School of Computing
National University of Singapore
CS4243 Computer Vision and Pattern Recognition
Semester 1, AY 2015/16

Lab 1: Numerical Computations with Python

(due date: end of each lab session)

Objectives:

- Learn to use Python and its numerical package NumPy for numerical computations.

Preparation:

- Create a folder in the PC with your name, e.g., `d:/myname`. This folder will be used as your working directory.
- Download the files `numcompAY1516.pdf` & `data.txt` into your working directory

Part 1. Starting Python

This part illustrates how to start using Python.

1. Start Python interpreter.
   Windows: Select Start → All Programs → Python → IDLE (Python GUI).

2. Explore and change working directory.
   a. Import `os` module for interface to operating system.
      ```
      >>> import os
      ```
   b. Get current working directory. This will display the default current working directory.
      ```
      >>> os.getcwd()
      ```
   c. Change current working directory to your working directory, e.g., `d:/myname`.
      This allows you to access the images in your working directory without having to specify the full path names.
      ```
      >>> os.chdir("d:/myname")
      ```
   d. Examine the current working directory. It should display the one that you have set.
      ```
      >>> os.getcwd()
      ```
   e. List the files in the current working directory.
      ```
      >>> os.listdir(".")
      ```

3. Set the working directory, e.g., `d:/myname`, and import the numpy module.

```
>>> import os
>>> os.chdir("d:/myname")
>>> import numpy as np
```

Part 2. Explore NumPy Arrays and Matrices

NumPy supports multi-dimensional array called ndarray. In NumPy, a matrix is a special 2D array that allows matrix multiplication. In most cases, a 2D array can be regarded as the same as a matrix, except for matrix multiplication.

Useful Note: To do the exercises in Part 2, you don't need to type everything. You can just copy and paste the statements, one at a time, into the Python interpreter.

1. 1D arrays
   - Create 1D arrays.

     ```
     >>> a = np.array([1, 2, 3])
     >>> a
     >>> print a
     >>> b = np.array([0.1, 0.2, 0.3])
     >>> print b
     ```

   - Array elements are indexed from 0 to *n*-1, where *n* is the number of elements.

     ```
     >>> for i in range(len(b)):
             print i, b[i]
     ```

     Note: len(b) should give the length of array. It works on macbook, but causes error in windows. For windows, replace len(b) with b.shape[0] and that should work.

   - Operations on arrays are performed in an element-by-element manner.

     ```
     >>> 0.5 * a
     >>> a + b
     >>> a – b
     >>> a * b
     ```

   - Explicit loop over array elements runs slower, particularly for large arrays.

     ```
     >>> c = np.zeros(3)
     >>> for i in range(len(c)):
             c[i] = a[i] + b[i]
     >>> print c
     ```

So, it is advisable to use the built-in array operations that not only runs faster but are also more compact and easier to read.

2. 2D arrays
   - Create 2D arrays.

```
>>> a = np.array([[1, 2, 3], [4, 5, 6]])
>>> a
>>> print a
>>> print a.shape
>>> b = np.array([[0.1, 0.2, 0.3], [0.4, 0.5, 0.6]])
>>> b
```

   - Get an element.

```
>>> a[1,2]
```

   - Get a row.

```
>>> a[1,:]
```

   - Get a column.

```
>>> a[:,2]
```

   - Operations on arrays are performed in an element-by-element manner.

```
>>> 0.5 * a
>>> a + b
>>> a – b
>>> a * b
```

3. Matrices
   - Create matrices.

```
>>> a = np.matrix([[1, 2, 3], [4, 5, 6]])
>>> a
>>> print a
>>> b = np.matrix([[0.1, 0.2, 0.3], [0.4, 0.5, 0.6]])
>>> b
```

   - These matrix operations produce the same results as operations on 2D arrays.

```
>>> 0.5 * a
>>> a + b
>>> a – b
```

- This operation is invalid because the two matrices are not of compatible sizes.

```
>>> a * b
```

- Matrix multiplication is valid only if the matrices have compatible sizes.

```
>>> c = np.matrix([[0.1, 0.2], [0.3, 0.4], [0.5, 0.6]])
>>> print a.shape, c.shape
>>> a * c
```

4. Special arrays
   - Create special 2D arrays.

```
>>> np.empty([2,3])       # array with arbitrary values
>>> np.eye(2,3)    # with ones along diagonal and zeros elsewhere
>>> np.identity(3)        # identity array
>>> np.ones([2,3])        # array filled with ones
>>> np.zeros([2,3])       # array filled with zeros
>>> np.random.rand(2,3)   # array with random values
```

   - Convert 2D arrays into matrices.

```
>>> a = np.identity(3)
>>> a
>>> b = np.matrix(a)
>>> b
>>> np.matrix(np.identity(3))   # short-hand method
```

Part 3. Systems of Linear Equations

1. System of well-determined Linear Equations

   A system of linear equations such as

$$x + y + z = 2$$
$$2x + z = 1$$
$$x + 2y + z = 3$$

   can be written in the matrix form as

$$\mathbf{A}\,\mathbf{x} = \mathbf{b}$$

where

$$\mathbf{A} = \begin{bmatrix} 1 & 1 & 1 \\ 2 & 0 & 1 \\ 1 & 2 & 1 \end{bmatrix}, \ \mathbf{x} = \begin{bmatrix} x \\ y \\ z \end{bmatrix}, \ \mathbf{b} = \begin{bmatrix} 2 \\ 1 \\ 3 \end{bmatrix}.$$

There are three independent equations with three unknowns. So, the unknowns can be solved by computing

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

You can write a Python program to solve for **x**.

```
import numpy as np
import numpy.linalg as la
A = np.matrix([[1, 1, 1], [2, 0, 1], [1, 2, 1]])
print "A =\n", A
b = np.matrix([[2], [1], [3]])
print "b =\n", b
x = la.inv(A) * b
print "x =\n", x
print "A * x=\n", A * x
```

Let's keep this program in the file `lineq-1.py`. This is also called a Python script file.

To execute the script file, do

```
>>> execfile("lineq-1.py")
```

The `execfile` function executes the statements in the script file as though they were entered into the interpreter one after another. If you edit the script file, you can re-run the script file in the same way. In Windows, you can also run the script file by double-clicking its file icon.

What is the value of **x**? Is **A x** = **b**?

Another way to solve for x is to use the built-in function `solve`:

```
x = la.solve(A, b)
```

Exercises

- Write and run the programme `lineq-1.py`.
- Append to `lineq-1.py` the method of solving equations using `solve`. Re-run the program and verify that the two methods produce the same results.

To edit your program, you can use the editor that comes with IDLE, the Python GUI. Alternatively, you can use NotePad++, Vim or NotePad in Windows, and Vim, GNU Emacs or any other editors in Linux.

System of Over-determined Linear Equations

In Computer Vision, we often encounter over-constrained equations in which there are more equations than the number of unknowns. Consider, for example, the following 2D affine transformation:

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x_i \\ y_i \\ 1 \end{bmatrix} = \begin{bmatrix} x'_i \\ y'_i \\ 1 \end{bmatrix} \quad \text{for } i = 1,\ldots, n.$$

This set of equations can be written as

$$\mathbf{A}\ \mathbf{x}_i = \mathbf{x}'_i\ . \tag{1}$$

To solve Eq. 1 for **A**, we rearrange the equations as follows:

$$\mathbf{M}\ \mathbf{a} = \mathbf{b} \tag{2}$$

where

$$\mathbf{M} = \begin{bmatrix} x_1 & y_1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & x_1 & y_1 & 1 \\ & & \vdots & & & \\ x_i & y_i & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & x_i & y_i & 1 \\ & & \vdots & & & \\ x_n & y_n & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & x_n & y_n & 1 \end{bmatrix}, \quad \mathbf{a} = \begin{bmatrix} a_{11} \\ a_{12} \\ a_{13} \\ a_{21} \\ a_{22} \\ a_{23} \end{bmatrix}, \quad \mathbf{b} = \begin{bmatrix} x'_1 \\ y'_1 \\ \vdots \\ x'_i \\ y'_i \\ \vdots \\ x'_n \\ y'_n \end{bmatrix}.$$

As **M** is not a square matrix, it does not have an inverse. Eq. 2 can be solved by pre-multiplying the transpose of **M** to the equation:

$$\mathbf{M}^{\mathrm{T}}\mathbf{M}\ \mathbf{a} = \mathbf{M}^{\mathrm{T}}\mathbf{b}.$$

The matrix $\mathbf{M}^{\mathrm{T}}\mathbf{M}$ is a square matrix with an inverse (if its determinant is not zero i.e. the matrix is of full-rank). So,

$$(\mathbf{M}^{\mathrm{T}}\mathbf{M})^{-1}(\mathbf{M}^{\mathrm{T}}\mathbf{M})\ \mathbf{a} = \mathbf{a} = (\mathbf{M}^{\mathrm{T}}\mathbf{M})^{-1}\mathbf{M}^{\mathrm{T}}\mathbf{b}.$$

In general, $\mathbf{M}^{\mathrm{T}}\mathbf{M}$ can be a very large matrix. So, other methods are used to solve Eq. 2. NumPy provides a function that solves for the least-squared error solution:

```
a, e, r, s = la.lstsq(M, b)
```

The results `a`, `e`, `r`, `s` are, respectively, the least-square solution `a`, the residue or error `e`, the rank `r` of `a`, and the singular values `s` of `a`.

Exercises (to be submitted)

Write a program to solve for the affine transformation parameters.

- The data are stored in `data.txt` in comma separated format called Comma Separated Values (CSV):

  x_1' , y_1' , x_2 , y_2
  ⋮
  x_n' , y_n' , x_n , y_n

- Read the data from the data file `data.txt` as follows:

```
file = open("data.txt")
data = np.genfromtxt(file, delimiter=",")
file.close()
```

- Print `data` to verify that the data are correctly read.
- Create the matrices (not 2D arrays) `M` and `b` of appropriate sizes. The number of data points in `data` can be obtained using the function `len(data[:,0])`.
- Copy the values in `data` into `M` and `b`.
- Print out `M` and `b` to verify that their values are correct.
- Solve for the parameter matrix `a` using `la.lstsq`.
- Print `a` to examine its values.
- The value of `M*a` should be close to that of `b`.
- Compute and print the sum-squared error between `M*a` and `b`. Instead of writing an explicit loop over the matrix elements, use `la.norm(M*a-b)` to compute the norm of the difference between `M*a` and `b`. The sum-squared error is just the square of the norm of the difference. The sum-squared error should be small.
- Print the residue computed by `la.lstsq`. This value is the sum-squared error.

Submit the softcopy of your Python program in IVLE, the hardcopy printout of your Python program, and the hard copy of screen outputs.

For the softcopy submission, please put your python program in a folder and submit the folder. Use the following convention to name your folder:

MatriculationNumber_yourName_Lab1. For example, if your matriculation number is A1234567B, and your name is Chow Yuen Fatt, for this lab, your file name should be A1234567B_ChowYuenFatt_Lab1.

For the hardcopy submission of screen outputs, remember to write your matriculation number, name and lab session (eg. Tue630pm, or Fri10am) on the hardcopy output.