Computational Physics

Practical exercises

David Buscher

Version 4.0 February 2024

Contents

Introduction

These exercises are designed to help you learn some `Python`, understand some physics and learn a little more about numerical programming. You should attempt the exercises in the order given. Demonstrators will be on hand during the practical sessions to assist.

The speed at which people can program varies widely. It even varies for individuals day by day — all programmers will sympathise with how much time can be wasted trying to fix an obscure bug. So do

keep an eye on the clock, and do seek assistance. Talk to your colleagues so you can share experience and spot each others' mistakes. The goal of the exercises is for you to learn – and you can only do this by experience.

Each exercise is split into tasks. The *core task or tasks* are ones that I hope you will all be able to achieve in the class in the time available. If you hand in a working solution to this you will get approximately 60–70% of the credit for the exercise. There is also one or more *supplementary tasks* per exercise, which take the problem further, or in a different direction. I hope that many of you will achieve some or most of these, and they count for the remaining fraction of the credit.

I have given structured hints to each problem. In addition, you should look at the code examples in the lectures and, importantly, **consult the relevant documentation for the libraries** such as `scipy` and `numpy` — finding and learning from online documentation and code examples is an important skill. There are some starting links from the course web-page or just use Google/Bing/DuckDuckGo/Kagi.

Assessment

Over the 4 weeks you should carry out the three exercises set out below. The marks available are a maximum of 6 for Exercises 1 and 2, and a maximum of 8 for Exercise 3, for a total of 20 marks.

You should upload your solution to the TiS. The **deadline** is posted on the course web site and the TiS. You are encouraged to submit your work as soon as it is in a reasonable state, and move on. So don't spend too much time on the tasks, but don't spend too little time either.

For each exercise you should submit a Jupyter notebook file (a `.ipynb` file) which is structured into tasks as explained in the next section. This notebook will contain Python code, plots illustrating relevant results, and text commentary. The text commentary describes very briefly what you did, any major problems, mentions any numerical answers required and describes the accompanying plots.

It is possible in Jupyter notebooks to execute your code cells in any order. You should make sure that, in your submitted notebook, the code will run correctly when you restart the Python interpreter and run all the cells in the order that they appear (for example using the "Resart and run all" function under the "Runtime" menu in Google Colab). This is so that the logical order that the code should be run is clear to someone reading the code. Breaking your code up into functions will help with the organisation and ordering of your code especially when you need to do different variations of the same calculation.

Note that the usual rules with respect to plagiarism apply: you are of course allowed to discuss the work with others and to refer to online or other sources, but you must clearly state where any of the work you submit is not your own work, or is joint work, and explicitly attribute the authors/sources of that work.

Code quality

In the first two exercises, the main emphasis of the marking will be on successful completion of the tasks. In the last exercise a percentage of the marks will be given for code quality (this will also be true for the optional project). High-quality code will include at minimum:

- Structured code: tasks broken down into sensible functions;
- Meaningful function names;
- Meaningful variable names (this is less important than for naming functions: single-letter variable names can be meaningful if the meaning can be inferred from context, e.g. loop counters);
- Appropriate levels of commenting (at minimum a Python "docstring" identifying what each function does);
- Sensible use of whitespace to indicate code structure.

Remember, code quality is important because (a) it helps to make the code easier to understand and debug for the person writing the code and (b) it makes the code easier to understand and debug for others who may have to modify the program later on.

Writing high-quality code is somewhat like writing high-quality prose or mathematics: clearly structuring and explaining your thoughts for someone else can help clarify your own thinking, and this clear explanation is what code quality is all about.

Getting started

Using Google Colab

We will be using the Jupyter notebooks for these exercises. You can run Jupyter on your own computer or use the notebooks provided by Google Colab at https://colab.research.google.com. The following assumes that you will be using Google Colab, but most of the advice should apply to any Jupyter notebook or Jupyter lab installation.

It is best to log in with your University of Cambridge account: you can do this by logging out of any other Google accounts you may be logged into and then logging back in using `crsid@cam.ac.uk` as your username — if you are then prompted you about a choice of accounts associated with this email address, always choose the "Google workspace account" rather than the "Personal account". You can also use Colab with any other Google account you may have, but this may give rise to problems when you want to get access to any shared files or to share notebooks with others, e.g. demonstrators.

There are plenty of tutorials available in Google Colab and elsewhere about how to use Jupyter notebooks, but an effective way to learn for many people is to just open a new notebook and start experimenting. To do this, click on the "New notebook" link in Colab. The first thing you should do after

creating a new notebook is to rename the notebook by clicking on the existing title (usually something like “Untitled0.ipynb”) and editing it to give it a descriptive name such as “Exercise 1”.

Structuring the notebook

One feature of Jupyter notebooks we will be making use of in the exercises is the use of text cells as well as code cells. Code cells are the default cell type and contain the Python code you are developing. You will use text cells to put in section headings to separate different tasks, and also to put in text commentary and conclusions.

The text cells use a format called “Markdown” which allows you to type in plain text into the cell and get various “rich text” effects such as section headings, bold, italic, and even nicely-formatted mathematics when the cell is “run”. You can find out more about markdown online, for example at https://colab.research.google.com/notebooks/markdown_guide.ipynb.

With this in mind, you should insert a text cell (you can use the + `Text` menu item near the top of the screen for this) at the beginning of your notebook that gives it a section heading, for example:

```
# Core task 1

Some text describing the task (optional - this can be used to
remind yourself what you are aiming to do).
```

When you press the arrow to “run” the cell it will turn it into a bold-font first-level heading, plus any text you typed in.

After that you should probably insert your first code cell. A suggested first cell is one which just imports all the relevant modules, e.g.

```
import matplotlib.pyplot as plt
import numpy as np
import scipy
```

You can then run this cell and in the next code cell start writing the first part of your code.

At the end of each task, you should add a text cell with a conclusions sub-heading and text in the form

```
## Conclusions

Some brief words here about what you did and what
you achieved in this task.
Any numerical results should be stated here.
```

You can of course add additional text cells in the notebook to annotate your work for yourself and/or the assessors. This section structure should be repeated for all the core and supplementary tasks.

You should submit one notebook per exercise, containing the code and results for all the tasks in that exercise.

Plotting your results

You will want to present many of your results in the form of graphs made using pyplot, which will appear below the code cell when the cell is run. **Remember to add captions for the axes and a title.** Here is a simple Python code fragment which plots a sine function with axes.

```
plt.plot(np.sin(np.linspace(-10,10,100)))
plt.xlabel("Distance along axis (m)")
plt.ylabel("Intensity (arbitrary units)")
plt.title("Diffraction pattern of a double slit");
```

Note the use of a semicolon on the last line to suppress any printed output — not essential but makes the notebook look cleaner. You can also use the "object-oriented" pyplot plotting style used in places in the lectures, which can be helpful when you are plotting multiple plots in the same figure.

It is usually a good idea to do the long-running part of any calculation in one code cell and then do the plotting in a subsequent code cell. This way, you can change the plotting code and re-run it to improve the appearance of the plot and add labels etc without re-running all of the calculations.

If you are doing very-long-running calculations you may want to save the results to a file to analyse/-plot later. You can use `numpy.savez()` function to do this.

Uploading your notebook to the TiS

You should use one notebook per exercise. When you are finished the exercise and are ready to submit it for assessment, you can use the "Download .ipynb" item under the "File" menu in Colab. You can then upload the file from your computer to the TiS.

Exercise 1: The Driven Pendulum

Goal

To explore the physics of a non-linear oscillator (a damped, driven pendulum) by accurate integration of its equation of motion.