1. Introduction & tools

The goals of Computational Physics are to learn the principles of using computers to perform calculations in physics. These principles include how numbers are represented, and the basic tools for calculating special functions, random numbers, linear algebra, interpolation, root-finding and optimization, differentiation, integration, and spectral analysis.

It is helpful to understand what we will not be doing:

- We will not here be considering explicitly problems in physics data analysis, but many of the techniques we learn here will be applicable in that context.
- I will not be teaching you the principles of software engineering. To the extent possible, I will emphasize the importance of documentation, modularity, validation, and version control in writing stable, maintainable code. But do not underestimate the level of experience, discipline, and effort that good software engineering requires, and respect it when you see it.

I will lecture and demonstrate in class, and there will be homeworks most of the semester. There will also be a set of large-ish projects. You will work on these in pairs, and you will let me know which teams you are on by Sept 26, at which point I will assign topics.

Your recitations will be spent working on homeworks and projects, with the TA available to help. The first recitation will be spent setting up your laptops appropriately to perform the work for the course.

We must choose a language to work in. There are many available computer languages. Useful ones are C, C++, Fortran, Java, Python, and Julia. There are other field-specific languages in use as well (e.g. in astronomy there are IRAF and IDL).

Here we will use Python, mostly because it is a modern language of broad applicability, which also allows easy visualization within the language itself. It has limitations: you would probably not want to write a very computationally heavy simulation in Python, but more likely directly in C++. But for many calculations it works fine and is convenient, and in any case in this course we will concern ourselves mostly with the concepts of computational physics rather than aiming at the most efficient possible implementations.

In addition, you will use a quite useful tool called Jupyter Notebooks. This is a browser-based tool, which allows a similar interface to work on the command line, but retains a clean record of what you did. It is good for testing code, for exploratory work, and for presenting and sharing results. The Python Data Science Handbook has a good introduction to their use.

It is not good for building a big pipeline or software system. For that it is better to use a more specifically designed "Integrated Development Environment" or IDE. Within Unix you also have its tools plus editors like Emacs. I mention this because I want you to know that eventually

(even within this course) you will need to graduate to building Python modules or other software systems outside the Notebook environment.

Unfortunately, for a class like this we have to go through the basics of getting everybody set up on their computers and using the tools of the course. So there is some boring stuff to get through before we get to the fun stuff.

First, let me note that there is a class website, also found linked from NYU Classes. This will have important resources including my class notes.

I want to describe what each of you will need in your environment. The end point requirements are:

- git
- Latex
- Python
- Jupyter

and as long as those work on your system you should be OK. In principle, these should all be able to work on Windows, Linux, or Mac OS X. Probably Linux and Mac OS X are the easiest though.

For `git`, you will need to install the `git` software and you will also need to create a GitHub account. You should create this account, and then start a GitHub repository specifically for this course following the instructions on github. I recommend that you structure your repo as follows:

```
cp210/
 scr/
 tex/
 homeworks/
  ps-[n].ipynb
```

where you can stored scratch work you want to save in `scr`, any work within Latex within `tex`, and put your homework for the TA to find them in `homeworks`, named as described.

For `Latex`, you will need this for your class report. On the class web site is a basic example of a report. You can use this to get started if you are running Latex on your laptop, and sharing with your team through Github. The other option is to use Overleaf, which is a cloud-based Latex document system, also good for sharing.

To install Python and Jupyter Notebooks on your computer and enough of its packages to function, I recommend the Anaconda Python 3.6 (or later) distribution. Other packages can be installed as described in the book. Key for all science done with Python is the NumPy package.

I will illustrate on the projector how a Jupyter notebook works and some of the basic features of Python.

2. Python: an Object Oriented language

Python is what is referred to as "Object Oriented." There is a lot of baggage associated with this classification, but the basic idea is that entities in Python are not just variables and functions, but also objects (in fact, the variables and functions are themselves objects).

I illustrate in the notebook how an object class is defined and used. If you are used to procedural programming, object-oriented programming requires some getting used to. You can use Python (almost) purely procedurally, but using object classes gives you new ways to make your code modular — if you are careful. Also, NumPy and other packages do force you to use classes.

Some language is useful. In particular, objects have "attributes" and they have "methods." Attributes are just quantities that are set for an object instance. Methods are functions that the object instance can run.

3. Representation of integers on computers

A basic fact about computation that you need to understand regards how numbers are represented on computers. All information stored on computers is stored in units of "bytes."

What is a byte?

A byte is 8 bits. Each bit can be 0 or 1. It is like a Boolean: either 0 or 1, or False or True. You can express a byte as a series of eight bits, for example this one `01000110`.

How can integers be represented by bytes?

The sequence of bits can be interpreted as a base-2 number. That is, the integer can be generated from the 8 bits as:

$$i = \sum_{i=0}^{7} b_i \times 2^i \tag{1}$$

E.g. assuming the "lowest significance" bit is on the right side of 01000110, this becomes:

$$i = 2^1 + 2^2 + 2^6 = 2 + 4 + 64 = 70 \tag{2}$$

If you store a number on a computer this way it is known as an "unsigned 8-bit integer."

What is the highest integral that can be represented by a byte in this way?

It is set by the following formula using $n = 8$:

$$i_{\max} = \sum_{i=0}^{n-1} 2^i = 2^n - 1 \tag{3}$$

which is $2^8 - 1 = 256 - 1 = 255$. If you try to increment an unsigned 8-bit integer beyond this limit, you will wrap back around to 0.

How might I represent negative numbers?

Usually this is done by reassigning the upper range (in this case 128...255) to the negative range (in this case −128...-1). If this is done, then the data type is known as a "signed 8-bit integer."

There are two interesting things this does. First, because it is the upper half of the range, for negative numbers the highest significance bit is always set. Second, it means that if you are incrementing a signed 8-bit integer and go beyond 127, you will "wrap around" to -128.

Most programming languages have a range of different integer types that you may use. Specifically, 8-bit, 16-bit, 32-bit, and 64-bit are the common ones (in addition to Boolean, or 1-bit). These can be used to represent increasingly large ranges of signed or unsigned integers.

32-bit is the usual standard integer type. However, an unsigned 32-bit integer can only count up to about 4 billion — this can actually be a practical limit.

4. Representation of floating point numbers on computers

Of course, for computation we need to be able to express non-integers. Bits can be used to do so. The resulting numbers are known as "floating point" numbers, e.g. in NumPy `float32` (single precision) or `float64` (double precision), depending on the number of bits. Please note that these definitions of single- and double-precision are *mostly* standard these days but you may run into older definitions at times.

The book makes a big point about all modern computational physics relying entirely on 64-bit precision. This reveals a limitation in these authors' experience: there are plenty of contexts where 32-bit precision is sufficient and 64-bit precision would be wasteful of computing time, I/O time, or memory. You cannot escape the use of judgment in determining whether or not to use high precision or not! It will not matter much in this course however.

The IEEE has defined standards for storing floating point numbers. To express a large enough dynamic range (more than just a factor of a few billion for 32 bits), floating point numbers use something akin to "scientific notation." Just like we write numbers like:

$$1.677 \times 10^{-34} \tag{4}$$

to avoid the use of a lot of zeros every time we write them down, a similar technique is used when expressing numbers on computers. In particular, there are three types of bits. First, there is a "sign" bit (±). Second, there is mantissa, which is the analog of the "1.677" above. Third, there is the exponent, which is the analog of the "−34" above.

More specifically, floating point numbers are broken up (e.g. for 32-bits) in the following way: bit 31 is the sign bit s; bits 30–23 express the 8-bit integer exponent e, and bits 22–0 are the

mantissa f. For a normal number, its value is:

$$\pm \left(1 + \sum_{i=0}^{22} f_i 2^{-(i-23)}\right) 2^{e-127} \tag{5}$$

In reality, normal numbers are cases where $1 \leq e \leq 254$. Special cases are :

- $e = 0$, $f \neq 0$, so called "subnormal" numbers, where the "1" above is replaced with 0, and $e - 127$ by -126.
- $e = 0$, $f = 0$, a signed zero
- $e = 255$, $f = 0$, a signed ∞
- $e = 255$, $f \neq 0$, "NaN" or "Not a Number" (e.g. the result of 0/0).

Inherent in any finite-bit representation on a computer is the effect of truncation error in calculations. This is of enormous importance to understand, because it affects the accuracy and stability of numerical calculations.

What is the range of numbers that can expressed in single precision?

The highest number is about 2^{128} (i.e. with exponent of 127, and the mantissa all 1s) or 3.4×10^{38}.

The lowest number is about $2^{-126-23}$, i.e. a subnormal number with $e = 0$, with only the lowest significant bit set, or 1.4×10^{-45}.

Imagine adding a small number (10^{-8}) to a larger one (1) on a computer. How does truncation error affect this?

Broadly, it will mean a loss of precision in this addition; in this specific case, you will lose all precision, because the result will be exactly one at single precision!

This happens because when the computer adds these numbers, it first needs to get them to the same exponent. Once that is true, it can just add their mantissas.

But if there is too large a dynamic range in the numbers it means that the mantissa shifts so far to the right that it is completely zeroed out. For single-precision, this 23 bits, or a relative amplitude $2^{-23} \sim 10^{-7}$. In this case, adding a number smaller than about this to 1 will result in no change at all to one, which I will demonstrate here!

Note that it is *relative* scale of the numbers that matters. As long as you are within about 10^7 of the edge the range of single-precision numbers, it isn't the absolute scale of the numbers that matters.

Finally, we're not going to be learning about exactly how arithmetic is performed on computers, which is an entire curriculum on its own. We just need to know this particular feature because it has such a great effect on our calculations.

If I have a large set of small numbers (e.g. 100 million numbers all of order 10^{-8}, how does this affect how the computer should best combine them to minimize truncation error?

It should definitely not just start adding them up one by one. By the time the sum has reached close to unity it will start losing precision — only a little bit per increment but it adds up. This is catastrophic in this particular case for single precision.

Doing this at double precision is better behaved, but you will notice that it still loses precision unnecessarily.

If I do this with a NumPy method `sum()` to add up elements of an array, I get back much better answers in both single- and double-precision. The summation in this case is done in a more stable manner apparently!