1. Why we need arrays of numbers

Last time we discussed data types, in the context of single numbers. However, if we are bothering to perform a computation with a computer, we probably have more than one number to deal with. For example, we might have a million numbers, which we want to either apply the same computation to, or combine in some way.

Thus, all computing programming languages have a way to store arrays of numbers. Often these can be thought of as a one- or more-dimensional matrix. In most cases, the array is implemented as a series of memory addresses containing numbers ordered sequentially. E.g. a section of memory containing an array of 4-byte floats might look like:

Address	+0	+1	+2	+3
0	00010001	01010101	00101010	11111100
4	10111001	11000001	11011010	11010100
8	01010011	10001011	01001110	01110101
...	...	...	...	...

What formula tells you the address of the ith element of an array?

To find the address of the ith element of an array ("`a[i]`") you can go to address:

$$\texttt{a[i]} = \texttt{a[0]} + N_{\text{byte}} \times i \tag{1}$$

where N_{byte} is the number of bytes in each element. Obviously this requires all the elements in the array to be the same size.

I have assumed here "0-indexed" arrays — i.e. the first element is `a[0]`. This is true of many languages, including C and Python. However, Fortran and some other languages use "1-indexed" arrays. It is important to know which indexing you should be using!

2. Multidimensional arrays

Virtually all computers have one-dimensional addressing of memory. So in order to express a two-dimensional array, the computer reorders them one-dimensionally, e.g. with row 0, then row 1, etc.

What formula tells you the address of the (i, j)th element of an array?

If the shape of the array is (P, Q), then the address of element (i, j), in Python denoted `a[i,j]` becomes:

$$\texttt{a[i, j]} = \texttt{a[0]} + N_{\text{byte}} \times (i \times P + j) \tag{2}$$

Another thing that tends to differ among languages is how what the ordering of (i, j) implies about which numbers are next to each other in memory; that is, whether I need to write $i \times P + j$ or $i + j \times Q$ in the above equation.

There is a lot of confusing documentation out there about "rows" and "columns." In 2D arrays the "row" is indicated by the first index the "column" is indicated by the second index. "Row-major" order is the order shown in the equation above — the bigger steps in memory are the rows. "Column-major" is the opposite.

For C, the convention is row-major, as shown above. The Fortran convention is column-major.

Now, let us consider NumPy. There is a default behavior of NumPy which is C-like (row-major, like the example above). However, you can also set NumPy to act differently. As I show in the notebook, you can actually get it to show you the "strides" each index implies in memory space.

Why am I emphasizing the location of the array elements in memory? It is because when you are dealing with large arrays in memory (and on disk) it really matters where the data is located. Your computers CPU needs to get information from its memory (RAM). The RAM is typically about a few Gbytes, but accessing it is slow; you don't want to go fetch every byte of information individually. Instead, what your computer does is gets big sections of memory from RAM simultaneously and put it into a cache near the CPU, where the access time is fast. Usually there are multiple layers of cache, with increasingly large sizes but slower access, up to 100 Mb or so. There is all sorts of strategy associated with this, done at the level of the compiler, optimizer, and in the CPU. We won't cover these issues now.

But the order in which you sort your data on the computer is something you typically control at a high level, that really impacts this process. In the examples I show in the notebook, you can see that the order matters quite a bit. Again, in many languages an optimizer will catch this and fix it (when it can be sure that it won't change the answer) but you need to be aware of this.

How do you express 3- or more-dimensional arrays?

This is just a generalization of the 2-dimensional case:

$$\texttt{a[i,j,k]} = \texttt{a[0]} + N_{\text{byte}} \times (i \times (P \times Q) + j \times Q + k) \tag{3}$$

for an array of shape (P, Q, R).

3. NumPy arrays

Now let's look closer at NumPy specifically. The first thing to note is that NumPy arrays are completely different than Python lists. They use similar syntax but they are very different objects. Lists are much more flexible and correspondingly less efficient for numerical computation.

The NumPy array object class is called `ndarray`. It is quite flexible, but does assume every

element of the array has the same `dtype`.

One of the nice things about NumPy arrays are that they allow functions to act on them as whole objects, or on some range of their elements. This makes programming simpler and also can allow the implementation to run more efficiently, as we will see. I'll note that Python is not the 100% best programming language for this sort of computation; the new language Julia is probably better, and Fortran 90 is excellent in this respect.

In the notebook I demonstrate the various ways in which arrays can be accessed and used. A more full demonstration can be found in the PDSH, Chapter 2 (this is true of the rest of the discussion here).

There are also a set of "ufuncs" for NumPy arrays. These are generally much faster to apply than by looping through each array element with a `for` loop. What happens in a `for` loop is that at every iteration, the machine checks the types of variables it is operating on and performs other bureaucratic tasks, which if it *knew* it was going to do over and over again, it would only do once. Within the ufunc it can assume that, and save a lot of time. It also allows the computer to vectorize at a lower level as well (pipelining multistep operations so it is working on multiple array elements at once).

We demonstrate several other methods as well:

- `min()`, `max()`
- `sum()`
- Comparisons ($<$, $>$, ...)
- Indexing
- Sorting

4. Quantifying the scaling of algorithmic cost

It is important when you are considering any implementation of a computational scheme to quantify how expensive that computation will be. Usually the most important factor is how the algorithm *scales* with problem size. This scaling is usually denoted by the $\mathcal{O}()$ notation.

Sorting is a good example. Above, we just called a "sort" function that did all the hard work in a clever way, since sorting is an ancient algorithmic problem in computer science. However, there are also dumbs way to sort.

One way to sort that you would probably think of first is called "insertion sort." It is basically how you sort a hand of cards. You start with the first two elements of the array and make sure they are sorted properly. Then, you check the third element, and insert it in the right place by scanning the already sorted elements (the first two). Then you just repeat this. Even if you are clever about dealing with the insertion steps, this means that for each of the N elements of the

arrays, you need to scan on average $\sim N/2$ other elements. This means the number of comparisons in the sort is $\sim N^2/2$. This is what we refer to as an $\mathcal{O}(N^2)$ algorithm.

NumPy's `sort()` method doesn't give you insertion sort as an option. It allows a few others, like `quicksort`, that have much better scaling. `quicksort` goes as follows:

1. Choose one element in the array as a pivot.
2. Partition the array so that values less than the pivot are on one side, and values greater than are on the other (this step is $\mathcal{O}(N)$).
3. Recursively apply (1) and (2) to the subarrays on the left and right of the pivot.

Implementing this efficiently is still tricky, but notice that the way the problem is structured is that you need to perform an $\mathcal{O}(N)$ algorithm around $\log_2 N$ times. So it scales as $\mathcal{O}(N \log N)$.

For large N there is a very large difference in cost!!

`quicksort` was invented about 60 years ago. It is important to remember as you do computational physics that there are a *lot* of tricks computer scientists have learned over the years. You should not be inventing them all yourself.