Systems and Signals 414: Python and Jupyter setup

This is a primer on Python, Numpy, Matplotlib and Jupyter (previously named The IPython Notebook). In this course we will rely heavily on the numeric, scientific, and plotting libraries, and use Jupyter as our main coding environment.

WinPython

This is a batteries-included portable Python 3 build for Windows with lots of additional libraries and Python orientated software. The Windows binaries for WinPython3 is on the course website (it's official home is here http://winpython.github.io). Winpython3 comes pre-installed on the SS207A computers under `C:\progs`.

Jupyter:

Jupyter (previously named The IPython Notebook) is a webui that allows you to write either markdown snippets or execute Python code in snippits (each snippit is called a cell). This tutorial is written in Jupyter and we ask you to also do your practicals in Jupyter. Since it is a webui, it needs to be served on a port and accessed through a browser. This process is luckily all automated through WinPython's "`IPython Notebook.exe`". Try it by downloading one of the practicals and drag-and-dropping it into "`IPython Notebook.exe`" (or by using open-with).

Markdown: This text here is readable because this cell is set to markdown. Here is some basic markdown syntax. When you want to write mathematics, do not use plain text; use latex-like equations. For example parenthesising `c_1\sin(x) + c^2_2\,{\rm sinc}(\frac{\pi}{c})` in double dollar signs will produce the separate line

$$c_1 \sin(x) + c_2^2 \,\mathrm{sinc}(\frac{\pi}{c}),$$

and parenthesising `ax+b` in single dollar signs will produce $ax + b$ inline.

Code: Take note that variables within each cell are stored for global use by other cells after the cell's execution. The side effect is that you can initialise a variable in one cell, run that cell, delete that cell, and the variable would still be available. Another side effect is that if a cell is updated, you have to run the consecutive cells in order for them to pick up the updated results.

Warning: make sure your code cells make sense when run in sequence from a clean slate; otherwise we will not be able to reconstruct your output. Test this before handing in by restarting the Python kernel (`Kernel` → `Restart`) and running all the cells.

The Numpy library

This provides MATLAB-like arrays and a lot of core mathematical functions. As a rule of thumb, choose the Numpy functions over the core Python functions. Use Numpy arrays `np.array([1,2,3])` for vector-based mathematical operations and stay clear of Python lists `[1,2,3]` for this purpose. Also, stay absolutely clear of the MATLAB compatible `np.mat` class, it has some unexpected behaviour.

```
In [5]: #Import under a quick-to-type namespace
        import numpy as np

        A = np.array([1,2,3,4.3])
```

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

        print('A:', A)
        print('Elementwise squared:', A**2)
        print('B:\n', B)
        print('B times B:\n', B.dot(B))
```

```
A: [ 1.   2.   3.   4.3]
Elementwise squared: [  1.     4.     9.    18.49]
B:
 [[1 2]
 [3 4]]
B times B:
 [[ 7 10]
 [15 22]]
```

The Matplotlib library

The library is under the namespace matplotlib and can be imported and used as such `import matplotlib`, but we rather import it from the namespace `import pylab`. Do not be confused by this, pylab imports matplotlib and a bunch of other things directly under the namespace `pylab` for easy and convenient access. Matplotlib is object-orientated, but can easily be used procedurally (in the same manner as MATLAB plotting). With regards to using it in Jupyter, some awkward-to-remember additional tweaking is recommended:

```
In [6]: #Jupyter specific - plots as inline png's
        %matplotlib inline

        #import the plotting library
        import pylab as pl

        # Jupyter specific - change the size of the output plot
        pl.rcParams['figure.figsize'] = (9, 2)
```

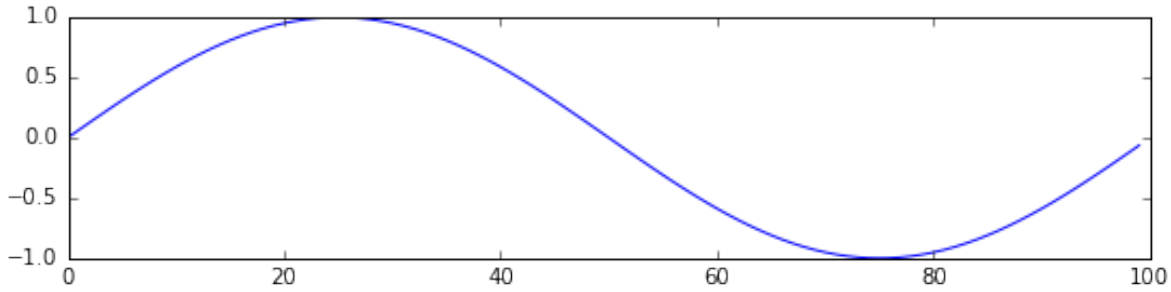
We recommend the above as a copy-paste setup for plotting. The `figure.figsize` parameter can always be reset before a `pl.figure()` to set a new default size.

`%matplotlib inline` is a magic function to render plots as `png`'s and display them underneath the cell. To rather produce interactive plots (with zooming and other features), replace that line with `%matplotlib notebook`. Your hand-ins must be in `inline` mode! To switch to and fro the `inline` and `notebook` behaviour, you have to restart the kernel and run all the cells again (for each switch).

Warning: be aware of the pipeline difference between `inline` and `notebook`. `%matplotlib inline` automatically generates a new figure object (i.e. a new plotting canvas) for each cell, where `%matplotlib notebook` will continue plotting on the current figure (this might even be a figure from a different cell). Therefore, in `notebook` mode, within each cell, be sure to call `pl.figure` before any plot calls in order to generate a new figure for each cell. In fact, **make a habit of calling `pl.figure()` for each cell containing a plot** regardless of `inline` or `notebook`; this will ensure similar behaviour when using either.

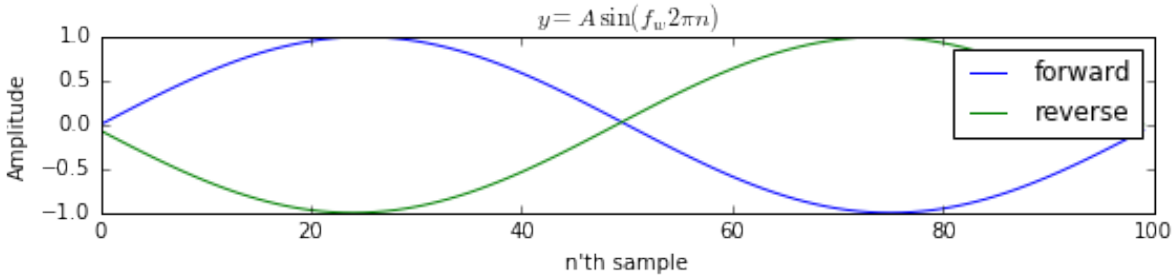
Example of a basic plot

```
In [7]: x = np.arange(100)*2*np.pi/100
        y = np.sin(x)
        pl.figure() #create a new figure
        pl.plot(y);
```



Example of an annotated plot

```
In [8]: pl.rcParams['figure.figsize'] = (9, 1.5)
        pl.figure() #generate a new fig
        pl.ylabel("Amplitude")
        pl.xlabel("n'th sample")
        #These labels are latex-compatible!
        #note r"..." produces a raw-string: no slash escaping
        pl.title(r"$y = A\,{\rm sin}(f_w 2 \pi n)$")
        pl.autoscale()
        pl.plot(y, label='forward');
        pl.plot(y[::-1], label='reverse');
        pl.legend();
```



Some useful Python equivalent Matlab code

Array creation

One of the core differences between Python and Matlab is that Python uses the more standard zero base indexing system. This also comes with a philosophy applied to logic, loops and indexing that any range or counting like logic performed on a as the start and b as the end, the logic will follow as "from a up to (but excluding) b".

```
Python:                                 Matlab:
a = np.arange(0, 1, 0.1)                a = 0:0.1:0.9
b = np.linspace(0, 1, 10)               b = linspace(0, 1.0, 10)
b = np.linspace(0, 1, 10, False)        b = linspace(0, 0.9, 10)
c = np.zeros(20)                        c = zeros(20,1)
d = np.ones(15)                         d = ones(15,1)
f = np.array([1, 2, 3, 4])              f = [1 2 3 4]'
```

Random sampling

```
Python:                                 Matlab:
a = np.random.normal(0.0, 1.0, 100)     a = randn(100, 1)
b = np.random.uniform(0.0, 1.0, 20)     b = rand(20, 1)
c = np.random.randint(1, 11, 50)        c = randi(10, 50, 1)
```

General mathematical functions

Note that for `b`, `c` and `d` the output is an array with the same dimensionality as the input.

```
Python:                                 Matlab:
b = np.exp(a)                           b = exp(a)
c = np.sin(a)                           c = sin(a)
d = np.sum(a)                           d = sum(a)
f = len(a)                              f = length(a)
```

Element-wise operations

The default behaviour in Numpy is usually element-wise, as opposed to matrix and vector operations in Matlab.

```
Python:                                 Matlab:
c = a + b                               c = a + b
d = a * b                               d = a .* b
f = a / b                               f = a ./ b
g = a**2 * np.exp(b)                    g = a.^2 .* exp(b)
```

Matrix and vector operations

Note that the lowercase variables are 1d numpy arrays (used as mathematical vectors), where the uppercase variables are 2d numpy arrays (used as mathematical matrices).

```
Python:
c = np.dot(a, b)
D = np.outer(a, b)
f = np.dot(A, b)
f = A.dot(b)
```

```
Matlab:
c = a' * b
D = a * b'
f = A * b
f = A * b
```

Logical operations

Binary values in Python are indicated with `True` or `False`, where Matlab uses `1` or `0`. Some logical operations can also be applied with equivalent mathematical functions (such as using multiplication `*` for logical and `&`). Note that `a` is a numpy array.

```
Python:
b = a > 1
c = (a >= 1) & (a < 5)
c = (a >= 1) * (a < 5)
d = (a >= 1) | (a < 5)
d = (a >= 1) + (a < 5)
f = ~(a >= 1)
```

```
Matlab:
b = a > 1
c = a >= 1 & a < 5
c = a >= 1 & a < 5
d = a >= 1 | a < 5
d = a >= 1 | a < 5
f = ~(a >= 1)
```

These logical operations can be useful for array indexing or counting.

```
g = a[a >= 1]
h = np.sum(a >= 1)
a[a >= 1] = 2 * a[a >= 1]
a[a >= 1] = 0
```

```
g = a(a >= 1)
h = sum(a >= 1)
a(a >= 1) = 2 * a(a >= 1)
a(a >= 1) = 0
```

In conclusion

You should now be able to setup a work environment for the practicals using Jupyter on a Windows system. A linux setup is also possible and quite easy, just be sure to install Python 3, IPython, Numpy, Scipy and Matplotlib and be prepared to google a bit. We will give some further instructions and templates for the practicals.