# 10

# Operation Count; Numerical Linear Algebra

## 10.1 Introduction

Many computations are limited simply by the sheer number of required additions, multiplications, or function evaluations. If floating-point operations are the dominant cost then the computation time is proportional to the number of mathematical operations. Therefore, we should practice counting. For example, $a_0 + a_1x + a_2x^2$ involves two additions and three multiplications, because the square also requires a multiplication, but the equivalent formula $a_0 + (a_1 + a_2x)x$ involves only two multiplications and two additions.

More generally, $a_Nx^N + ... + a_1x + a_0$ involves $N$ additions and $N + (N-1) + ... + 1 = N(N+1)/2$ multiplications, but $(a_Nx + ... + a_1)x + a_0$ only $N$ multiplications and $N$ additions. Both require $O(N)$ operations, but the first takes about $N^2/2$ flops for large $N$, the latter $2N$ for large $N$. (Although the second form of polynomial evaluation is superior to the first in terms of the number of floating-point operations, in terms of *roundoff* it may be the other way round.)

Multiplying two $N \times N$ matrices obviously requires $N$ multiplications and $N-1$ additions for each element. Since there are $N^2$ elements in the matrix this yields a total of $N^2(2N-1)$ floating-point operations, or about $2N^3$ for large $N$, that is, $O(N^3)$.

A precise definition of the "order of" symbol $O$ is in place (Big-O notation). A function is of order $x^p$ if there is a constant $c$ such that the absolute value of the function is no larger than $cx^p$ for sufficiently large $x$. For example, $2N^2 + 4N + \log(N) + 7 - 1/N$ is $O(N^2)$. With this definition, a function that is $O(N^6)$ is also $O(N^7)$, but it is usually implied that the power is the lowest possible. The analogous definition is also applicable for small numbers, as in chapter 7. More generally, a function is of order $g(x)$ if $|f(x)| \leq c|g(x)|$ for $x > x_0$.

An actual comparison of the relative speed of floating-point operations is given in table 8-I. According to that table, we do not need to distinguish between addition, subtraction, and multiplication, but divisions take somewhat longer.

## 10.2 Slow and fast method for the determinant of a matrix

It is easy to exceed the computational ability of even the most powerful computer. Hence methods are needed that solve a problem quickly. As a demonstration we calculate the determinant of a matrix. Doing these calculations *by hand* gives us a feel for the problem. Although we are ultimately interested in the $N \times N$ case, the following $3 \times 3$ matrix serves as an example:

$$A = \begin{pmatrix} 1 & 1 & 0 \\ 2 & 1 & -1 \\ -1 & 2 & 5 \end{pmatrix}$$

One way to evaluate a determinant is Cramer's rule, according to which the determinant can be calculated in terms of the determinants of submatrices. Cramer's rule using the first row yields $\det(A) = 1 \times (5+2) - 1 \times (10-1) + 0 \times (4+1) = 7 - 9 = -2$. For a matrix of size $N$ this requires calculating $N$ subdeterminants, each of which in turn requires $N-1$ subdeterminants, and so on. Hence the number of necessary operations is $O(N!)$.

A faster way of evaluating the determinant of a large matrix is to bring the matrix to upper triangular or lower triangular form by linear transformations. Appropriate linear transformations preserve the value of the determinant. The determinant is then the product of diagonal elements, as is clear from the previous definition. For our example the transforms (row2$-$2$\times$row1) $\rightarrow$ row2 and (row3+row1) $\rightarrow$ row3 yield zeros in the first column below the first matrix element. Then the transform (row3+3×row2) $\rightarrow$ row3 yields zeros below the second element on the diagonal:

$$\begin{pmatrix} 1 & 1 & 0 \\ 2 & 1 & -1 \\ -1 & 2 & 5 \end{pmatrix} \rightarrow \begin{pmatrix} 1 & 1 & 0 \\ 0 & -1 & -1 \\ 0 & 3 & 5 \end{pmatrix} \rightarrow \begin{pmatrix} 1 & 1 & 0 \\ 0 & -1 & -1 \\ 0 & 0 & 2 \end{pmatrix}$$

Now, the matrix is in triangular form and $\det(A) = 1 \times (-1) \times 2 = -2$. An $N \times N$ matrix requires $N$ such steps; each linear transformation involves adding a multiple of one row to another row, that is, $N$ or fewer additions and $N$ or fewer multiplications. Hence this is an $O(N^3)$ procedure. Therefore calculation of the determinant by bringing the matrix to upper triangular form is far more efficient than either of the previous two methods. For say $N = 10$, the change from $N!$ to $N^3$ means a speedup of very roughly a thousand. This enormous speedup is achieved through a better choice of numerical method.

## 10.3 Non-intuitive operation counts in linear algebra

We all know how to solve a linear system of equations by hand, by extracting one variable at a time and repeatedly substituting it in all remaining equations, a method called Gauss elimination. This is essentially the same as we have done above in eliminating columns. The following symbolizes the procedure again on a $4 \times 4$ matrix:

$$\begin{pmatrix} * & * & * & * \\ * & * & * & * \\ * & * & * & * \\ * & * & * & * \end{pmatrix} \rightarrow \begin{pmatrix} * & * & * & * \\ & * & * & * \\ & * & * & * \\ & * & * & * \end{pmatrix} \rightarrow \begin{pmatrix} * & * & * & * \\ & * & * & * \\ & & * & * \\ & & * & * \end{pmatrix} \rightarrow \begin{pmatrix} * & * & * & * \\ & * & * & * \\ & & * & * \\ & & & * \end{pmatrix}$$

Stars indicate nonzero elements and blank elements are zero. Eliminating the first column takes about $2N^2$ floating-point operations, the second column $2(N-1)^2$, the third column $2(N-2)^2$, and so on. This yields a total of about $2N^3/3$ floating-point operations. (One way to see that is to approximate the sum by an integral, and the integral of $N^2$ is $N^3/3$.)

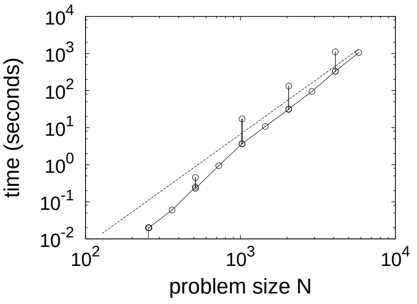

Figure 10-1: *Execution time of a program solving a system of N linear equations in N variables. For comparison, the dashed line shows an increase proportional to $N^3$.*

Once triangular form is reached, the value of one variable is known and can be substituted in all other equations, and so on. These substitutions require only $O(N^2)$ operations. A count of $2N^3/3$ is less than the approximately $2N^3$ operations for matrix multiplication. Solving a linear system is faster than multiplying two matrices!

During Gauss elimination the right-hand side of a system of linear equations is transformed along with the matrix. Many right-hand sides can be transformed simultaneously, but they need to be known in advance.

Inversion of a square matrix can be achieved by solving a system with $N$ different right-hand sides. Since the right-hand side(s) can be carried along in the transformation process, this is still $O(N^3)$. Given $Ax = b$, the solution $x = A^{-1}b$ can be obtained by multiplying the inverse of $A$ with $b$, but it is *not* necessary to invert a matrix to solve a linear system. Solving a linear system is faster than inverting and inverting a matrix is faster than multiplying two matrices.

We have only considered efficiency. It is also desirable to avoid dividing by too small a number or to optimize roundoff behavior or to introduce parallel efficiency. Since the solution of linear systems is an important and ubiquitous application, all these issues have received detailed attention and elaborate routines are available.

Figure 10-1 shows the actual time of execution for a program that solves a linear system of $N$ equations in $N$ variables. First of all, note the tremendous computational power of computers: Solving a linear system in 1000 variables, requiring about 600 million floating point operations, takes less than one second. (Using my workstation bought around 2010 and Matlab, I can solve a linear system of about 3500 equations in one second. Matlab automatically took advantage of all 4 CPU cores.) The increase in computation time with the number of variables is not as ideal as expected from the operation count, because time is required not only for arithmetic operations but also for data movement. For this particular implementation the execution time is larger when $N$ is a power of two. Other wiggles in the graph arise because the execution time is not exactly the same every time the program is run.

---

Table 10-I shows the operation count for several important problems. Operation count also goes under the name of "computational cost" or "computational complexity" (of an algorithm).

| problem | operation count | memory count |
|---|---|---|
| Solving system of $N$ linear equations in $N$ variables | $O(N^3)$ | $O(N^2)$ |
| Inversion of $N \times N$ matrix | $O(N^3)$ | $O(N^2)$ |
| Inversion of tridiagonal $N \times N$ matrix | $O(N)$ | $O(N)$ |
| Sorting $N$ real numbers | $O(N \log N)$ | $O(N)$ |
| Fourier transform of $N$ equidistant points | $O(N \log N)$ | $O(N)$ |

Table 10-I: *Order counts and required memory for several important problems solved by standard algorithms.*

A matrix where most elements are zero is called "sparse." An example is the tridiagonal matrix: A tridiagonal matrix has the following shape:

$$\begin{pmatrix} * & * & & & & \\ * & * & * & & & \\ & * & * & * & & \\ & & \ddots & \ddots & \ddots & \\ & & & * & * & * \\ & & & & * & * \end{pmatrix}$$

A sparse matrix, where most elements are zero, can be solved with appropriate algorithms, much faster and with less storage than a full matrix. For example, a tridiagonal system of equations can be solved with $O(N)$ effort, not $O(N^2)$ but $O(N)$, as compared to $O(N^3)$ for the a matrix.

The ratio of floating-point operations to bytes of main memory accessed is called the "arithmetic intensity." Large problems with low arithmetic intensity are limited by memory bandwidth, while large problems with high arithmetic intensity are floating point limited. For example, a matrix transpose has an arithmetic intensity of zero, matrix addition of double precision numbers $1/(3 \times 8)$ (very low), and naive matrix multiplication $2N/24$ (high).

## 10.4 Data locality

Calculations may not be limited by FLOPs/second but by data movement; hence we may also want to keep track of and minimize access to main memory. For multiplication of two $N \times N$ matrices, simple row times column multiplication involves $2N$ memory pulls for each of the $N^2$ output elements. If the variables are 8-byte double precision numbers, then the arithmetic intensity is $(2N-1)/(8(2N+1)) \approx 1/8$ FLOPs/byte. Hence the bottleneck may be memory bandwidth.

*Tiled matrix multiplication:* Data traffic from main memory can be reduced by *tiling* the matrix. The idea is to divide the matrix into $m \times m$ tiles small enough that they fit into a fast (usually on-chip) memory. For each pair of input tiles, there are $2m^2$ movements out of and $m^2$ into the (slow) main memory. In addition, the partial sums, which will have to be stored temporarily in main memory, have to be brought in again for all but the first block; that is about another $m^2$ of traffic per tile. For each output tile, we have to go through $N/m$ pairs of input tiles. In total, this amounts to $(N/m)^3 4m^2 = 4N^3/m$ memory movements, or $4N/m$ for each matrix element, versus $2N$ for the naive method. This reduces data access to the main memory and turns matrix multiplication into a truly floating point limited operation.

There are enough highly-optimized matrix multiplication routines around that we never have to write one on our own, but the same concept—grouping of data to reduce traffic to main memory—can be applied to various problems with low arithmetic intensity.

**Recommended Reading:** Golub & Van Loan, *Matrix Computations* is a standard work on numerical linear algebra.