8 Computational complexity

In the last section I showed that the code that implements Cramer's method runs very slow even for small n, namely, for $n = 9$. The Gaussian elimination, contrary to this, was able to work with a system of 300 equations with 300 unknowns in quite a manageable time. The goal of this section is to give a precise mathematical formalization for this phenomenon.

8.1 Counting the operations

Since the time required for a certain algorithm to run depends on the details of the hardware we use, it is more representative to count the number of some elementary operations, such as multiplications, divisions, additions, and subtractions. I note that strictly speaking we should count separately additions/subtractions and multiplications/divisions, since the latter take way more time to be performed, but here I will count everything together.

I start with Cramer's rule and recall that to solve a system of n equations with n unknowns I need to calculate $n + 1$ determinants. Let D_n be the number of elementary operations that is necessary to be performed to calculate the determinant of an $n \times n$ matrix. Clearly I have that $D_2 = 2 + 1 = 3$, i.e., I need to perform two multiplications and one subtraction. Now, to compute D_3, I recall that I need to compute 3 times the determinant of 2×2 matrices, multiply them by the elements of the first row, and make two additions/subtractions, in other words,

$$D_3 = 3D_2 + 3 + 2.$$

Generalizing to an arbitrary n, I get

$$D_n = nD_{n-1} + n + (n - 1) = nD_{n-1} + 2n - 1, \quad D_2 = 3,$$

which is a linear difference equation, which is principle can be solved exactly. Since at this point I do not care too much about the exact result (but I invite the reader to try to improve on my calculations), I consider instead a smaller number d_n, for which

$$d_n = nd_{n-1}, \quad d_2 = 3,$$

such that clearly $d_n < D_n$ for any $n > 2$ (that is, I am computing a very conservative lower estimate for D_n).

Since

$$d_n = nd_{n-1} = n(n-1)d_{n-2} = n(n-1)(n-2)\ldots 4 \cdot 3 \cdot d_2 > n(n-1)\ldots 4 \cdot 3 \cdot 2 = n!,$$

I can very conservatively estimate that

$$D_n \approx n!$$

and hence Cramer's method requires a ridiculous

$$T_{Cramer} \approx (n + 1)n! = (n + 1)!$$

Math 488/688: Numerical Analysis
e-mail: artem.novozhilov@ndsu.edu. Fall 2021

elementary operations to be performed (a more careful analysis would find that this number is actually around $e(n+1)!$).

Now let me count the same operations for the Gauss elimination with back substitution. Recall that this method consists of two steps. First I transform $[\boldsymbol{A} \mid \boldsymbol{b}]$ into $[\boldsymbol{U} \mid \boldsymbol{b}']$, where $\boldsymbol{U}$ is an upper triangular matrix. After this, I perform back substitution computing $x_n, x_{n-1}, \ldots, x_1$.

I start with the first step, which is done sequentially on $n-1$ rows (I do not need to do anything when I get to the last row).

Assume that at step i I have

$$\left[\begin{array}{cccccc|c} a_{11} & a_{12} & \ldots & \ldots & \ldots & a_{1n} & b_1 \\ 0 & a_{22} & \ldots & \ldots & \ldots & a_{2n} & b_2 \\ 0 & 0 & \ddots & & & & \vdots \\ \vdots & & & a_{ii} & \ldots & a_{in} & b_i \\ \vdots & & & a_{i+1,i} & \ldots & a_{i+1,n} & b_{i+1} \\ & & & \vdots & & \vdots & \vdots \\ 0 & & \ldots & a_{ni} & \ldots & a_{nn} & b_n \end{array}\right].$$

How many operations I need to perform to move to step $i+1$? Note that I need to modify the elements of the $(n-i)\times(n-i)$ submatrix of $\boldsymbol{A}$ composed of rows and columns with indexes $i+1,\ldots,n$; each of these entries must be multiplied by some factor, hence $(n-i)^2$ multiplications, to compute these factors I need $(n-i)$ divisions, I also need to perform $(n-i)^2$ additions for the same reason. For the $n-i$ elements of vector $\boldsymbol{b}$ I have $(n-i)$ multiplications and $(n-i)$ divisions. Summarizing, multiplication/division are performed for the forward step of Gaussian elimination

$$M_f = \sum_{i=1}^{n-1} \big((n-i)^2 + 2(n-i)\big)$$

times, and additions/subtractions are performed

$$A_f = \sum_{i=1}^{n-1} \big((n-i)^2 + (n-i)\big)$$

times. These sums can be calculated by recalling that

$$\sum_{k=1}^{n} k = \frac{n(n+1)}{2}\,, \quad \sum_{k=1}^{n} k^2 = \frac{n(n+1)(2n+1)}{6}\,.$$

I find that

$$M_f = \frac{2n^3+3n^2-5n}{6}\,, \quad A_f = \frac{n^3-n}{3}\,.$$

For the backward step first I need one division to find x_n, then one multiplication, one addition, and one division to find x_{n-1}, two multiplications, two additions, and one division to find x_{n-2}, and so on, in total

$$M_b = n + \sum_{i=1}^{n-1} i = \frac{n(n+1)}{2}$$

multiplications and divisions, and

$$A_b = \sum_{i=1}^{n-1} i = \frac{n(n-1)}{2}$$

additions/subtractions.

Finally, the total numbers are

$$M_t = M_f + M_b = \frac{n(n^2+3n-1)}{3}\,, \quad A_t = A_f + A_b = \frac{n(2n^2+3n-5)}{6}\,,$$

and the total number of operations is

$$T_{Gauss} = M_t + A_t = \frac{2n^3}{3} + \frac{3n^2}{2} - \frac{7n}{6}\,.$$

Clearly for sufficiently large n

$$T_{Gauss} \approx \frac{2n^3}{3}\,.$$

As an example, let's take $n = 100$ and assume that we have a computer performing at 100 gigaflops (FLOP is an abbreviation for floating point operations per second, hence I assume 10^{11} operations per second, which is way more than for any standard desktop computer). For Gauss method I get

$$T_{Gauss}(100) = 681550$$

operations, which gives me the time $6.8155 \cdot 10^{-6}$ seconds for my hypothetical computer.

Now for Cramer's method I find

$$T_{Cramer}(100) \approx 9.42595 \cdot 10^{159}$$

operations, which would take approximately $2.98895 \cdot 10^{141}$ years to compute on my imaginary computer (the best estimate we have nowadays for the age of our Universe is $13.77 \cdot 10^9$ years). So, jokes aside, one should never (absolutely never, even for $n = 2$) use Cramer's method to solve a system of linear algebraic equations.

8.2 Computational time complexity

What I estimated in the previous subsection is known as computational time complexity of an algorithm. In general, *computational complexity* of an algorithm is the amount of resources required to run it. Two main resources are memory and time. I did not care much about memory (I invite the student to think about how much memory each algorithm would require). The time complexity is usually counted in elementary operations (which could be multiplications, divisions, additions, subtractions which we counted, and also assignments and functional calls within the given code) versus the size of the input. In our case the input is specified by the number of unknowns n. In majority of cases the exact number of operations is difficult (or even impossible) to calculate and relatively small n are irrelevant, therefore frequently time complexity is computed *asymptotically*, in terms of "big O" notation.

Definition 8.1. *Let f, g be two real values positive functions of integer argument n. We say that*

$$f(n) = \mathcal{O}\big(g(n)\big)$$

(this reads "big O of $g(n)$"), if

$$f(n) \leq Mg(n)$$

for all $n \geq n_0$ and some constant M.

So when someone says that the given algorithm has a quadratic complexity, or big O of n^2, it means that for sufficiently large n the number of operations that need to be performed for the algorithm to end is bounded above by the expression Mn^2.

Using this definition I can say that Gaussian elimination has complexity $\mathcal{O}(n^3)$ (since other powers of n will not contribute for large n), or cubic complexity, whereas Cramer's method has complexity $\mathcal{O}((n+1)!)$ or factorial complexity.

A few words on terminology. An $\mathcal{O}(1)$ algorithms is said to have constant time complexity, $\mathcal{O}(\log n)$ — logarithmic complexity, $\mathcal{O}(n)$ — linear complexity, $\mathcal{O}(n^k)$ — polynomial complexity, $\mathcal{O}(2^n)$ — exponential complexity.

Before I finish this section, I would like to make two important remarks. First, if there are two algorithms, one, say, $\mathcal{O}(n^2)$ and the other one is $\mathcal{O}(n^3)$, it *does not* mean that the latter algorithm will perform slower in reality. Remember, that the big O notation is asymptotic, and that for fixed finite n the constant M could be much larger for the quadratic algorithm than for the cubic one ($1000 \cdot 10^2$ is of course smaller than $10 \cdot 10^3$). Hence, while the asymptotic estimate of time complexity is important, for actual algorithms very often the exact number of elementary operations should be calculated or estimated, as it was done for the Gauss and Cramer methods.

Second, the asymptotic complexity, among other things, shows how much one can increase the size of the input n if the computational power grows (which happens since the creation of computers). Assume that we have R resources and we work on a problem that has exponential complexity $\mathcal{O}(2^n)$ and the largest n we can afford to run is, say, fixed $n = N$. Assume also that our resources doubled: $R_n = 2R$. How much we can increase N? Since $R_n = 2R = 2 \cdot 2^N = 2^{N+1}$, this implies that for an algorithm with exponential complexity doubling the resources allows to increase the size of the input by 1! For a quadratic algorithm the same reasoning yields that $R_n = 2R = 2n^2 = (\sqrt{2}N)^2$, i.e., now we can increase our input by the factor $\sqrt{2} \approx 1.41$. Convince yourself that for algorithms with logarithmic complexity I can quare the size of the input, i.e., $N_n = N^2$. These simple informal arguments should convince the reader that developing algorithms with lower asymptotic time complexity is very important, even if they run slower for sufficiently small n.

8.3 Examples

8.3.1 Fibonacci numbers

8.3.2 Sorting algorithms

8.3.3 Graph algorithms