UECM1703 Introduction to Scientific Computing Practical 2: Advanced Array Programming

Dr Liew How Hui

Oct 2021

Outline

Revision on Weeks 1&2 (for Test)

- know the basic numeric data types (floats, integers, complex numbers), their arithmetics and the functions in math (and cmath). E.g. translating $\frac{\sqrt{2}+e^{\pi i}}{\sqrt{2}-e^{\pi i}}$ to Python and vice vesa.
- Defining function and use scipy algorithms
- handling strings and formatting numbers (essential in Python scripting). An example is given here:
  `https://github.com/jonasbostoen/simple-neural-network/blob/master/main.py`
  - Input: stringval = input("Enter a value: ")
  - Convert strings to numeric data types: a = float(stringval), b = int(stringval), c = complex(stringval)
  - Output: print("a:10.6f".format(a=a,...)), tripple single/double quotes for multiline strings.

Time: 8 mins.

Revision on Weeks 1&2 (cont)

Assumption:

- import numpy as np
- from scipy import linalg

Important array programming skills:

- Array recognition and construction. From Practical 1:

$$A = \begin{bmatrix} 2 & 0 & 0 & 0 \\ 0 & 2 & 0 & 0 \\ 0 & 0 & 2 & 0 \\ 0 & 0 & 0 & 2 \end{bmatrix}, \; B = \begin{bmatrix} 0 & 0 \\ 0 & 0 \\ 0 & 0 \\ 0 & 0 \end{bmatrix}, \; C = \begin{bmatrix} 10 & 10 & 10 \\ 10 & 10 & 10 \\ 10 & 10 & 10 \\ 10 & 10 & 10 \end{bmatrix}$$

  - For A: np.diag([2,2,2,2]), np.diag([2]*4), np.eye(4)*2, linalg.circulant[2,0,0,0])
  - For B: np.zeros((4,2)), np.ones((4,2))*0, np.full((4,2),0)
  - For C: np.full((4,3),10), np.ones((4,3))*10

Time: 8 mins.

Revision on Weeks 1&2 (cont)

Array recognition and construction (cont)

- Arithmetic sequence: $a, a+d, ..., a+(n-1)d \rightarrow$ np.arange(a, a+n*d, d), np.r_[a:a+n*d:d]
- Geometric sequence: $a, ar, ar^2, ..., ar^{n-1} \rightarrow$ a*r**np.arange(n), a*r**np.r_[:n]
- A.reshape((m1, ..., mk)), A.ravel(), A.flatten()
- Special matrices: Practical 1 example

$$D = \begin{bmatrix} 3 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 4 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 3 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 4 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 3 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 4 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 3 \end{bmatrix} E = \begin{bmatrix} 3 & -2 & 0 & 0 & 0 & 0 & 0 \\ -2 & 4 & -2 & 0 & 0 & 0 & 0 \\ 0 & -2 & 3 & -2 & 0 & 0 & 0 \\ 0 & 0 & -2 & 4 & -2 & 0 & 0 \\ 0 & 0 & 0 & -2 & 3 & -2 & 0 \\ 0 & 0 & 0 & 0 & -2 & 4 & -2 \\ 0 & 0 & 0 & 0 & 0 & -2 & 3 \end{bmatrix}$$

Answer: See the next slide

Revision on Weeks 1&2 (cont)

Array recognition and construction (cont)

- Answer:
  - D: np.diag([3,4,3,4,3,4,3]), np.diag([3,4]*3+[3])
  - E: elementwise operations
    - D + np.diag([-2]*6,1) + np.diag([-2]*6,-1) (Mentioned in week 9)
    - D + linalg.toeplitz([0,-2,0,0,0,0,0],[0,-2,0,0,0,0,0])
  - E: array indexing (most steps)
    - E = np.diag([3,4]*3+[3])
    - E[np.r_[1:7], np.r_[:6]] = -2
    - E[np.r_[:6], np.r_[1:7]] = -2
- Block matrices: np.vstack((np.hstack((A,B)), np.hstack((C,D))))

Time: 15 mins.

Revision on Weeks 1&2 (cont)

- Array construction .... ✓
- Array comparison: `np.array_equal(A,B)` ...... ✓
- Array indexing !!!
- Array arithmetic (we have made used of them in the **array construction**): elementwise (+, -, *, /) @ (matrix multiplication) → array of numbers (floats, integers, Booleans) .... ✓
- Array functions !!!
- Elementwise relational operations: `==`, `<=`, `<`, `>=`, `>=` → array of Booleans .... ✓
- Some if-else and for statements (E.g. Oct 2020 Test & Exam). .... ✓

Time: 5 mins.

Revision on Weeks 1&2 (cont)

Array indexing:

- start:stop:step indexing (::-indexing): A[1:], A[:4], A[1:4], A[1:9:2], A[9:-1:-1], A[:,2], ...
- list indexing: A[[1,2,3],[4,5,6]] is the same as np.array([A[1,4],A[2,5],[A[3,6]])
- list indexing mixed with ::-indexing): A[:, [3,2,3,4,3,5]]
- negative indexing: A[:,-1] (last column), A[:,-2] (second last column)
- Boolean indexing: A[elementwise_predicate(A)], e.g. `A[A<0]`, `A[A % 3 == 0]`. In general, A[Boolean matrix of appropriate shape]. E.g. `A[(x-x0)**2 + (y-y0)**2 < 20] = ...` (demo in Session 4)

Time: 8 mins.

Revision on Weeks 1&2 (cont)

Array functions:

- Elementwise array functions: np.exp, np.sin, np.cos, ...
- Matrix functions (only for 2D array!): linalg.expm, ...
- Statistical functions: A.mean(), A.mean(axis=0), A.std(axis=1), ...

Applications:

- Generating array for drawing graphs, e.g. x=np.r_[0:2*np.pi:0.001], y=np.sin(x)
- Scaling along the columns/rows
- Checking properties of matrix: E.g. Final Exam Qct 2020 Q1(a)'s matrix's diagonally dominant properties can be checked with `np.all(A.diagonal() > np.abs(A - A.diagonal()).sum(1))`.

Time: 8 mins. (8 mins break / Q&A)

Outline

Application 1 of 2-D Array: Oct 2020 Test Question (Vandermonde matrix)

You are trying to use a polynomial $y = a_0 + a_1 x + a_2 x^2 + a_3 x^3 + a_4 x^4$ to fit the following 2D data points:

$$(145, 7),\ (155, 17),\ (165, 32),\ (175, 51),\ (180, 60).$$

This will lead to the following system of linear equations:

$$a_0 + 145 a_1 + 145^2 a_2 + 145^3 a_3 + 145^4 a_4 = 7$$
$$a_0 + 155 a_1 + 155^2 a_2 + 155^3 a_3 + 155^4 a_4 = 17$$
$$a_0 + 165 a_1 + 165^2 a_2 + 165^3 a_3 + 165^4 a_4 = 32$$
$$a_0 + 175 a_1 + 175^2 a_2 + 175^3 a_3 + 175^4 a_4 = 51$$
$$a_0 + 185 a_1 + 185^2 a_2 + 185^3 a_3 + 185^4 a_4 = 60$$

Oct 2020 Test Question (cont)

which can be transformed into a matrix form:

$$A\boldsymbol{a} = \begin{bmatrix} 1 & 145 & 145^2 & 145^3 & 145^4 \\ 1 & 155 & 155^2 & 155^3 & 155^4 \\ 1 & 165 & 165^2 & 165^3 & 165^4 \\ 1 & 175 & 175^2 & 175^3 & 175^4 \\ 1 & 185 & 185^2 & 185^3 & 185^4 \end{bmatrix} \begin{bmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \\ a_4 \end{bmatrix} = \begin{bmatrix} 7 \\ 17 \\ 32 \\ 51 \\ 60 \end{bmatrix} = \boldsymbol{b}.$$

where A is the 5×5 matrix and $\boldsymbol{b}$ is the y-values.

Oct 2020 Test Question (cont)

1. Construct the matrix A (using for loop).

Answer

A sample implementation is shown below. An implementation without using for loops will receive mark deduction.

```
import numpy as np
N = 5
A = np.ones((N,N))
cs = [145., 155., 165., 175., 185.]
b = [7,17,32,51,60]
for i in range(N):
    for j in range(1,N):
        A[i,j] = cs[i]**j
print(A)
```

Oct 2020 Test Question (cont)

1. Construct the matrix A (cont).

Answer (cont)

- Appropriate imports .... [0.5 mark]
- Appropriate initialisation .... [1 mark]
- Correct for loop .... [1 mark]

Without for loop: `np.vander([145., 155., 165., 175., 185.])[:,::-1]`

Oct 2020 Test Question (cont)

2. Find the determinant of the matrix A by writing down both the Python command and the result.
(1 mark)

Answer

np.linalg.det(A) .... [0.5 mark]
2879999999999.994 .... [0.5 mark]

Oct 2020 Test Question (cont)

3. Solve $A\boldsymbol{a} = \boldsymbol{b}$ where $\boldsymbol{a}$ is the vector of the unknown coefficients a_0, a_1, a_2, a_3, a_4. (1 mark)

Answer

np.linalg.solve(A,[7,17,32,51,60]) . . . . [0.5 mark]
$a_0 = -3.41103672e+04$, $a_1 = 8.64637500e+02$,
$a_2 = -8.20395833e+00$, $a_3 = 3.45000000e-02$,
$a_4 = -5.41666667e-05$ . . . . [0.5 mark]

Oct 2020 Test Question (cont)

4. Identify the problem of using the polynomial $y = a_0 + a_1 x + a_2 x^2 + a_3 x^3 + a_4 x^4$ to fit the 2D data points and propose a solution to solve the problem you state. [Hint: The answer provided must be relevant to scientific computing.] (0.5 mark)

Answer

The determinant of the matrix is too large. . . [0.2 mark]
A possible solution: Use
$y = a_0 + a_1(x - \bar{x}) + a_2(x - \bar{x})^2 + a_3(x - \bar{x})^3 + a_4(x - \bar{x})^4$
and it is possible to have a matrix with smaller determinant. . . . .[0.3 mark]

Time: 10 mins

Final Exam Oct 2020, Q2(a)

Suppose that the Taylor series of a cosine function at $x = 0$ with the first 11 terms:

$$\cos x = 1 - \frac{x^2}{2!} + \frac{x^4}{4!} - \frac{x^6}{6!} + \cdots + (-1)^{10}\frac{x^{20}}{(20)!}$$

is going to be used in an embedded system.

1. Write a script containing the function `mycosfor(x,n=10)` which implements the cosine series **using for loops** without importing anything from Python modules. If you import anything from any Python module (e.g. `math`, `numpy`), marks will be deducted. (1.5 marks)

Not coming out again in 2021 — fast forward.

Final Exam Oct 2020, Q2(a) cont

1

Answer

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

```
def mycosfor(x,n=10):
    val = 1.0
    for k in range(1,n+1):
        fac = 1.0
        for i in range(2,2*k+1):
            fac *= float(i)
        val += (-1)**k*x**(2*k)/fac
    return val
```

Purpose: Test the knowledge of loop .... [1.5 marks]

Final Exam Oct 2020, Q2(a) cont

2. Write a script containing the function `mycosrec(x,n=10)` which implements the cosine series **using recursive function** without importing anything from Python modules. If you import anything from any Python module (e.g. `math`, `numpy`) or use a for loop, marks will be deducted.

   (1.5 marks)

Sample Answer .... [1.5 marks]

```python
def mycosrec(x,n=10):
    def cos_aux(n0, x):
        if n0 > n: return 1.0
        return 1-x*x/(2*n0)/(2*n0-1) * cos_aux(n0+1, x)
    return cos_aux(1,x)
```

Mentioned in Oct 2020 Slide

Final Exam Sept 2015, Q2(d)

The Taylor series of a cosine function at $x = 0$ can be rewritten as

$$\cos x = 1 - \frac{x^2}{2!}\left[1 - \frac{x^2}{4\times 3}\left[1 - \frac{x^2}{6\times 5}\left[1 - \cdots\right]\right]\right].$$

Write a recursive function that computes and approximation of cosine.

```
def mycos(x,n=16):
    def cos_aux(n0, x):
        if n0 > n: return 1.0
        return 1-x*x/(2*n0)/(2*n0-1) * cos_aux(n0+1, x)
    return cos_aux(1,x)
```

Lesson: Some equivalent formulation are more efficient in computer calculation.

Final Exam Oct 2020, Q2(a) cont

3. Write a script which imports the functions `mycosfor(x,n=10)` `mycosrec (x,n=10)` to calculate their values at $x = -1, 0, 1, 2, \ldots, 10$ and their absolute difference errors with the standard numpy implementation `np.cos`. Your script should output the following text:

```
   x    mycosfor    abs.err   mycosrec     abs.err
-1.0   0.5403023 1.1102e-16  0.5403023           0
 0.0   1.0000000          0  1.0000000           0
 1.0   0.5403023 1.1102e-16  0.5403023           0
 2.0  -0.4161468 3.6637e-15 -0.4161468  3.6082e-15
 3.0  -0.9899925  2.747e-11 -0.9899925   2.747e-11
 4.0  -0.6536436 1.5209e-08 -0.6536436  1.5209e-08
 5.0   0.2836642 2.0287e-06  0.2836642  2.0287e-06
 6.0   0.9602802 0.00010987  0.9602802  0.00010987
 7.0   0.7570938  0.0031916  0.7570938   0.0031916
 8.0  -0.0867742   0.058726 -0.0867742    0.058726
 9.0  -0.1491107    0.76202 -0.1491107     0.76202
10.0   6.6645643     7.5036  6.6645643      7.5036
```

Give one reason why the recursive implementation is better compared to the for loop implementation. (2 marks)

Final Exam Oct 2020, Q2(a) cont

3

Sample Answer

```
import numpy as np
mycosrec = np.vectorize(mycosrec)  # part (i)
mycosfor = np.vectorize(mycosfor)  # part (ii)
a, b, h = -1.0, 10.0, 1.0
N = round((b-a)/h)
x = np.linspace(a,b,N+1)
y  = np.cos(x)
y2 = mycosrec(x)
d2 = abs(y2-y)
y1 = mycosfor(x)
d1 = abs(y1-y)
print("{:>4s} {:>10s} {:>10s} {:>10s} {:>10s}".format("x",
        "mycosfor", "abs.err", "mycosrec", "abs.err"))
for i in range(len(x)):
    print("{:4.1f} {:10.7f} {:10.5g}".format(x[i], y1[i], d1[i]) +
          " {:10.7f} {:10.5g}".format(y2[i], d2[i]))
```

.... [1.5 marks]

Final Exam Oct 2020, Q2(a) cont

3.

Sample Answer (cont)

One reason why the recursive implementation is better than the loop implementation is the reduction in multiplication. The loop implementation is $O(n^2)$ while the recursive implementation is $O(n)$. ....[0.5 mark]

Time: 5 mins

A Little 'Array' Statististics

Consider the matrix

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

There two basic 'relations' between **each row**:

- Pearson correlation ('scaled' covariance matrix): $\text{cor}(\boldsymbol{x}, \boldsymbol{y}) = \frac{(\boldsymbol{x}-\mu_x)\cdot(\boldsymbol{y}-\mu_y)}{n\sigma_{\boldsymbol{x}}\sigma_{\boldsymbol{y}}}$

```
i, j = 0,1; n=len(A[i])  # compare row i & row j
np.dot(A[i]-A[i].mean(),A[j]-A[j].mean())/n/A[i].std()/A[j].std()
#Or: np.corrcoeff(A)  # same as the scaling mentioned in Topic 2
#( (A.T-A.mean(1))/A.std(1) ).T @ ((A.T-A.mean(1))/A.std(1)) / n
```

- Cosine similarity: $\text{cos.sim}(\boldsymbol{x}, \boldsymbol{y}) = 1 - \frac{\boldsymbol{x}\cdot\boldsymbol{y}}{\|\boldsymbol{x}\|\|\boldsymbol{y}\|}$

```
i, j = 0,1  # compare row i & row j
1 - np.dot(A[i], A[j])/np.linalg.norm(A[i])/np.linalg.norm(A[j])
#Or: scipy.spatial.distance.cdist(A,A,'cosine')
```

Time: 5 mins (not coming out in test)

Toeplitz Matrix

It arises from the the 'convolution' of 1D arrays $h[n]$ and $x[n]$:

$$y[n] = (h * x)[n] = \sum_{i=-\infty}^{\infty} h[n-i]x[i]$$

in the response $y[n]$ of the Linear Time Invariant system with an impulse response $h[n]$ and an input sequence $x[n]$ (which is 0 when $n < 0$).

Toeplitz Matrix (cont)

For example, if $h[0] = 3$, $h[1] = 2$, $h[2] = 5$, $h[3] = 7$, $h[n] = 0$ for $n \neq 0, 1, 2, 3$. Then

- $y[0] = \sum_{i=-\infty}^{\infty} h[-i]x[i] = h[0]x[0]$
- $y[1] = \sum_{i=-\infty}^{\infty} h[1-i]x[i] = h[1]x[0] + h[0]x[1]$
- $y[2] = h[2]x[0] + h[1]x[1] + h[0]x[2]$
- $y[3] = h[3]x[0] + h[2]x[1] + h[1]x[2] + h[0]x[3]$
- $y[4] = h[4]x[0] + h[3]x[1] + h[2]x[2] + h[1]x[3]$ $+h[0]x[4]$

Note that $h[4] = 0$.

Toeplitz Matrix (cont)

This leads to the following matrix representation:

$$\begin{bmatrix} y[0] \\ y[1] \\ y[2] \\ y[3] \\ y[4] \end{bmatrix} = \begin{bmatrix} h[0] & 0 & 0 & 0 & 0 \\ h[1] & h[0] & 0 & 0 & 0 \\ h[2] & h[1] & h[0] & 0 & 0 \\ h[3] & h[2] & h[1] & h[0] & 0 \\ 0 & h[3] & h[2] & h[1] & h[0] \end{bmatrix} \begin{bmatrix} x[0] \\ x[1] \\ x[2] \\ x[3] \\ x[4] \end{bmatrix}$$

The coefficient matrix in Python is

```
h0=3; h1=2; h2=5; h3=7
H = linalg.toeplitz([h0,h1,h2,h3,0],[h0,0,0,0,0])
x = ...
y = H @ x
```

Time: 5 mins

Matrix Sine Fn vs Sine Fn

Consider numbers $x = 1, 2, \pi$, the corresponding values of the sine function (from math import sin) are

- sin(1) $\Rightarrow$?
- sin(np.pi/2) $\Rightarrow$?
- sin(np.pi) $\Rightarrow$?

Consider 1x1 matrices $x = [1], [2], [\pi]$, the corresponding values of the matrix sine function are

- linalg.sinm(np.array([[1]])) $\Rightarrow$?
- linalg.sinm(np.array([[np.pi/2]])) $\Rightarrow$?
- linalg.sinm(np.array([[np.pi]])) $\Rightarrow$?

Time: 5 mins

Matrix Sine Fn vs Sine Fn (cont)

Consider 2x2 matrices

$$A1 = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}, \quad A2 = \begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix}, \quad A3 = \begin{bmatrix} 1 & \pi/2 \\ \pi/2 & \pi \end{bmatrix}$$

The corresponding values of the **elementsize sine function** are

- np.sin(A1) $\Rightarrow$?; np.sin(A2) $\Rightarrow$?; np.sin(A3) $\Rightarrow$?

The corresponding values of the **matrix sine function** are

- linalg.sinm(A1) $\Rightarrow$?; linalg.sinm(A2) $\Rightarrow$?; linalg.sinm(A3) $\Rightarrow$?

What did you observe?

Time: 10 mins

Truth Table as a 2-D Array

Write a Python script to generate the truth table for the following statement

$$q \wedge \sim (\sim p \rightarrow r).$$

Discrete/Basic Maths Answer

```
for p in [True, False]:
    for q in [True, False]:
        for r in [True, False]:
            statement = q and not (r if not p else True)
            print("{} {} {} | {}".format(str(p)[0],
               str(q)[0],str(r)[0],str(statement)[0]))
```

Truth Table as a 2-D Array (cont)

Answer using Array

```
p = np.array([True]*4 + [False]*4)
q = np.array(([True]*2 + [False]*2)*2)
r = np.array(([True]*1 + [False]*1)*4)
# We need to logical equivalence P -> Q = ~P \/ Q
truthtable = np.vstack((p, q, r, q & ~ ( p | r))).T
print(truthtable)
# Constructing a prettier table
truthtable = pd.DataFrame(truthtable,
  columns=['p','q','r','q /\ ~(~p->r)'])
print(truthtable)
```

Time: 10 mins (10 mins break + Q&A)

Outline

Working with Image Arrays

An image array can be 2-D or 3-D depending on whether it is “grey-scale” or “coloured”. In the case of a coloured image, red, green, blue (and alpha) are required an this means that a $m \times n \times 3$-array (or a $m \times n \times 4$-array) is required. Python supports the loading of images using functions from `imageio` or `matplotlib`. By using `imread` from `matplotlib.pyplot` (which uses `pillow` (PIL) module), typical image types such as Jpeg, Png and Bmp can be read and `imshow` and `imsave` can be used to view and save the image.

Working with Image Arrays (cont)

The module `scipy.ndimage` provides many image array processing functions for *measuring*, *filtering*, *interpolating* and *morphing* a given image. To test and add more functions, newer Scipy package includes two images `ascent` (gray) and `face` (colour).

We will explore the image array manipulation of the two images.

```
>>> from scipy import misc
>>> ascent = misc.ascent()
>>> face = misc.face()
>>> import matplotlib.pyplot as plt
>>> plt.imshow(ascent, cmap=plt.cm.gray)
>>> plt.show()
```

Working with Image Arrays (cont)

1. How do you know that the type of the image `ascent` and its dimension using Python?
2. How "large" is the image `ascent` (by pixels?)
3. Write down the commands to find minimum, maximum and average values of the image `ascent`.
4. Explain what does the following commands do?

```
face2 = face[:-200,200:-50] #image cropping
plt.imshow(face2)
plt.show()    # face[y_axis, x_axis, z_axis]
```

Hint: Negative indexing

Time: 30 minutes

Working with Image Arrays (cont)

Using the array indexing, we are able to 'crop' image (as discuss in the previous slide) and to 'change' colours in an image which will explore now.

- First, note that in 8-bit colour system, the colour ranges from 0 (black) to 255 (R/G/B).
- We colour the top 50 pixels and bottom 50 pixels of `face` to black

```
face3 = face.copy()
face3[:50,:,:] = 0
face3[:-50:-1,:,:] = 0
plt.imshow(face3)
plt.show()
```

Time : 10 mins

Working with Image Arrays (cont)

Let us continue to work on the `face` image:

- We colour the left 50 pixels to red and right 50 pixels to green.

```
face3 = face.copy()
face3[:,:50,:] = 0
face3[:,:50,0] = 255
face3[:,:-50:-1,:] = 0
face3[:,:-50:-1,1] = 255
plt.imshow(face3)
plt.show()
```

- For fancier colours, we need to search them on the Internet.

Time : 5 mins

Working with Image Arrays (cont)

- We can even change the colours in the horizontal middle to blue and verticle middle to yellow (=red+green) by careful calculations.

```
face3 = face.copy() # face3.shape => (768, 1024, 3)
face3[768//2-25:768//2+25] = 0
face3[768//2-25:768//2+25,:,2] = 255
face3[:, 1024//2-25:1024//2+25] = 0
face3[:, 1024//2-25:1024//2+25,[0,1]] = 255
plt.imshow(face3); plt.show()
```

Time : 5 mins (10 mins break + Q&A)

Outline

Working with Image Arrays (cont)

- Changing the colours of the diagonals is possible but linear algebra is involved!!!
  - The four corners of the `face` image is (0,0) and (1023,0) (0,767) and (1023,767). The equations of the lines are probably

$$y_1 = \frac{767 - 0}{1023 - 0} x$$
$$y_2 = \frac{767 - 0}{0 - 1023} x + 767$$

Working with Image Arrays (cont)

- With appropriate rounding, we have

```
face3 = face.copy() # face3.shape => (768, 1024, 3)
x   = np.arange(face3.shape[1])
y1 = (767/1023*x).astype('int')
y2 = (767-767/1023*x).astype('int')
for i in range(-25,26):
    upb = face3.shape[0]-1
    shifted_y1 = y1+i
    shifted_y1[shifted_y1<0] = 0
    shifted_y1[shifted_y1>upb] = upb
    shifted_y2 = y2+i
    shifted_y2[shifted_y2<0] = 0
    shifted_y2[shifted_y2>upb] = upb
    face3[shifted_y1,x] = 0
    face3[shifted_y2,x] = 0
plt.imshow(face3)
plt.show()
```

Time : 10 mins

Working with Image Arrays (cont)

- Drawing any 'curve' onto the image is possible as long as we can know the appropriate mathematical formula. Let us consider the quadratic curve which we use a lot in SPM:

$$y = k(x - 1023/2)^2$$

We need to choose the value k so that the quadratic curve passes through the points (0,767) and (1023,767).

$$767 = k(0 - 1023/2)^2 = k(1023 - 1023/2)^2.$$

This implies

$$k = \frac{767 \times 4}{1023^2}.$$

Working with Image Arrays (cont)

- A possible Python implementation:

```
face3 = face.copy() # face3.shape => (768, 1024, 3)
x = np.arange(face3.shape[1])
k = 767*4/1023**2
y = (k*(x-1023/2)**2).astype('int')
for i in range(-25,26):
    upb = face3.shape[0]-1
    shifted_y = y+i
    shifted_y[shifted_y<0] = 0
    shifted_y[shifted_y>upb] = upb
    face3[shifted_y,x] = 0
    face3[shifted_y,x,0] = 255
plt.imshow(face3)
plt.show()
```

Time : 10 mins

Working with Image Arrays (cont)

So far, we have been using **array of integers** for indexing an array A. It is possible to use **array of Booleans** as a mask to change the array A.

The **mask** M of an array A is an array of Booleans which is like a new layer above the array A, which is used to select the 'portion' of A in the mask M which is true. For 1D example,

```
A = np.array([0.26, 0.52, 0.33, 0.1 , 0.15, 0.19, 0.8 , 0.38,
     0.18, 0.71, 0.27, 0.09])
```

We can 'pick' out the numbers larger than 0.5 using the mask

```
M = np.array([False,  True, False, False, False, False, True, False,
     False, True, False, False])
```

using

```
Alarger_than_half = A[M]
```

Working with Image Arrays (cont)

For a 2D-array A, the **mask** of A is a 2D array of Booleans over the indices:

$$\begin{array}{cccc} (0,0) & (0,1) & \ldots & (0,m-1) \\ (1,0) & (1,1) & \ldots & (1,m-1) \\ \vdots & \vdots & \ddots & \vdots \\ (n-1,0) & (n-1,1) & \ldots & (n-1,m-1) \end{array}$$

For face3, the shape is (768,1024) (ignoring the last index), so $n = 768$ and $m = 1024$.

For 2D array or 3D image array (e.g. `face3`), we need to form a 2D array of Booleans mask for **Boolean indexing / mask**.

Working with Image Arrays (cont)

Let see how we can use Boolean indexing to

- colour the top 50 pixels and bottom 50 pixels of face to black.
- colour the left 50 pixels to red and right 50 pixels to green.
- colour the horizontal middle to blue and verticle middle to yellow(=red+green)
- colour the diagonals to blue
- draw a quadratic curve $y = \frac{767\times4}{1023^2}(x - \frac{1023}{2})^2$ in red.

For all the above, we need to 'paint' True on an array of False and use it to make changes the 'face' image.

Time: 20 mins on Boolean indexing (answers given)

Working with Image Arrays (cont)

```
n = face.shape[0]; m = face.shape[1]
y = np.r_[:n].reshape((-1,1))
x = np.r_[:m].reshape((1,-1)) #Or: y, x = np.ogrid[:n,:m]

# Case 1
face3 = face.copy()
top_50 = y<50; bot_50 = y>n-50
M1 = np.repeat(top_50|bot_50, m, axis=1)
face3[M1] = 0
plt.imshow(face3); plt.show()

# Case 2
face3 = face.copy()
left_50  = np.repeat(x<50, n, axis=0)
right_50 = np.repeat(x>m-50, n, axis=0)
face3[left_50] = [255,0,0]
face3[right_50] = [0,255,0]
plt.imshow(face3); plt.show()
```

Working with Image Arrays (cont)

```
# Case 3
face3 = face.copy()
mid_h = np.repeat( (n/2-25<y) & (y<n/2+25), m, axis=1)
mid_v = np.repeat( (m/2-25<x) & (x<m/2+25), n, axis=0)
face3[mid_h] = [0,0,255]; face3[mid_v] = [255,255,0]
plt.imshow(face3); plt.show()

# Case 4: The y - n/m * x and y + n/m * (m-x)
#         will generate 2D arrays
face3 = face.copy()
diag = ( np.abs(y - n/m * x) < 25 ) | \
       ( np.abs(y + n/m * x - n) < 25 )
face3[diag] = [0,0,255]; plt.imshow(face3); plt.show()

# Case 5:
face3 = face.copy()
M = np.abs(y - 767*4/1023**2*(x-1023/2)**2) < 25
face3[M] = [255,0,0]; plt.imshow(face3); plt.show()
```

Working with Image Arrays (cont)

1. Write down the Python commands to generate the following image on the left from `face`:

Extra lab: Try to write your commands so that it is easy for you to work on your favourite image.

Time: 10 minutes. Ending at 5:50pm.