Week 3 Coding Lecture 3: Computational Complexity

It is generally not enough to be able to solve a problem numerically; we also need our solution to be efficient. To figure out how efficient our method is, we need an estimate of how much work we actually have to do to solve a system. The usual approach with numerical methods is to give a rough estimate of how the work changes with the size of our problem. "Problem size" is a somewhat amorphous concept and it can often be difficult to come up with a good definition. In our case, we are interested in systems of N equations with N variables, so we will use N as a measure of the size of our problem. So, for instance, the first example we looked at in the last lecture had $N = 2$ and the second example had $N = 3$. It is very common to have to solve systems with $N = 100$ or $N = 1,000$, and there are many fields where problems may be much larger than that.

Detailed estimates of speed are certainly important, but they quickly become very difficult. One of the reasons for this difficulty is that the fundamental operations that your computer executes when you type a line of code can vary widely from one version of python to another, and even from one computer to another. Since we can't possibly go into all of these details, our estimates will be quite rough. This means that we are free to ignore what might seem like substantial details, since these details will be affected by many other issues that we can't measure. (For instance, we will throw away all constant factors whenever convenient, so we would treat N steps the same as $3N$ steps or $10N$ steps.) Fortunately, our rough estimates will still prove very useful, and most mathematicians, computer scientists and programmers never have to worry about better estimates than these.

Before we can start, we need to know what fundamental steps (usually called *instructions* or *operations*) our computer can execute. As mentioned above, this varies from computer to computer, but we will assume that our processor can do any of the following in a single instruction:

1) Add two numbers together. That is, if x and y are numbers (not arrays), calculate $x + y$.

2) Subtract two numbers. That is, if x and y are numbers (not arrays), calculate $x - y$.

3) Multiply two numbers. That is, if x and y are numbers (not arrays), calculate xy.

4) Divide two numbers. That is, if x and y are numbers (not arrays), calculate x/y.

5) Multiply two numbers and add them to another. That is, if x, y and z are numbers (not arrays), calculate $xy + z$.

The first four are probably a single instruction on every computer you have ever used. The fifth one (called a *fused multiply add* instruction) is common on modern processors, but takes two instructions on older computers.

(Side note: Many processors can also perform several of these operations at once. For instance, most modern laptops can add two pairs of (standard python) floats together in the same amount of time they could add just one pair. Python may or may not take advantage of this fact, depending on how you write your code, and it is well beyond the scope of this class. In any case, it won't affect our final estimates.)

When the numbers involved (x, y and z) are floats (i.e., decimals instead of whole numbers), we call these instructions *floating point operations* or *flops*. Our goal is to estimate how many flops it takes to solve an $N \times N$ system, then get an estimate of how long a

Speed of back/forward substitution

To begin, let's assume we are trying to solve an $N \times N$ system of equations that is already in (upper/lower) triangular form. As we said in the last lecture, you can solve these in python with the `solve_triangular` command, which uses back or forward substitution. We will illustrate these calculations with an upper triangular system, but it is easy to translate this argument to one for a lower triangular system as well.

In general, our problem will look like this:

$$\begin{cases} a_{11}x_1 + a_{12}x_2 + \cdots + a_{1n}x_n &= b_1, \\ 0x_1 + a_{22}x_2 + \cdots + a_{2n}x_n &= b_2, \\ \vdots &= \vdots \\ 0x_1 + 0x_2 + \cdots + a_{nn}x_n &= b_n. \end{cases}$$

Solving for x_n takes only one flop: $x_n = b_n/a_{nn}$ can be calculated with a single division operation. Once we have x_n, solving for x_{n-1} takes two flops. We first do a fused multiply add to calculate $b_{n-1} - a_{n-1}x_n$, then a division to calculate $x_{n-1} = (b_{n-1} - a_{n-1}x_n)/a_{n-1,n-1}$. (It is easy to get confused by the first operation: It is written in a different order than our definition of a fused multiply add instruction, and there is a negative sign here. However, your processor calculates $(-a_{n-1})x_n + b_{n-1}$. If you wrote $x = -a_{n-1}$, $y = x_n$ and $z = b_{n-1}$, this would be the same as our original definition.) Continuing this pattern, each row takes one more flop until we get to x_1, which takes N flops: One each to multiply the x values we already know by their corresponding coefficients and subtract them to the right hand side, then one more to divide everything by a_{11}.

This means that it takes $1 + 2 + 3 + \cdots + (N-1) + N$ flops to solve the whole system. It is easy to check that this adds up to $N(N+1)/2 = N^2/2 + N/2$ total flops. We are only interested in a very rough estimate of the speed, so we will simplify this considerably. First, when N is very large (which is the only time when it really matters how fast our code is), $N^2/2$ is far bigger than $N/2$. This means that the $N/2$ term does not make a very big difference in our estimate, so we might as well just say that it takes $N^2/2$ flops. Second, we don't really care about constant factors. There are several reasons why these factors are customarily ignored, but the most relevant to us is that the factor of 2 will make less difference in the final speed estimate than all the other approximations we are making about exactly what a flop is or how long it takes to execute. We will therefore just say that back/forward substitution takes approximately N^2 flops.

We say that back/forward substitution is $\mathcal{O}(N^2)$ (pronounced "order N squared" or "Big-oh of N squared"). In Big O notation, we ignore any constant factors and any terms with smaller powers, so $\mathcal{O}(N^2)$ and $\mathcal{O}(N^2/2)$ and $\mathcal{O}(N^2/2 + N/2)$ are the same thing.

What good is this estimate? If we knew how long it took to execute a flop, we could then estimate how long it would take to solve a system with back/forward substitution. Let's say it takes t seconds to execute a flop. The total time for the whole algorithm is therefore approximately tN^2. The key thing to notice here is that the t in our total time has a power of 1, while the N in our total time is raised to a power of 2. This means that if you get a computer that is ten times as fast (so t is ten times smaller), then your algorithm will take ten times less time to run. However, if you make your problem ten times as big (so N is ten times larger), then your algorithm will take $10^2 = 100$ times longer.

We will assume that $t \approx 10^{-9} = 0.000000001$ seconds. There are a lot of complications that go into processor speed (in particular, not all operations take the same amount of time, and your processor spends a lot of its time doing other tasks besides solving linear systems, like printing to the screen or keeping the operating system running) but we can get decent order of magnitude estimates by using a small constant value like this.

This means that we could solve a 100×100 system in $10^{-9} \cdot 100^2 = 0.00001$ seconds, while a 1000×1000 system would take $10^{-9} \cdot 1000^2 = 0.001$ seconds and a $10,000 \times 10,000$ system would take $10^{-9} \cdot 10000^2 = 0.1$ seconds and a $100,000 \times 100,000$ system would take $10^{-9} \cdot 100000^2 = 10$ seconds.

Ten seconds is not that slow, so this means that even quite large triangular systems can be solved in a reasonable time. (You probably can't even make a $100,000 \times 100,000$ system on your personal computer, so practically speaking this is the upper limit on how long back substitution will take you.) You can verify these times yourself using the `time` function, which can be found in the `time` module. The `time` function takes no arguments and returns the current time in seconds since epoch. (Epoch is the official beginning of time in the Unix world. It is 12am UTC on January 1, 1970.) To find out how long your code takes to run, you can find the time before

you start the code and after the code finishes and then subtract the two. For example, you could use the following code to make an $N \times N$ upper triangular system and then time the `solve` function with it.

```
import numpy as np
import scipy.linalg
import time

N = 10000
# Make a random NxN matrix and zero out everything below the diagonal
A = np.random.rand(N, N)
for i in range(N):
    for j in range(i):
        A[i, j] = 0;
b = np.random.rand(N, 1)

# Time the solution
initial_time = time.time()
scipy.linalg.solve_triangular(A, b)
final_time = time.time()
print("The solve function took this many seconds: ")
print(final_time - initial_time)
```

```
The solve function took this many seconds:
0.1463172435760498
```

Note that `time` is at best as accurate as your system clock. Depending on your operating system and other details, it is probably on the order of ms accuracy, so not all of the decimal places you see are relevant. The exact times you get will depend heavily on your exact computer and python setup. I recommend you try this code with various values of N with your computer plugged in (because laptops will usually throttle your processor if running on battery). This is also an interesting place to compare python and MATLAB if you are interested in using both.

Speed of Gaussian elimination

Now let's do the same thing for a general (not triangular) $N \times N$ system. Our standard approach is to eliminate all coefficients below the diagonal so that our system becomes upper triangular, then use back substitution to solve the resulting system. We already know that it takes $\mathcal{O}(N^2)$ flops to use back substitution, so we just need to check how many flops are needed for the elimination step.

For each coefficient, we have to multiply a row by some number and then add it to another row. That sounds like a fused multiply/add operation, but instead of operating on single numbers, we are operating on an entire row. This means that we have to use $N + 1$ flops to zero out a single coefficient - one multiply/add for each variable, plus one more for the right hand side. How many coefficients do we need to zero out? We need to eliminate everything below the diagonal, and there are N^2 total coefficients, but N of those are on the diagonal, and so we have to zero out $(N^2 - N)/2$ coefficients. We also know that each coefficient takes $N + 1$ flops to zero out. Therefore, the whole elimination process takes $(N + 1) \cdot (N^2 - N)/2 = (N^3/2) - (N/2)$ flops. In addition, we just found out that the back substitution process takes $(N^2/2) + (N/2)$ additional flops, and so the entire process of Gaussian elimination takes $(N^3/2) - (N/2) + (N^2/2) + (N/2) = (N^3/2) + (N^2/2)$ flops.

Remember that in big O notation we ignore constant factors and terms with lower powers, so $\mathcal{O}(N^3/2 + N^2/2)$ is the same thing as $\mathcal{O}(N^3)$. This means that we can just call the entire process of Gaussian elimination $\mathcal{O}(N^3)$. (As above, this is a gross oversimplification, but when N is very large, the only important feature of this estimate is the biggest power of N, and so we get good qualitative estimates without worrying about the extra details.)

Just like with substitution, we can use this to estimate how long the backslash command will take. If we assume that an individual instruction takes $t \approx 10^{-9}$ seconds, then Gaussian elimination would take roughly tN^3 seconds. As with substitution, these exact numbers are pretty close to meaningless; the key thing to notice here is the exponents and how the speed scales up as N changes. The t is only raised to the first power, while the N is raised to the third power. This means that if you get a computer that runs ten times as fast (so t is ten times smaller), the backslash command will run ten times quicker. However, if you make your problem ten times larger, then the backslash command will take $10^3 = 1000$ times longer. For example, we could solve a 100×100 system in approximately $10^{-9} \cdot 100^3 = 0.001$ seconds, while a 1000×1000 system would take roughly $10^{-9} \cdot 1000^3$ = 1 seconds, and a $10,000 \times 10,000$ system would take approximately $10^{-9} \cdot 10000^3 = 1000$ seconds, or about 16 minutes. A $100,000 \times 100,000$ system would take almost twelve days to solve. As you can see, the cubic power means that our run times will get out of hand very quickly.

As above, you can use the `time` function to time python code yourself. We could time Gaussian elimination with code like this:

```
import numpy as np
import scipy.linalg
import time

N = 10000
# Make a random NxN system
A = np.random.rand(N, N)
b = np.random.rand(N, 1)

# Time the solution
initial_time = time.time()
scipy.linalg.solve(A, b)
final_time = time.time()
print("The solve function took this many seconds: ")
print(final_time - initial_time)
```

```
The solve function took this many seconds:
6.667825937271118
```

As before, it's important to keep in mind that the exact times can vary a lot from run to run and computer to computer. What we are really interested in here is comparing different values of N and comparing different methods.