Programming in Matlab/Octave or Python

There will be 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 demonstrations are only in Matlab/Octave. If you want to use Python exclusively, being able to port code snippets in other languages is a useful skill! We're happy to answer porting questions on the forum if you've tried and are genuinely stuck.

Why Matlab/Octave: you might choose Matlab or Octave for this course if you've used it before, or if you don't have much programming experience and won't need Python for another course. 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. Matlab/Octave examples are often 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: PyTorch, TensorFlow and many others. You should probably use Python if you already know the base Python language well, or have lots of programming experience, or if you'll have to learn it for other courses anyway (e.g., MLP). However, you will probably have to do more work to get set up, and you'll have to learn to routinely import some modules as outlined below.

What about other languages? Fashions can change quickly. Until 2016, Lua was used with Torch as the main machine learning framework at Facebook AI and Google DeepMind, and so might have seemed like a safe and good option. However, there is now next to no community around this framework. R could also be a sensible choice 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 (as used internally by TensorFlow).

The important thing is to learn the principles of array-based computation for machine learning. If you start with Matlab or Python, you should be able to rapidly generalize to whatever tool you need to use in the future.

1 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 to use it on 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

You may also be able to access more extensive training from Mathworks through the university link above, if we currently license it. 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

1.1 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.

1.2 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 told you to do this:

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

Then `bsxfun` was introduced, which avoids creating the large intermediate matrix:

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

If you have Matlab 2016b or later, or a recent version of Octave, they support "broadcasting", so `bsxfun` isn't needed. You can instead subtract the row vector off each row of the matrix with the simplest possible syntax:

```
A - rv
```

However, you might be surprised that subtracting two "vectors" can now give you a matrix:

```
v1 = randn(3, 1);
v2 = randn(1, 3);
v1 - v2           % element (i,j) is v1(i) - v2(j)
```

It's worth regularly checking the sizes of the arrays that you compute while debugging your code, as Matlab will often execute without error expressions that you didn't intend.

If writing a function, users of the function might provide vectors as row- or column-vectors. To deal with either, the above examples could be replaced with:

```
A - rv(:)'      % subtract a vector off every row
v1(:) - v2(:)   % for a vector of differences
v1(:)' - v2(:)  % to create a matrix as before
```

where "`(:)`" puts all the numbers of an array into a column vector.

2 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, we 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. If you can, install Python 3, but this course's materials are still backwards compatible with Python 2 (more below).

If you don't already know the basics of Python, you should first 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".

You can 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 --matplotlib` then plotting works smoothly: there's no need for `plt.show()` commands, and plot windows don't cause the interpreter to hang. Alternatively type `%matplotlib` after starting ipython.

Those that like a graphical environment could try Spyder. There are also popular heavy-weight commercial environments such as PyCharm.

IPython or Jupyter notebooks are becoming popular, and are used in some other courses. If you like the notebook interface, feel free to use it yourself. I think they're great for producing a demonstration of how to use a library, or for working notes where you can save results inline. However, I don't personally find them to be a good way of holding the main code for a project, or for collaboration. Putting results in the file doesn't work well with version control, and if you send someone a notebook, you're forcing them to launch a server and open a web-browser, rather than using the text editor of their choice. Make sure you are also able to work with code stored in `.py` files.

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

which means code can directly call functions like dot() and plot() without a "np." or "plt." prefix. Some simple Matlab examples work unaltered (although care is required). 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.

2.2 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, even 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.

2.3 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.

2.4 "Assignment" and pass-by-reference

This section is about a common misunderstanding that can lead to incorrect Python code.

In Python "=" is used for "assignment", but when there's just a variable name on the left, a more precise description is that it's for "attaching the name on the left-hand side to the object on the right-hand side". A simple example is:

```
A = np.ones((2, 2))
B = A
B[0, 0] = 25
print(A)
```

The second line "`B = A`" attaches the name `B` to the same object that the name `A` is already attached to — a 2×2 array of ones.

"`B[0, 0] = 25`" modifies the first element of the underlying object, so both `A` and `B` are changed.

If you don't want to accidentally change arrays, write "`B = A.copy()`" not "`B = A`". Also for *slices*: "`first_row = A[0].copy()`"

For objects other than NumPy arrays, you might need:

```
import copy
B = copy.deepcopy(A)
```

Similarly, arguments to functions are references to objects that might have other names. You shouldn't alter the original objects, unless the caller definitely knows that the arguments could be modified. Here's one pattern to make a function safer to use:

```
def my_function(A, in_place=False):
    if not in_place:
        A = A.copy()
    A += 1 # ... modify A
    return A

A = np.ones((2, 2))
B = my_function(A)                # A and B are different
B = my_function(A, in_place=True) # A and B are the same (saves memory)
```

Or you could just always take the copy, losing the ability to save some memory and time, but making the code simpler.

If you don't take copies, you'll have to be pretty careful to track when names share objects. For example, it takes a moment to be sure what the following does:

```
A = np.ones((2, 2))
B = A
B = B + 6
print(A)
B += 6
print(A)
```

The right-hand side of "`B = B + 6`" creates a new object[1]. So this line *doesn't* affect the object referred to by `A`. Moreover, this line attaches `B` to a different object than `A`. Therefore, the line "`B += 6`" doesn't affect `A` either — although it would have done without the "`B = B + 6`" line!

So "`B = A.copy()`" isn't necessary in this example. But it would have been a good idea for clarity, and could avoid bugs later when the code is altered.

If none of that is confusing: congratulations! You're probably an experienced programmer.

If it is confusing, you're not alone. Two options: 1) Matlab/Octave code is simpler to reason about. 2) If you need or want to learn Python, invest time to work through examples like those above at a Python prompt. Also try out each part of code you write on small example arrays to check it does what you think it does. (Which you should do in general.)

2.5 Python 2 vs Python 3

The code examples in the notes should all work in both Python 2.7 and Python 3, so you could use either. However, Python 2 support is falling away, so it would be sensible to write new code in a Python 3 compatible way.

The migration from Python 2 to 3 has been slow and painful. Pretty recently, OpenAI stated many researchers were still using Python 2, and you're still likely to find Python 2 only code.

1. *In theory*, if `B` referred to an object from a different library than NumPy, "`+`" *could* modify `A` in place and return the original object. In that case, `A` would be modified, and would still refer to the same object as `B`. Ouch! Fortunately, the classes in most libraries aren't written to make "`+`" have surprising side-effects.

However, Python 3 is now — finally — widely enough used that you should definitely use it for new code.

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 could 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[2]. There is also no easy way to get the `@` operator in earlier versions of Python, so the examples in the notes use `np.dot`. But if you're using Python $\geq$ 3.5, you could go ahead and try out the `@` operator in your own code.

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