WHRHS HillsHack 2018
Computational Science Coding Exercises

Matthew Carbone

The following will outline two exercises for you to attempt during this workshop. *Please do not feel pressured to complete or even necessarily understand everything going on here*, as this is relatively advanced material, and the point is that you struggle a bit! That's why we're all here. You should feel free to discuss any issues or questions you may have with me or your peers. This work is *not* intended to be completed alone.

Furthermore, please note that this workshop is geared towards computational science applications, not computer science, so we will not focus so much on being most optimal with the way we code. We will focus on the bare basics of the code, and how we can tame it to do what we want. In other words, this is probably not rigorous from the standpoint of the computer scientist or software engineer, but it is from the perspective of the physical scientist. Here are the two exercises.

I Introduction to Monte-Carlo

A critical benchmarking technique in the physical sciences is called Monte-Carlo. It utilizes the power of random sampling to determine quantities that are otherwise well defined, but difficult to calculate. We will use a very simple example to demonstrate this technique.

Task: Estimate the area of a circle, circumscribed inside a square of side length s by randomly sampling points within the square.

What you do know is the following. A circle is defined by a radius r, and a circle circumscribed inside a square of side length s has a radius $r = s/2$. If a point (x, y) randomly sampled within the square satisfies

$$\sqrt{x^2 + y^2} \leq s/2,$$

then that point is contained in the circle. Here are some things you can do:

1. Write a Python function that takes the side length s as one argument and N, the number of points you wish to sample, as another. It should output your estimation for the area of the circle.
2. You already know analytically that the area of a circle is πr^2, so compute the error in your computation for a given N. Write a function to do this! Then, figure out how to plot the error as a function of N. What do you observe?
3. Think about how you can use symmetry to make this process more efficient.

II Horner's Algorithm

Suppose you have a function $f(x)$, and this function is a very large (high-degree) polynomial. Maybe it is something like

$$f_n(x) = c_0 + c_1 x + c_2 x^2 + ... + c_n x^n,$$

where n is a very large number, like 50000, and $c_0, c_1, ..., c_n$ are just constants that you can choose.

Let's say you want to evaluate this polynomial (just find the value $f_n(x)$ for some x). One might imagine that this is not an easy thing to calculate, let alone even write down on the page, because of the sheer number of terms. Fortunately, we can write computer code to do it for us, but how? To really get into the details of how we should do this, we need to understand the concept of a *floating point operation* (FPO). In this context, a FPO is an addition, subtraction, multiplication or division of two objects. For instance, $c_1 x$ requires one FPO to execute. A term like $c_4 x^4$, deceptively requires four FPO's. Why? Because the computer doesn't know any other way to break down the operation x^4, except by doing $x \times x \times x \times x$. Thus,

c_4x^4 requires 4 FPO's, and in general, the mth term will require m FPO's to execute.

Task: Compute $f_n(x)$ (with the simplification $c_i = 1$ for all i) for an arbitrary n in the shortest amount of computer time possible.

This is of course a coding exercise, so I will walk you through to the solution of our "shortest time" possible. First, let's look at how many FPO's the above method requires.

- If $n = 0$, we have no FPO's, since $f = c_0 = 1$.
- We have only one FPO if $n = 1$, since our constant $c_1 = 1$ and therefore there is no operation to evaluate there, but we still have to add the two terms together, which counts as one FPO.
- What if $n = 2$? Three FPO's. Why? Because we get one from multiplying $x \times x$ and two from adding the three terms together.
- If $n = 3$ we have the three FPO's from before *plus* the two from the $n = 3$ term, plus one more addition.

So in summary, for some n, the number of FPO's contributing from multiplications, additions and all in total are shown:

$$f_n(x) = 1 + x + x^2 + ... + x^n,$$

n	$f_n(x)$	$\text{FPO}_\times$	FPO_+	$\text{FPO}_\times + \text{FPO}_+$
0	1	0	0	0
1	$1+x$	0	1	1
2	$1+x+x^2$	1	2	3
3	$1+x+x^2+x^3$	3	3	6
4	$1+x+x^2+x^3+x^4$	6	4	10
5	$1+x+x^2+x^3+x^4+x^5$	10	5	15
6	$1+x+x^2+x^3+x^4+x^5+x^6$	15	6	21
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$
n	$\sum_{m=0}^{n} x^m$	$\sum_{m=2}^{n} m - n + 1$	n	$n(n+1)/2$

The above are some rough calculations that I did which show that the amount of FPO's this algorithm requires as a function of n scales quadratically with n. If we assume (reasonably) that as the number of required FPO's goes up, so does the amount of time required to run the program, we may say that the running time of calculating this polynomial t is proportional to n^2:

$$t \propto n^2.$$

Can we do better? The answer is yes. Horner's algorithm groups polynomial terms in the following way; you may confirm for yourself that analytically the groupings are correct. The groupings severely reduce the scaling, as we can see by inspecting the following chart.

n	$f_n(x)$	$\text{FPO}_\times$	FPO_+	$\text{FPO}_\times + \text{FPO}_+$
0	1	0	0	0
1	$1+x$	0	1	1
2	$1+x(1+x)$	1	2	3
3	$1+x(1+x(1+x))$	2	3	5
4	$1+x(1+x(1+x(1+x)))$	3	4	7
5	$1+x(1+x(1+x(1+x(1+x))))$	4	5	9
6	$1+x(1+x(1+x(1+x(1+x(1+x)))))$	5	6	11
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$
n		$\max(n-1,0)$	n	$\max(2n-1,0)$

Looking at the total number of FPO's required now, we see that the scaling has been reduced by a simple algorithmic trick, from $n^2 \to n$, leading to

$$t \propto n.$$

That is the beauty of Horner's algorithm. It is a purely analytical simplification that drastically reduces the scaling of a process that one might assume at first glance, is trivially simple.

Here are some things you can do:

1. Write Python functions to calculate the polynomial $f_n(x)$ with $c_i = 0$ for all i, using both the brute force method (the first one discussed) and Horner's algorithm. The function should take arguments x and n.

2. Compare the time it takes to execute them as a function of n.

3. Write a more general function which takes a list containing the arbitrary coefficients c_i (which are now not taken to be 1's), and computes $f_n(x)$ (using Horner's algorithm).