MAS212 Assignment 1: **Feedback Points**

Sam Dolan (2019)

General feedback

- Read the brief thoroughly, *including the appendices*. Appendix A was the suggested structure for the code file. Appendix B was the example output from your script. Submissions that followed the format suggested in the appendices were much easier to mark, and likely received more credit.
- Shorter codes are generally better than longer codes. Avoid code repetition where possible. Use functions, loops and vectorization.
- Use comments judiciously, to explain the key points. There is no need to comment on every line.
- **Answer all sections**. Attempt as many sections of the brief as possible. Ask for a hint at an early stage if you are stuck.
- **Do** extend in interesting directions, but only after covering the main brief.
- **Do not** leave until just before the deadline to begin.
- **Proof-read** your code and comments before submission, just as you would proof-read a report.

Specific feedback

An example code is available on the course webpage.

(a) **Use `for` loops instead of `while` loops**. `while` loops are dangerous. If you forget to add break conditions, then the loop will run indefinitely, and may even crash your computer (or mine!). Use `for` loops instead, with the `break` keyword if necessary.

(b) **Return values**. Typically, a function should return something. For example, the brief asked that `NRmethod()` should return a list of the sequence of estimates, `[x0, x1, x2, ...]`. In some submissions, the function did not return this list, but merely printed the list without returning it. This meant that you could not re-use your function later. In other codes, something else was returned.

(c) **Code reuse**. Some codes defined several functions which did nearly the same thing: `NRmethod()`, `NRmethod2()`, etc. This was not necessary; it is possible to re-use the original function for multiple tasks, by passing new functions in the first two arguments of `NRmethod()`.

(d) **Decimal places**. Parts 1(b) asked you to present your results to 10 decimal places. To do this, you could either use the old-style syntax, `print("%.10f" % myvalue)`, or the new-style syntax `print("{0:.10f}".format(myvalue))`. The latter is recommended in Python 3, but the former is also widely used. It is not advisable to round the numerical values to 10dp while the calculation is in progress. Instead, calculate at full (machine) precision, and then *display* to 10dp.

(e) **Transpose**. For part 2(c) many plots inadvertently had the real axis running in the vertical direction. This could be fixed by taking the transpose of the 2D array, `arr.transpose()`.

(f) **Testing for convergence**. In part 2(c), you needed to determine whether the result of `NRmethod()` – let's call it `z` – had converged on root 1, 2 or 3 (i.e. $z_0 = 1$, $z_1 = e^{2i\pi/3}$ or $z_3 = e^{-2i\pi/3}$). Many codes tested for exact equality, i.e. `if z == z0:`. This is not wise. Two real numbers are not equal if they differ even slightly, even at machine precision, so the test can be `False` even if the numbers are close enough for convergence to have been practically achieved. Other codes checked that only the real part, or only the imaginary part, were close enough; and others used some range condition, e.g. `if z > z0 - 0.001 and z < z0 + 0.001:`. These are all suboptimal solutions. Much better is to use a condition like this: `if np.abs(z - z0) < tol:`, where `tol` is some sufficiently small number which is still larger than the tolerance used in the Newton-Raphson method. Another good solution was to use the `np.isclose()` function.

(g) **Efficiency**. Some codes took longer to produce the 300×300 plot in 2(c). The most common causes were (i) using lists instead of numpy arrays; (ii) calling the NRmethod() function 3 times within the body of the loop (in determining which root had been found), instead of calling it once and assigning the value to a variable, or (iii) some non-standard way of checking which root had been found (for instance, casting to strings); (iv) an unnecessary repeated calculation inside the loop (for example, calculating the exact root

using `np.exp()` or `np.sqrt` each time). The most efficient way was to use broadcasting, avoiding loops entirely.

(h) **Formatting**. Please format your code and output neatly. See Appendix A for the preferred code format and Appendix B for the preferred output format. In your submitted code, avoid additional output which is not relevant to the subtasks. Your script should produce human-readable output for each part of the brief. Please ensure that the person marking the code can quickly check whether each part is working as intended. The marker does not have time to edit your code.

(i) **Docstrings**. In part 1(a) you were asked to add a docstring to the beginning of the function. A docstring should appear immediately below the function definition (the first line), and should be enclosed with a triple set of double quotes (`"""`). It can run over several lines. See e.g. `http://www.pythonforbeginners.com/basics/python-docstrings`. In the docstring you should also describe the arguments and return value.

(j) **Vectorization**. In part 2(c), the code can be made more efficient by using numpy arrays with vectorization. In essence, to vectorize the code we replace the for loops over arrays with single-line statements that act on the entire array. These single-line statements then use a compiled code to loop over the array in an efficient way. Please see the example code for how vectorization can be used to find the basins of attraction.

(k) **Passing functions**. Python allows us to pass functions as arguments to other functions. In `NRmethod(f, fprime, )`, the first two arguments are functions that can be used inside the body of this function. A common mistake in the code for `NRmethod()` was to call `f1` and `f1prime`, rather than `f` and `fprime`, in the body of NRmethod. This prevented the function `NRmethod()` from being used again in part 2, as the functions `f1` and `f1prime` were 'hard-coded' into it. See also *(c) code reuse*.

Here are some additional comments on coding style:

1. **Use functions**. By defining several small functions, you can split the task into smaller subtasks. Each subtask (or function) can be tested separately.

2. **Function re-use**. Re-use your functions whenever there is an opportunity. If you need to adapt the function slightly, you could add optional arguments with default values.

3. **Code repetition / excessive length**. Consider how you could use variables, loops, functions etc., to reduce the length of your code. Brevity helps with clarity. If you find you have many consecutive lines of code which look similar, but with small changes, you should be using a loop.

4. **Do not nest functions**. Do not define one function in the body of another function – this is bad practice. Good practice is to define your functions, separately but adjacently, somewhere near the top of your code file.

5. **Do not** put import statements inside functions.

6. **Hard-coding**. Avoid hard-coding where possible. For instance, rather than 'hard-coding' an array size, use a variable instead (e.g. `n = 300`).

7. **Global variables and functions**. Although functions can refer to 'global' variables (i.e. variables which were defined outside of the body of the function and its arguments), this is not usually good practice. Try to make each function self-contained. If a function needs to use or change a variable, then include this as one of the function's arguments.