BRYN MAWR COLLEGE

Department of Physics

Computational Module 2 – Numerical Errors and Computational Speed

Prerequisite modules: Module 0; Module 1

Estimated completion time: 2-4 hours

Learning objectives: understand sources of error in numerical computations, become familiar with simple loops and vectorization; learn how to characterize the scaling of computation time with the size of the data set being analyzed, and see one way computation time can be measured.

Image credit: www.qwantz.com/index.php?comic=2886

This module presents some basic concepts of computing and data representation. It discusses how numerical computations are inherently limited due to the fact that a computer can represent a number to only a finite number of decimal places. It also describes limitations on computational speed, introducing a special notation for representing how a computation scales with the size of the data set processed. The module also notes a method available in Python and some other computing languages for speeding up some types of computations.

2.1 Numerical Errors

While a computer can store an exact representation of an integer (up to the limit of the computer's memory), that is not the case for floating point (real) numbers: they can be stored only to a limited number of decimal places. This limit to the precision of stored numbers is the phenomenon of ***numerical rounding error***, usually represented by the Greek letter *epsilon* (ϵ). There are numerous ways to determine a machine's numerical error. In Python, the easiest way probably is to run the following code (which calls the `float_info` function from the `sys` package):

```
from sys import float_info
float_info.epsilon
```

Breakpoint 1: Try this in whatever (I)Python implementation you're using. What do you get?

Breakpoint 2: To see an actual example of numerical error, try running this simple calculation

```
a = 1/49.0 * 49
print("a = %.20f" % a)
```

What's the output? In what decimal position is there an error? Does this match with what you found in the previous breakpoint? Explain.

Rounding error can lead to problems when comparing a variable to a floating point number. For example, if we want to check if a certain variable `x` has the value 4.7, the natural syntax would be `if x == 4.7`. However, even if `x` was defined with that value, it may be stored as something like 4.699999999999999, in which case the equality check would fail. Thus, such comparisons must always be done within a small "window"; e.g., `if abs(x - 4.7) < 1e-12`. Here, `abs` takes the absolute value of its argument (since we don't care if `x` is stored as a slightly smaller or slightly larger number), and `1e-12` means 10^{-12} – a somewhat arbitrarily chosen small number. (An appropriate window to choose in an actual application would depend on the computer's epsilon and on the desired precision of the computation.)

Generally speaking, it is safe to model the rounding error in any floating-point number stored by a particular computer as being directly proportional to the value of that number. The constant of proportionality is determined by the number of decimal places the computer uses to store numbers. If, for example, floating-point numbers are stored to 16 decimal places, then the error in a number x is roughly ϵx, where $\epsilon \sim 10^{-16}$. The errors resulting from the addition/subtraction or multiplication/division of stored numbers can be obtained using the standard rules of error analysis.[1]

[1] See, e.g., Ch. 3 of *An Introduction to Error Analysis* by John R. Taylor.

Breakpoint 3: Let ϵ be the value you got in *Breakpoint 1.* If you were to make that fractional error in measuring the radius of the Earth (6371 km), what would your absolute error be (in meters)?

In addition to the problem of rounding error, you should be aware that there are largest and smallest numbers that can be stored in a computer: $2^{1024} \simeq 1.8 \times 10^{308}$, and $2^{-1022} \simeq 2.2 \times 10^{-308}$, respectively. Exceeding these limits leads to an overflow or underflow situation, in which a too-large variable acquires the value `inf` (for infinity) and a too-small variable simply is set to 0.

2.2 Computational Speed and Big-O Notation

The speed with which a computer program runs depends on how many basic operations it has to perform. These operations include simple mathematical ones, such as addition and multiplication, as well as other, more complicated types, such as searching through or sorting lists. The speed also obviously depends on the size of the set of data the algorithm has to analyze: the larger that set, the longer execution would be expected to take.

The efficiency of an algorithm often is described in terms of a mathematical notation called ***big-O notation***. The details of big-O analyses (which can get a little technical) are not critical to us, but the basic idea is that the big-O rating of an algorithm characterizes how its computation time depends on the size of an input data set as that size becomes very large. Suppose that your program is to analyze a data set of size n – this could represent n numbers to be added, n names to be alphabetized, etc. If the time to run your algorithm was *linearly* proportional to n, then the code would be said to have a big-O of n, which would be denoted by $\mathrm{O}(n)$ (the "O" in this notation refers to the "order" of the algorithm). In this case, if you doubled the size of the data set, the algorithm would take twice as long to run. This would be the case for an algorithm that simply added n numbers together. [Of course, one could design a dumb algorithm – e.g., one that added, then subtracted, then re-added some numbers – but its efficiency would be worse than $\mathrm{O}(n)$.]

An algorithm whose run time depended *quadratically* on the size of the data set (as it became very large) would have a big-O of n^2, which would be written as $\mathrm{O}(n^2)$. In a case like this, doubling the size of the data set would *quadruple* the computation time. An example is an algorithm to add two $n \times n$ matrices. Such an algorithm would be slower – i.e., less efficient – than one that was $\mathrm{O}(n)$ for large data sets. Thus, the "stronger" the dependence on n, the longer it will take code to process a large data set. [There is, of course, a limit to how efficiently a given type of computation can be done; e.g., the addition of two $n \times n$ matrices could not be done faster than $\mathrm{O}(n^2)$].

Besides these types of power-law functions, another function that commonly characterizes algorithmic computation times is the logarithm of n, $\log(n)$. One situation in which this function arises is when performing a "binary search" through an ordered list. For example, suppose you want to find someone named Smith in the telephone book. The way *not* to do it is to start at the beginning

and go through all of the names in order until you reach Smith! Such a brute force search would be $O(n)$, since if you doubled the number of names in the phone book, that would double the time to reach Smith. A cleverer way would be to open the phone book to somewhere near where you expect Smith to be, and compare a name on the page with that target name. Suppose the name you find is Tran; you now know that Smith is in the part of the book before the current page, so you jump, say, 20 pages toward the front and find a new name for comparison. If the new name is Sanchez, then you know Smith is between the current page and the previous page, so you jump toward the back by, say, 5 pages. And so on. Properly designed, such a binary search, which in each step splits the group of elements under consideration into two groups – one in which the target item is known to lie and one in which it doesn't – will find the target name in $\log(n)$ time; that is, it's $O(\log(n))$. Since $\log(n)$ grows more slowly than n as n increases, this method of search is faster than the brute force search for large n. Note that these conclusions do not refer to any particular base for the logarithm: it could be base-10, or base-e (the natural log), or some other.

In addition to the polynomial and log behaviors already discussed, two other big-O behaviors arise often: the *exponential* case, with $O(2^n)$ (the base could be a number other than 2), and the "n *log* n" case, $O(n\log(n))$, which comes up in other search algorithms, among other situations.

It's important to mention that the big-O rating of an algorithm is only a sort of "order of magnitude" estimate of its efficiency. For example, if the time taken by an algorithm to perform some computation depended on n according to the function $4n^2 + 10n$, this function still would be rated as $O(n^2)$ since, for very large n, the quadratic term would dominate the linear one. Thus, the big-O rating of an algorithm depends only on the dominant term in its large-n behavior.

These examples point to two notable conclusions about big-O notation:

> If an algorithm's dependence on the data set size n is a polynomial with degree r (i.e., if it has the form $a_r n^r + a_{r-1}n^{r-1} + \cdots + a_1 n + a_0$), then the algorithm is $O(n^r)$.
>
> Any logarithm of n grows more slowly (as $n \to \infty$) than any positive power of n. Therefore, an algorithm with $\log(n)$ dependence on data set size is $O(n^k)$ for *all* $k > 0$.

Breakpoint 4: What would be the big-O of an algorithm whose computation time was described by the function $5n^3 + n\log(n)$?

Table 1 shows the most common big-O behaviors and some of the applications characterized by those behaviors.[2]

[2] From the Wikipedia page on big-O notation.

Order	Name	Example Application(s)
O(1)	Constant	Determining if a binary number is odd or even
O(n)	Linear	Finding an item in an unsorted list or array
O(n^2)	Quadratic	Some sorting algorithms (e.g., "insertion sort," "selection sort")
O(n^3)	Cubic	Naive matrix multiplication
O(log(n))	Logarithmic	Finding an item in a sorted array with binary search
O(n log(n))	Loglinear	Fast Fourier Transform, "heapsort"

Table 1: Common big-O orders and some of their applications. See http://xlinux.nist.gov/dads// for a big list of algorithms.

Breakpoint 5: Write the different big-O orders in the table in order, from fastest to slowest (or shortest time to longest time).

How is big-O determined for a line or block of code, or a program? It depends on the particular types of code statements involved. For example, in the case of simple statements (i.e., ones that don't depend on the data set size, such as the declaration or computation of a constant), e.g.

```
hbar = 6.626E-24
angles = [pi, 2*pi, 3*pi, 4*pi]
```

each statement takes a fixed amount of time. Thus, the time for a block of such statements is constant (just the sum of times for each of the statements to execute), and so is labeled O(1).

A loop of n repeated cycles through a set of simple statements is O(n): each cycle is O(1) but there are n of them, so the execution time scales as n.

A ***nested loop*** consists of one loop inside another, having the general form (for two loops):

```
Loop through n cycles
   Loop through m cycles
      [simple statements]
```

Such a structure might be used to construct a matrix, for example. In this case, based on what was just described for a single loop, the inner loop which repeats m times would be O(m). This loop is repeated n times by the outer loop, so the overall complexity of the structure is O($n \times m$). Often in these cases $m = n$, so such a nested loop would be O(n^2). (This is the approach one would use to add two matrices without the option of vectorization, which is discussed below.)

In the case of an `if-else` (or `elif`) structure consisting of simple statements, only one of the conditional blocks will execute, so the time is determined by the *slowest* case. If, for example, the `if` condition block is O(1) and the `else` block is O(n), then the entire structure is considered O(n).

Breakpoint 6: What would be big-O for three nested `for` loops in which the outer one repeats p times, the one inside that repeats q times, and the innermost one repeats p times?

2.3 Vectorization

In scientific applications of computing, we often work with collections of objects, such as vectors and matrices. Python's natural collection of objects is a ***list***. This is just a set of objects that Python thinks of as being grouped together. Notably, they do not all have to be of the same type. For example, `stuff = [3,'text', 7.569]` is a perfectly good list in Python. Items can be added to the end of a list using the `append` function, as in `stuff.append(42)`, giving `stuff` as `[3, 'text', 7.569, 42]`. Objects in a list can be removed using the `pop` function: `stuff.pop()` removes the last item; `stuff.pop(n)` removes the item with index `n` (remember that in Python the first index always is 0).

While lists are very flexible, performing computations with numbers in lists sometimes is less efficient than using ***arrays***. A one-dimensional array corresponds to a vector, while a two-dimensional array is a matrix. Higher-dimensional arrays also can be constructed. While arrays are more efficient for mathematical computation, they have the disadvantage, relative to lists, of being limited to containing only one type of object; i.e., all integers, or all floating point numbers. Arrays are available in the `numpy` package.

One can create an array from scratch: e.g., `A = zeros([3, 2], float)` creates a matrix with three rows and two columns, with each set to a value of floating-point zero. Additionally, a list (of numbers) can be converted into an array; for example, if `L = [3.5, 2.77, 1]`, then `v = array(L, float)` converts the list into an array of floating-point numbers. (If you convert a list containing floating point numbers into an integer array, the numbers get truncated: every digit after a decimal point is discarded, and numbers with absolute value less than 1 are set to 0.) An array can be created with non-zero values using a combination of the kinds of statements above, for instance `u = array([[1.5, 7], [-13.55, 0.1]], float)` creates a 2×2 matrix of floating point values. An element of an array can be picked out using its indices; e.g., in matrix `u`, `u[0,0]` would be `1.5`, `u[1,0]` would be `-13.55`, etc.

Some languages, including Python (and especially MATLAB) are designed, at least to some degree, to perform some operations with arrays more efficiently, and with more compact code, than one could do with a loop – this more efficient treatment of these objects is referred to as ***vectorization***. As a simple example of the efficiency with which Python can manipulate vector-like objects, imagine that we want to create a 100-element object `vec` with the value `1` in each position. The brute force approach would be to use a loop that sets each element individually:

```
for i in range(100):
    vec[i] = 1
```

A somewhat more efficient method uses a feature unique to Python called ***list comprehension***, which essentially is a `for` loop that indexes directly into (and creates) a list (not an array):

```
vec = [1 for i in range(100)]
```

There is an even faster method for creation of this array, implemented in the `numpy` package:

```
vec = ones(100)
```

This version essentially creates the entire array at once.

`numpy`'s built-in functions, such as `sin`, `cos`, `exp`, and others, are designed to work on entire arrays at once, in a vectorized manner. (Some of the functions also work on lists.) Additionally, entire rows or columns of arrays can be acted on at once; e.g., if `A` and `B` are square 2-D `numpy` arrays, then `A[m] = B[:,n]/2` would set the row of `A` with index `m` equal to one half of the column of `B` with index `n`. (For better notational "symmetry," one could write `A[m,:]` in place of `A[m]`.)

It's worth pointing out that the `numpy` library contains a type of object that it calls a `matrix`, which behaves somewhat differently from an array. In some cases, it might make sense to use these matrices rather than arrays, but one has to keep in mind their particular behaviors. See the online `numpy` documentation pages for further information. In these modules, the term "matrix" as applied to a Python object will refer to a `numpy` array.

2.4 Profiling

How can we measure which of these vector-building methods is fastest? One way is by ***profiling***, a technique available in many programming languages for determining the computer resources (CPU time, memory, etc.) used by a set of instructions in code. This profiling capability often is implemented most fully in an ***IDE*** (integrated development environment), such as the environment in which MATLAB normally is run. Even outside of an IDE, some profiling capability may be available. For example, Python has a `timeit` function that can be used to time a user-defined function. `timeit` will execute the function a certain number of times (1,000,000 by default; adjustable by the user), and report the CPU (central processor unit) time used, in seconds.

The code below defines functions to implement the three array and list initialization methods described above, and prints out the time for each. The first 3 lines simply import the necessary functions from `numpy` and declare a couple of variables, the intermediate lines implement the three loop definitions as functions, and the last three lines print the time for each of the three loops.

```
""" Program to time vector creation using different approaches. """

from numpy import zeros, ones
N = 100           # length of vectors/list
vec1 = zeros(N)   # has to be created before being indexed, below

def loop1():
    for i in range(N):
        vec1[i] = 1

def loop2():
```

```
    vec2 = [1 for i in range(N)]  # this is a list, not a vector

def loop3():
    vec3 = ones(N)

import timeit
print("time for loop1 is: ", timeit.timeit('"loop1()"', number=1000000))
print("time for loop2 is: ", timeit.timeit('"loop2()"', number=1000000))
print("time for loop3 is: ", timeit.timeit('"loop3()"', number=1000000))
```

Running this on the author's Windows laptop, the results are loop1: 0.0123 sec, loop2: 0.0097 sec, loop3: 0.0095 sec. So we can see that the `numpy` method is the fastest, although in this case not by a lot. But even the small 2% improvement relative to the next-fastest method would be meaningful in a computation taking many days (and the latter two approaches are significantly faster than the loop approach).

The timing approach above will work in IPython, but one also can use one of its special functions, known as ***magics***, to do timing. An IPython magic is called in one of two ways: (i) `%func` applies the magic "`func`" only to the remainder of the line in which it is typed, (ii) `%%func` applies the magic to all of the lines below it in the cell in which it's typed. The magic function used for timing is called `time`. To call it on, say, loop2 above, we would first have to define `N` in an earlier cell, and then we'd type

```
%%time
vec2 = [1 for i in range(N)]
```

Be aware that the times reported by either of these timing methods are not actually the CPU time needed to run your code! If you run the timing code a few times, the results will vary a bit. That's because the reported time is affected by whatever else the CPU is doing while running the code being tested, and the user has limited control over those other operations. Therefore, the most you can do with such a profiling tool is to run it several times and take the shortest time as an estimate of the actual CPU time needed to run the code. (The shortest time represents the case when the CPU was doing the least other work besides running your code.) If you have Python or IPython installed on your own computer, you might try running the timing test for comparison.

We will not be doing much of this kind of profiling in future modules, but it is useful to know about such tools in case you need to optimize code to run very fast, and to use simple practices to improve computational efficiency. Here are a few tips (not a complete list!) for speeding up code:

- Minimize the computation done in loops: compute/define *before the loop* any constants to be used inside the loop.
- Try a vectorized approach when possible; it might not always be faster, but it usually is.
- If code runs slowly, try profiling it to find out what line(s) of code are taking the most time. It may be possible to write them more efficiently.

While optimizing code for speed can be important in some situations, one should be strategic in doing it. As the eminent computer scientist, Donald Knuth (creator of TeX), has written:[3]

> Programmers waste enormous amounts of time thinking about, or worrying about, the speed of noncritical parts of their programs, and these attempts at efficiency actually have a strong negative impact when debugging and maintenance are considered. We should forget about small efficiencies, say about 97% of the time: **premature optimization is the root of all evil**. Yet we should not pass up our opportunities in that critical 3%.

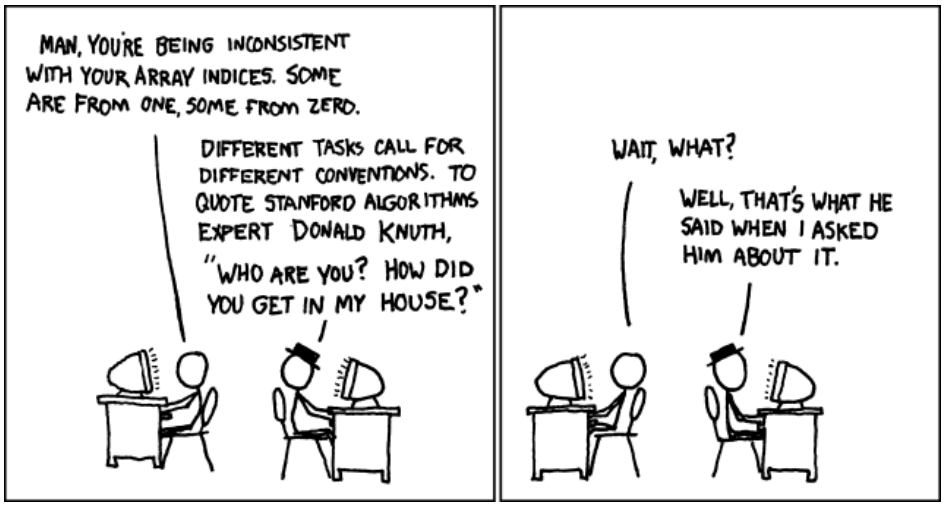

Image credit: xkcd.com

[3] "Structured Programming with `go to` Statements," Computing Surveys, Vol. 6, No. 4, December 1974

Exercises

Exercise #1

Consider the function $f(n) = n^{0.1}$. (a) Plot it and $\log_{10}(n)$ (available in `numpy` as `log10`) on the same graph over the range $n = 0 \rightarrow 50$. What does the graph suggest about which function is greater for large n? (b) There is a second crossing at a *much* larger value of n, beyond which $n^{0.1} > \log_{10} n$. At what very large value of n does $n^{0.1}$ equal $\log_{10}(n)$? Demonstrate your conclusion analytically (you may need a trial-and-error approach – think powers of 10), and replot the two functions on the same graph over a suitable range of n to confirm your answer. *Python reminders*: (1) The plot command in `matplotlib` has the syntax `plot(xvalues, yvalues, decorators)`, where "`decorators`" refers to features like marker and line color, marker and line type, legend label, etc.). (2) To run `matplotlib` inside an IPython notebook, the first code cell executed should contain the lines

```
%matplotlib inline
from matplotlib.pyplot import *
```

Exercise #2

Plot the functions $f_1(n) = \log_{10}(n)$, $f_2(n) = n$, and $f_3(n) = n \log_{10}(n)$ on the same graph (in different colors) over the range $n = 0 \rightarrow 100$ and include a legend. Briefly comment on their growth rates for larger n.

Exercise #3

Prove *analytically* that big-O of a simple binary search algorithm, in which the set of elements to be searched in each step of the algorithm is divided *in half*, is log(n). To do this, you want to figure out how many steps (m) it would take to find a *randomly* chosen element in a sequential list of n elements, like a phone book, if the set of elements that contains the desired one is divided in half with each step. (We don't want to choose a "special" element, such as the first or last one, since they may be easier to find than random elements, depending on the algorithm used). You should ignore subtleties such as whether the half-intervals generated in this process contain even or odd numbers of elements. Try this problem without reading past this sentence; if you get stuck, go ahead and read the following hint. [Hint: In step 0, before any subdivision, the desired element lies within the entire set of n elements. After step 1, we have reduced the set that could contain our desired element to a size of $n/2$. After step 2, the set of elements containing our desired one has a size of $n/4$. Extend this pattern to the appropriate limit to obtain an expression for m in terms of n.]

Exercise #4

Write two Python functions to compute the inner product (also known as the dot product) of two vectors ($u[1] \cdot v[1] + u[2] \cdot v[2] + \ldots$ for vectors u and v) of 100 elements each, in two ways: (i) using a `for` loop, and (ii) using `numpy`'s built-in, vectorized inner product function. (In both methods, the code should check that the two vectors have the same length and print an error message if not. It' always

good to anticipate user error!) Your two functions should take the two vectors as inputs. Time the two functions using `timeit`. How do they compare? Note: the inner product can be written in *summation notation* as $\mathbf{u} \cdot \mathbf{v} = \sum_i u_i \, v_i$, where the sum runs from 1-N, where N is the number of elements (but recall that Python's vectors and matrices start counting at 0, not 1). This way of representing the inner product may be useful in the mastery exercise.

Mastery Exercise

Matrix multiplication is a common mathematical procedure in physics, engineering, and other fields. The `numpy` package has functions for matrix manipulation, but here you are to do brute force multiplication.

(a) Start with two 3 x 3 arrays **A** $= [[1,2,3],[4,5,6],[7,8,9]]$ and **B** $= [[3,5,7],[9,11,13],[17,19,23]]$ (in list form), where each triplet of values represents a row in the array. Write a simple code to multiply these two arrays (i.e., compute **A** $\cdot$ **B** $=$ **C**), recalling that in matrix multiplication the procedure is as follows: compute the inner product of the first row of the first matrix (**A**) with the first column of the second matrix (**B**): this gives the value in the first row and first column of the result matrix (**C**). The inner product of the first row of **A** with the second column of **B** gives the element of the first row, second column of **C**, and so on. (In other words, the element in the ith row and kth column of **C**, C[i,k], equals the inner product of row i of matrix **A** with column k of matrix **B**.) To do this, you can use a set of three nested `for` loops. It may help if you first write out the elements of **C** using index notation (e.g., as in $\sum_i u_i \, v_i$ for the inner product of vectors $\mathbf{u}$ and $\mathbf{v}$).

(b) If you were able to complete part (a) without too much trouble, try generalizing your program to handle matrices with non-identical shapes (the number of columns of the first matrix must still equal the number of rows of the second matrix).

Breakpoint Answers

Breakpoint 1: A number on the order of 10^{-16}.

Breakpoint 2: A number close to but not equal to 1; e.g., 0.99999999999999988898.

Breakpoint 3: The error would be $\sim (6 \times 10^6 \text{ m}) \times 10^{-16} = 6 \times 10^{-10}$ m, or 0.6 nanometers – pretty darn small! The precision of modern computers is more than sufficient for most applications.

Breakpoint 4: $O(n^3)$ – it's determined by the term with the strongest n-dependence.

Breakpoint 5: $O(1)$, $O(\log(n))$, $O(n)$, $O(n \log(n))$, $O(n^2)$, $O(n^3)$.

Breakpoint 6: $O(p \times q \times p)$, or $O(p^2 q)$.

Scientist Profile

Arguably the first computer programmer, Augusta Ada Byron – the Countess of Lovelace, daughter of famous poet Lord Byron, and known as **Ada Lovelace** – was born 10 December 1815 in London. Her mother insisted that her tutors teach her mathematics and science, which was unusual for young women of the time. At the age of 17, she met Charles Babbage, a mathematician and inventor who is known as the father of computing for inventing the "difference engine" a mechanical computer. He also had plans for building a more sophisticated computer, the "analytical engine." Lovelace was asked to translate an article on the latter from French to English, and added her own comments about the machine. In those comments, she described how codes could be used by the machine to represent letters and symbols, as well as numbers. She also proposed ways in which the machine could repeat certain operations a precursor of modern-day loops in computer code. Ada Lovelace died of cancer on 27 November 1852, just weeks shy of her 37th birthday. On 10 December 1982 the anniversary of her birth the U.S. Department of Defense officially approved the reference manual for a recently developed computing language named "Ada" in her honor.