Computer Modelling Exercise 1

Python Refresher & Numpy Essentials

Due: 16:00, Thursday, Week 4, Semester 1

1 Aims

In this exercise, you will gain familiarity with `numpy`. You can learn more about `numpy` from section 3.2 of the course notes on learn, or the website `https://numpy.org/`

At the end of the exercise you will submit a single file containing a set of functions to perform some simple tasks, which we will test to see how well it works.

2 Preparation

1. If you are using your own computer, and haven't already, install the Anaconda Distribution. We will use the Spyder environment in this course.

2. Create a new folder (directory) for your work on this exercise.

3. Download the files `exercise1.py` and `test_exercise1.py` from the assignment Learn page and save them both in your new folder.

4. Select `test_exercise1.py` and run it. It should say that lots of tests have failed; this is expected, because you have not written your code yet. As you complete this exercise the tests should start to pass. **The test program does not completely test that your functions are right - you will have to do your own checking too.**

5. Every time you complete one of the tasks below, one of the tests should start to pass. You should make all of them pass by the time you submit. You can scroll up in the console window to find out exactly why tests are failing.

3 Testing Code

Most programming is debugging; this takes longer than initially writing the code. These notes may help you do this efficiently.

3.1 Interactive Testing

It is sometimes useful to be able to call your functions interactively - i.e., without writing a file, and instead typing commands directly to python. This can let you explore problems more quickly and find faster solutions. One way to do this is using the *console*.

In the console window in the bottom right corner of Spyder you can type python commands interactively. You can use this to test things by hand as you go along.

1. Click the window in the bottom left with the numbered prompts.

2. To get started, type `cd "XXXXX"`, replacing XXXX with the folder you are working in[1], e.g. `cd "My Documents\Work\CompMod\Exercise 1"`.

3. Type `import exercise1`, since that is the name of your main python file (removing the `.py` at the end).

4. If you have a function called `task1` from the first exercise below in your file, you can then run it by typing `exercise1.task1([1.0, 2.0], 1, False)`.

5. You can also create and use variables in this console just like you can in python code. That can be useful if you use the same value over and over again. You can also import other packages like numpy.

3.2 Automated tests

The file `test_exercise1.py` runs a set of automated tests of your code. In many industries every single line of code you write must be covered by a test, and sometimes the tests are even written before the code itself.

Have a look at the tests in that file, and try to understand how they work. You can write your own tests if it helps.

[1] `cd` stands for *change directory*.

4 Tasks

Add code to complete the tasks below in `exercise1.py`. Running the test program will help check if the task is complete. It will print out a listing of which tests pass and fail, and for those that fail it will print a reason why, and above that some more details.

Your code should not stop to ask for user input with `input()` - that will break the tester.

4.1 Basic Python Refresher

Task 1 Complete the function in `exercise1.py` called `task1`. This function will be given three arguments, in this order:

1. A list of floating point numbers called `x`
2. An integer called `n`
3. A boolean (True/False) value called `b`

Right now the function is empty. Modify it so that it returns this value:

$$\text{task1}(x, n, b) = \begin{cases} \sum_i x_i^n & \text{if b is True} \\ \sum_i x_i^{2n} & \text{if b is False} \end{cases} \tag{1}$$

where the i sum runs over the length of the list, and x_i is the i'th element of the list.

Hint: There's no need to do anything fancy for this, it's just to make sure you recall basic python. You could use a `for` *loop to go through every element of the list and an* `if` *statement to see if* `b` *is True or not.*

4.2 Creating numpy arrays

Task 2a Write a function called `task2` that takes as input two integer values, m and n, and returns three numpy arrays:

1. a 1D array of all zeros of size m
2. a 1D array containing the numbers 1, 2, 3, ... n
3. a 2D array of uniform random numbers in the range [0, 1) of shape (m, n)

Task 2b Write a *docstring* for your function `task2`.

A docstring is a string (text) which must appear as the first line in a python function, and is designed for users of your code to read to help them understand the function. They are usually triple-quoted strings, like this:

```
def my_function(x):
    """
    This is a docstring.

    It can be split over multiple lines, and should explain
    what the function does, and its inputs and outputs.
    """
```

4.3 Basic vector arithmetic

Task 3a Write a function called `task3a` which takes as input two numpy vectors $\mathbf{a}$ and $\mathbf{b}$ and a number t, and returns the numpy vector $2t(\mathbf{a}+\mathbf{b})$

Hint: Don't use `np.add` or `np.multiply` - these are not necessary and obscure your code.

Task 3b Write a function called `task3b` which takes as input two position vectors $\mathbf{x}$ and $\mathbf{y}$, and returns the distance between the positions.

Hint: The length of a vector is also called its norm, and numpy has a function to calculate it.

4.4 Basic vector algebra

Task 4 Write three functions called `task4a`, `task4b`, `task4c` that each return two numpy vectors, representing the left- and right-hand sides of these three vector equations:

$$\boldsymbol{v}_1 \times \boldsymbol{v}_2 = -\boldsymbol{v}_2 \times \boldsymbol{v}_1 \qquad (a)$$

$$\boldsymbol{v}_1 \times (\boldsymbol{v}_2 + \boldsymbol{v}_3) = (\boldsymbol{v}_1 \times \boldsymbol{v}_2) + (\boldsymbol{v}_1 \times \boldsymbol{v}_3) \qquad (b)$$

$$\boldsymbol{v}_1 \times (\boldsymbol{v}_2 \times \boldsymbol{v}_3) = (\boldsymbol{v}_1 \cdot \boldsymbol{v}_3)\boldsymbol{v}_2 - (\boldsymbol{v}_1 \cdot \boldsymbol{v}_2)\boldsymbol{v}_3 \qquad (c)$$

where $\times$ is the vector cross product and $\cdot$ is the dot product.

When you test these functions you might find that sometimes the two sides of the equation are not exactly the same, but are very slightly different (e.g. at the 10^{-12}

level). This is because computers round off stored numbers to a certain number of significant figures, and this difference can start to build up as you do calculations. It is why you should never compare floats (or arrays of floats) using `==`. Instead you can check that the difference between them is very small.

4.5 Gravity

Task 5 Write a function called `task5` which takes four arguments, in this order: two vector positions: $\mathbf{x}_1$, $\mathbf{x}_2$ in units of metres, then two masses M_1, M_2 in units of kilograms, and returns two values: (i) a vector of the Newtonian gravitational force on particle 1 due to particle 2, and (ii) the gravitational potential energy of the system.

Your function should include both a docstring and careful comments using the # symbol that explain anything that would not have been obvious to you, and explain why you're doing things, if relevant.

Hint: $G = 6.6743 \times 10^{-11} \mathrm{m}^3 \mathrm{kg}^{-1} \mathrm{s}^{-2}$. *Recall that you can return two things from a python function using* `return a, b`

4.6 2D arrays

We can use a 2D array to represent a matrix.

Task 6a Write a function called `task6a`, which takes an integer `n` and makes a square 2D array M of shape $n \times n$, with elements filled in so that $\mathrm{M}_{ij} = i + 2j$ where i and j start from zero.

Hint: There are some clever ways to do this, but for now just stick with two loops, over i *and* j.

Task 6b Write a function called `task6b` which takes an integer `n` as input, uses your previous function, `task6a`, and then sums its output over one axis to generate and return a 1D array $y_i = \sum_{j=0}^{j=n-1} M_{ij}$

Hint: This time, try to avoid doing a loop - look at the numpy `sum` *function, or the array method* `M.sum`.

5 Submission

To submit your work:

1. Go to the Learn page for the course.
2. Click *Assessment* then *Assignment Submission* then *Exercise 1* then *Submit Exercise 1*.
3. Click *Browse Local Files*
4. Navigate and find your file `exercise1.py` and click *Open* or *Select* (depending on your computer).
5. Click *Submit*.

You can resubmit your file as many times as you like. Only the last submission will be marked.

6 Marking Scheme

Task 1: 1 mark

Task 2: 3 marks

Task 3: 2 marks

Task 4: 3 marks

Task 5: 3 marks

Task 6: 3 mark

Explanation : 5 marks

The total mark available is 20. The tasks will be marked according to how many of your functions work as described.

The 5 marks for "explanation" are assessed by a TA or lecturer at a workshop session. You will need to attend one of the sessions after submission and will have to talk through your code and answer questions on how it works.

This assignment counts for 10% of your total course mark