Programming in Matlab/Octave or Python

I will give code snippets in Matlab and Python during the course. All Matlab examples will also work in the largely-compatible free package Octave. You are expected to become familiar with one of Matlab/Octave or Python, and use it to check your understanding through the course, and for the assessed assignment. Sometimes I will only give demonstrations with Matlab/Octave. If you want to use Python exclusively, being able to port code snippets in other languages is a useful skill! I'm happy to answer porting questions if you've tried and are genuinely stuck.

Why Matlab/Octave: If you don't have much programming experience, you are likely to have fewer problems getting started with Matlab or Octave. The language is simpler than the combination of Python and its scientific libraries, and the base install of Matlab or Octave will do everything you need for this course. I often prefer giving Matlab/Octave examples, as they tend to be quicker to set up, and make fewer assumptions about your installation. Although Python+NumPy is neater for some types of calculation.

Why Python: Python is good for writing larger programs, and accessing large machine learning frameworks like TensorFlow, Theano, or Keras. If you know Python well, or have lots of programming experience, you will probably want to work with Python. Personally I'm using it more and more in my research. However, you'll probably have to do more work to get set up, and you'll have to learn to routinely import some modules as outlined below.

Other languages? Languages like R or Lua (with Torch), are also sensible choices for machine learning. I'm not personally used to the quirks in the R language, however it has a large collection of well-documented statistical packages in CRAN, and is a good choice if you primarily want to use existing statistical toolboxes. If you want to write compiled code, you might look at using the C++ library Eigen. If you learn the principles of array-based computation for machine learning with Matlab or Python, you should be able to rapidly generalize to whatever tool you need to use in the future.

Getting started with Matlab/Octave

Matlab and Octave are both installed for your use on the Informatics DICE computer system. You can obtain a free personal MATLAB license from the university for your own machine. However, if you use the free software Octave, it will do everything you need for this course, you won't need to fiddle about with licenses, and you could use it after leaving the University.

You should work through a tutorial that covers creating vectors and matrices, evaluating expressions containing matrices and vectors, basic plotting (line graphs, histograms), and basic programming constructs (for loops, and functions). Suitable tutorials include:

- Cambridge Engineering Octave tutorial
- The Mathwork's own Matlab tutorial

The Mathworks also have more extensive online training, which would normally cost money, but you can access with an Edinburgh University email address. Warning: Mathworks like to encourage you to use their toolboxes (so they will sell more toolbox licenses in future). In this class you will need to understand the fundamentals and be able to implement algorithms from scratch (from linear algebra primitives), rather than learning how to use some toolbox of pre-packaged machine-learning routines.

Optional extra, for keen students:

- Outline notes on writing fast Matlab/Octave code

Matlab/Octave datatypes

Something that makes Matlab/Octave simple for beginners is that when you start out, every variable will be a matrix, or a 2-dimensional array of numbers. If you say "`xx = 3.14`", then xx is a 1×1 matrix, which you can confirm with "`size(xx)`". Similarly, a column vector "`yy = randn(3,1)`", is a 3×1 matrix. In contrast, Python distinguishes between vectors and 2-dimensional arrays. In fact, Python makes many more subtle distinctions (sketched later), which you need to have some awareness of if you use Python.

The second Matlab/Octave type you are likely to need are "cell arrays". These are arrays that can contain matrices (and other types) of different sizes. These might be useful for grouping together all of the parameters of a model. You can create cell arrays with curly braces or the `cell()` function, for example: "`Z = {randn(3,3), randn(2,1)}`". You would access the first matrix using a curly-braced index: `Z{1}`. You'll rarely want `Z(1)`, with round brackets, which returns a cell array of length one, containing the first matrix.

repmat, bsxfun, and broadcasting

It's common to want to subtract a `1xM` row-vector `rv` from every row of an `NxM` matrix `A`. Really old Matlab tutorials (before `bsxfun` was available) told you to do this:

```
A - repmat(rv, N, 1) % repmat gives NxM matrix with rv in each row
```

Then `bsxfun` was introduced, which means you don't need to create the intermediate large matrix:

```
bsxfun(@minus, A, rv) % same result as above
```

So if you see any old tutorials using `repmat`, you may want to check whether you can use `bsxfun` instead.

If you have Matlab 2016b or later, or a recent version of Octave, they support "broadcasting". You can subtract the row vector off each row of the matrix with the simplest possible syntax:

```
A - rv
```

Getting started with Python

Python and its associated scientific libraries are installed on the Informatics DICE system. However, these packages change quickly, and DICE is likely to have older versions than you would install on your own machine.

If installing on your own machine, I recommend trying the Anaconda distribution, unless the package manager you normally use to install software has well-maintained Python packages. Some software distributions come with fairly old Python packages, whereas Anaconda usually "just works". Whatever route you take, you'll want at least Python, NumPy, SciPy, and matplotlib. Either Python 2 or Python 3 is ok (more below).

If you don't already know the basics of Python, you should probably just start out with Octave instead. Otherwise you'd first have to find a Python tutorial at your level, and work through it. The official Python tutorial is a good start. (You don't need the more advanced topics, like classes, or to work through all the standard library examples.) Then you would need to learn the NumPy and matplotlib libraries. Again, there are many tutorials online. You might start with the official quickstart guide. For more, you could work through some of `http://www.scipy-lectures.org/`, which aims to be "One document to learn numerics, science, and data with Python".

I use Python interactively from the ipython command-line program. From there you can type `%paste` to run code in the clipboard, or use the `%run` command to run code stored in a file. If you get an error, you can use `%debug` to enter a debugger. If you start ipython with `ipython --pylab` then plotting works smoothly: there's no need for show() commands, and plot windows don't cause the interpreter to hang. The pylab environment also imports commonly-used functions like plot() into the top level, so you can use it more like Matlab.

Those that like a graphical environment could try Spyder.

Ipython or Jupyter notebooks are becoming popular, and are used in the MLP course. I've chosen not to create notebooks for this course because these tools are still changing quickly. Getting everyone running the right version could take a lot of time, and I don't want to rely on a central server, which you wouldn't be able to access after the course is over. However, if you like the notebook interface, feel free to use it yourself.

Commonly-used Python modules

If you use Python you will use NumPy extensively. The standard way to use this module is

```
import numpy as np
```

Then some example code would be:

```
A = np.random.randn(3, 3)
matrix_product = np.dot(A, A) # simply "A @ A" with python >=3.5
```

I might not always specify the `import` line in my Python examples, but you'll need it if my code refers to `np.something`. Similarly if I refer to `plt`, a Matlab-like plotting interface, you'll need to import it as follows:

```
import matplotlib.pyplot as plt
```

Some people reduce the amount of typing they need to do with:

```
from numpy import *
from matplotlib.pyplot import *
```

which means code can directly call functions like dot() and plot() without a "np." or "plt." prefix. Ready access to the functions is convenient for interactive use, but importing a large set of functions is usually considered poor practice in "real code". For example Python's sum() and max() and NumPy's np.sum() and np.max() could become confused with each other, which can lead to subtle bugs.

Python/NumPy Arrays, matrices, vectors, lists, tuples, . . .

One reason that numerical computation with Python is more complicated for beginners than Matlab is the larger number of types you have to deal with immediately.

Python's usual `tuple` and `list` types don't provide convenient array-based arithmetic operations. For example

```
xx = [1, 2, 3] # python list
print(xx*3)          # prints [1, 2, 3, 1, 2, 3, 1, 2, 3]
print((1,2) + (3,4))  # prints (1, 2, 3, 4)
```

You will use the list or tuple types to initialize NumPy arrays, and also as containers of NumPy arrays of different shapes (like Matlab's cell arrays).

NumPy has a "matrix" type (created with `np.matrix`), which I strongly recommend you avoid completely (as does the wider NumPy community). Standard practice is to use NumPy *arrays* for all vectors, matrices, and larger arrays of numbers. Attempting to mix NumPy matrix and array types in your code is likely to lead to confusion and bugs.

One way to ensure you're dealing with NumPy arrays is to convert to them at the top of functions you write:

```
def my_function(A):
    A = np.array(A) # does nothing if A was already a numpy array
    N, D = A.shape # now works if A was originally a list of lists
```

Unlike Matlab, NumPy distinguishes between scalars, vectors, and matrices. If you're going to use NumPy, you should know (or work out) what the following code outputs, and why:

```
A = np.random.randn(3, 2)
print(A.shape)
print(np.sum(A,1).shape)
print(np.sum(A).shape)
```

If some NumPy code expects an array of shape `(N,)`, a vector of length `N`, it might not work if you give it an array of shape `(N,1)` or `(1,N)` (and vice-versa). You can convert between vectors and 2D arrays using `np.reshape`, `np.ravel()`, and indexing tricks.

Broadcasting

A common NumPy task is to subtract a vector `rv` from every row of a Matrix `A` stored in an array:

```
# For shape N,M array A, and shape M, array rv:
A - rv #  or more explicitly: A - rv[None,:]
```

To subtract a vector `cv` from every column:

```
# for shape N, array cv
A - cv[:,None]
```

Here "`None`" creates a new axis: `cv[:,None]` is a 2-dimensional array with shape `N,1`. The single column is automatically "broadcast" by NumPy across the M columns of `A`. If you didn't expand `cv` into a 2-dimensional array, the subtraction would fail. You can use `newaxis` from the numpy module instead of `None`, which is more explicit. However, I don't always want to have to import `newaxis`, and `np.newaxis` is too long to spatter all over indexing code. I don't think the NumPy people are going to break the use of `None`, because lots of code uses it and it's documented.

Python 2 vs Python 3

Python 2 support is due to end in 2020, so most people will need to migrate to Python 3 sooner or later. I often use Python 3 for my personal code, and you may wish to do the same. Unfortunately, when working with others, things often have to move more slowly. As OpenAI state, many machine learning researchers are still using Python 2, and that's what is currently best supported on the Informatics DICE system. Although *finally* this year, I'm beginning to see some meaningful shift towards Python 3.

I've tried to make all my code examples work in both Python 2 and Python 3.

The main change in Python 3 is Unicode string handling, which isn't relevant for the sort of code we'll write in this course. The minor issue you'll have to deal with in practice is avoiding Python 2 print statements:

```
print "Hello World!"  # Python 2 code that will crash in Python 3
```

Add parenthesis around the string as follows:

```
print("Hello World!")  # Works in both Python 2 and Python 3
```

Replace any more complicated Python 2 print statements with Python 3 style print functions. Then you can add a magic import line at the top of your code to make them work in Python 2 as well. For example:

```
from __future__ import print_function

print('thing1', 'thing2', sep=', ')
```

Python 3.5 comes with a matrix multiply operator `@` which performs `np.matmul`. You can often write `A @ B` instead of `np.dot(A, B)`. However, be careful: `np.matmul` has different broadcasting rules and doesn't work with scalars[1]. There is also no easy way to get the `@` operator in earlier versions of Python, so I'll try to remember to use `np.dot` in my examples. But if you're using Python 3, you could go ahead and try out the `@` operator in your own code.

---

1. Thanks to James Ritchie for this warning about blanket replacing `np.dot` with `@`.