Numbers on Computers

1. Introduction & tools

The goals of Computational Physics are to learn the principles of using computers to perform calculations in physics. For example, using computer simulations of the laws of physics enables us to calculate (and therefore test) the predictions of the cosmological theory of the formation of galaxies; one recent example is from the EAGLE simulations.

The principles that underlie these techniques 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 or triplets, and you will let me know which teams you are on by Sept 21, 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.

The homeworks will be handed in in the form of Python scripts and results. You will need to have an editor to edit your Python files and a system to run them. Typically these days people use an Integrated Development Environment (IDE), like Spyder or VSCode. Although I am quite prescriptive about how you will hand in projects and homework, I do not have strong opinions about what editors or IDEs you use.

I will show some examples in 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 results. The Python Data Science Handbook has a good introduction to their use. But you should not get in the habit of using them for final versions of research-grade software, which is why I do not accept homework handed in as notebooks.

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 Brightspace. 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`

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 (which is Unix-based) are the easiest though; honestly if you are a Physics graduate student (at least in the current era) you really should have familiarity with and access to the Unix operating system.

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 called `phys-ga2000` specifically for this course following the instructions on GitHub. The first problem set will describe how problem sets will be handed in.

You will need to use LaTeX for the homeworks and 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. If you use Overleaf for your homeworks, you will upload the PDF to the problem set directory.

To install Python and Jupyter Notebooks on your computer and enough of its packages to function, I recommend the Anaconda Python 3.11 (or later) distribution. Other packages can be also be installed using Anaconda's `conda` utility or `pip`. 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?

How can integers be represented by bytes?

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

How might I represent negative numbers?

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.

Many books claim that all modern computational physics relies entirely on 64-bit precision. Such statements reveal a limitation in the 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.

IEEE 754 defines 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:

$$1.677 \times 10^{-34} \tag{1}$$

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 ($\pm$). 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:

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

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?

How does the computer add or subtract two floating point numbers?

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

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?

Why shouldn't I just always use the highest precision version of a number that I can?

Can you think of a way to multiply two numbers?

The way that computers multiply numbers is more complicated than addition obviously. There is a simple way to multiply integers, which is basically long multiplication: you multiply each bit of the first number by the bits of the second number; each multiplication produces a shifted version of the second number; then you add them. But this is fairly inefficient. There are improvements in this method that modern CPU implement in hardware that prevent performing so many additions by first storing all of the results from the pairwise bit multiplications, and then recombining them cleverly, leading to a scaling of $\log N_b$ in the number of bits N_b.

Multiplication of floating points is more complicated of course because you need to add the exposures, worry about overflow etc.

What about dividing?

Floating point (or integer) division is tricky numerically for the same reasons it is tricky by hand. When computers see a/b, the usual approach is to calculate $a \times (1/b)$, reducing the problem to calculating $1/b$.

To calculate $1/b$, the approach uses a root-finding approach called Newton-Raphson that we will learn about later, and which it turns out has a very simple form for this case.

I hope I have impressed on you some of the complexity of even the simplest calculations you have to do—entire courses are taught on (and careers built on) the implementation of just what I cover here. For this course, we will blithely move on, and take for granted all of the hard work that computer scientists have done for physicists and others.