldots,k} \neq b_{i,j,\ldots,k}]$
`A<B`	$[a_{i,j,\ldots,k} < b_{i,j,\ldots,k}]$	`A<=B`	$[a_{i,j,\ldots,k} \leq b_{i,j,\ldots,k}]$
`A>B`	$[a_{i,j,\ldots,k} > b_{i,j,\ldots,k}]$	`A>=B`	$[a_{i,j,\ldots,k} \geq b_{i,j,\ldots,k}]$

Note that the above relational operations can extended to an array and a number. For example, if c is a real number, then

- `A<c`: $[a_{i,j,\ldots,k} < c]$, etc.

Relational Operations (cont)

An important use of the elementwise relational operations is to perform counting. A typical scenario would be “count how many students in a school is taller than 170cm”. Tall students are usually encouraged to participate in sports, etc.

The testing of level antigen are used to detect virus. It is important to count the level of antigen. The Python command can be something like

```
(Level > threshold).sum()
```

Note that Python will convert True to 1 and False to 0, so sum() will give the correct count!

Logical Operations

For Booleans, we have not True = False, True and False = False, True or False = True, etc.
The elementwise Logical operations for n-D array of the same shape are defined as

- Negation: `~A` which means $[\text{not } a_{i,j,\ldots,k}]$
- Conjunction: `A&B` which means $[a_{i,j,\ldots,k} \text{ and } b_{i,j,\ldots,k}]$
- Disjunction: `A|B` which means $[a_{i,j,\ldots,k} \text{ or } b_{i,j,\ldots,k}]$

Applications:

- `~ ((A<0) | (A>100))`
- `(0<=A) & (A<=100)`

Boolean Indexing

The "Boolean" array for an array A generated with the use of relational operations 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 "filter" out wanted or unwanted data in an array. In the following, we show examples of the applications of Boolean indexing.

Note that 'Boolean indexing' has variation in SQL and other programming languages as 'select', 'filter', etc.

Boolean Indexing Example

The test 2 results of UECM3033 Numerical Methods for the Jan 2018 semester are

11.5, 15.1, 10.8, 14.1, 5.8, 4.1, 15.7, 13.3, 14.6, 5.2, 13.1, 8.6, 8.8, 16.3, 11.7, 13.9, 13.6, 14.5, 11, 14, 13.7, 16.1, 12.1, 9.7, 14.9, 12, 10.5, 12.8, 15.3, 4.8, 13.1, 0, 12.5, 8.8, 14.4, 12.7, 8.8, 11.9, 13.1, 14.4, 7.3, 17.1, 9.3, 11, 13.5, 9, 7.9, 4.7, 13.8, 15.5, 13.2, 8.8, 10.1, 13.3, 9.5, 10.5, 12.6, 14.6, 12.8, 11, 2.7, 6.2, 10.6, 14.5, 13.4, 10.5, 11.3, 14.8, 9.9, 8.8, 14.2, 9.7, 9.4, 9, 13.5, 10.3

The full mark for test 2 is 20 marks and the passing mark is 10. Find all those marks which is below 10. How many students fail?

Answer:

```
M=np.array([11.5,...])
below10 = M[M<10]
fails = (M<10).sum()  # below10.shape[0]
```

Example: Sept 2013 FE, Q1(b)

Write a Python script to perform the following actions:

- Generate a 2-by-3 array of random numbers using `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.

Analysis

The intention of the lecturer who set the question is to ask you to express this in for loop but in reality, but everyone just use the array functions to achieve the stated requirements.

Example: Sept 2013 FE, Q1(b) cont

Sample answer:

```
M = np.random.rand(6).reshape((2,3)) # item 1
M[M<0.2] = 0      # item 2
M[M>=0.2] = 1     # item 2
```

The previous lecturer wanted the following answer using loop:

```
M = np.random.rand(6).reshape((2,3)) # item 1
for i in range(2):
    for j in range(3):
        M[i,j] = 0 if M[i,j] < 0.2 else 1
```

Example

Suppose you have keyed in an array of the following exam data (out of 30 marks):

10 24 NaN 22 25 17 23

The “NaN” indicates that a student is absent. Find the average and standard deviation by filtering “NaN”.

This question is a bit challenging because the following answers are WRONG!!!

```
a = np.array([10,24,np.nan,22,25,17,23])
np.mean(a); np.std(a)
np.mean(a[a!=np.nan]); np.std(a[a!=np.nan])
```

The correct answer is

```
np.mean(a[~np.isnan(a)]);np.std(a[~np.isnan(a)])
```

Example related Number Theory

Extract from the array `B= 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.

Example related Number Theory (cont)

To answer this question, one needs to know number theory: A number a is divisible by b is $a = bq$ where a, b and q are all integers. (Are they taught in SPM???)

If a is **not** divisible by b, then $a = bq + r$ where r is some **non-zero integer**.

Recall the symbol % from Topic 1: `a % b` gives `r`!
So numbers not divisible by 3 means: `a % 3` ≠ `0`

Example related Number Theory (cont)

Sample Answer:

```
B = np.array([3,4,6,10,24,89,45,43,46,99,100])
B[B % 3 != 0]
B[B % 5 == 0]
B[(B % 3 == 0) & (B % 5 == 0)]
B[B % 3 == 0] = 42
```