

# Computational Basics of Linear Algebra

Sunday, November 22, 2020

11:10 PM

## Flop counts of multiplication

① Dot product of two length  $n$  vectors

$$\vec{x}^T \vec{y} = [x_1, \dots, x_n] \cdot \begin{bmatrix} y_1 \\ \vdots \\ y_n \end{bmatrix} = \sum_{i=1}^n x_i y_i$$

takes  $O(n)$  flops  
floating point operations

② mat-vec

$$m \begin{bmatrix} A \end{bmatrix} \cdot \begin{bmatrix} \vec{x} \end{bmatrix} = m \begin{bmatrix} A \vec{x} \end{bmatrix} \quad \text{is just } m \text{ dot products, each } O(n),$$

so  $O(mn)$  flops

③ mat-mat

$$m \begin{bmatrix} A \end{bmatrix} \begin{bmatrix} B \end{bmatrix} = m \begin{bmatrix} C \end{bmatrix} \quad \text{is just } k \text{ mat-vecs}$$

$$\text{Since if } B = \begin{bmatrix} | & & | \\ b_1 & \dots & b_k \\ | & & | \end{bmatrix} \text{ then } AB = \begin{bmatrix} | & & | \\ Ab_1 & \dots & Ab_k \\ | & & | \end{bmatrix}$$

so  $O(kmn)$  flops

ie, if  $A, B \in \mathbb{R}^{n \times n}$ , then  $O(n^3)$  flops

possibly on exam



supplemented,  
not on exam

FACT

An absolutely amazing fact is that you can compute matrix-matrix products in  $O(n^{2.8})$  via Strassen's Algo. (1969) and even faster w/ Coppersmith-Winograd and others.

Major open problem in theoretical computer science (TCS):

If  $\epsilon > 0$ , does there exist an algorithm for matrix multiplication with  $O(n^{2+\epsilon})$  flops.

**Strassen** is only helpful for large matrices. It's not that helpful for most medium-sized matrices, so not commonly implemented, though people keep publishing papers showing it can be competitive in practice.

On the computer

Rule #1 in numerical analysis: ~~do not talk about numerical analysis~~

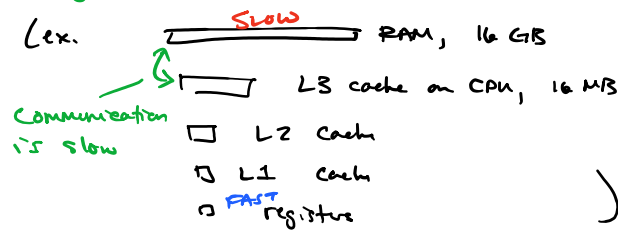
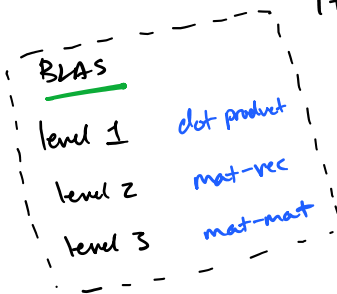
Do not ever implement matrix multiplication on your own

Why? It's the #1 most used subroutine, and heavily optimized

Use a good BLAS library such as Intel's MKL

## Basic Linear Algebra Subsystems

It's all about **minimizing data movement** through different memory hierarchies (ex. **SLOW** RAM, 16 GB)



and avoiding **cache misses**

and exploiting CPU **pipeline instructions** (max, Arx)  
aka **vector instructions**

A modern approach to developing good code is to have it work in blocks  
well, 1990

of a matrix, and use as many "Level 3 BLAS" calls as possible  
and **cache-oblivious** if possible

Ex. is **LAPACK**, a library (dependent on a BLAS library) to compute linear algebra (eigenvalues, etc) ... everything we'll cover in this class

Matlab and Julia use LAPACK (and numpy/scipy might also)

## Rule #2 of numerical analysis

to solve  $A\vec{x} = \vec{b}$ , never compute  $A^{-1}$  and

use  $\vec{x} = A^{-1} \vec{b}$ .

Faster / more accurate to do the LU (or LDL<sup>T</sup> or Cholesky) decompositions that we'll be talking about in this chapter.

Or, for specialty matrices (e.g., sparse), there are even more techniques

( All of applied math is about solving linear equations  
(and approximating nonlinear equations by linear equations) )

### Importance of linear algebra

- Linear algebra routines are how we benchmark super-computers

- PDEs / ODEs model physics

Systems of PDEs / ODEs require matrix multiplication  
and inversion (for implicit solvers)

- Linear algebra code is optimized and fast and reliable

⇒ we try to use it when possible

⇒ it gets used a lot

⇒ we spend time to make lin. algebra code even faster

If you're interested in implementation:

- MIT's "[Introduction to Numerical Methods](https://github.com/mitmath/18335#lecture-11-feb-26)" (18.335) on github, taught by Steven Johnson, <https://github.com/mitmath/18335#lecture-11-feb-26> (lecture 11 on caching), <https://github.com/mitmath/18335/blob/master/notes/matmults.pdf>, <https://github.com/mitmath/18335/blob/master/notes/Memory-and-Matrices.ipynb>
- Eijkhout (2017), [Introduction to High-Performance Scientific Computing](https://pages.tacc.utexas.edu/~eijkhout/istc/istc.html), <https://pages.tacc.utexas.edu/~eijkhout/istc/istc.html>. Part of libFLAME team (now "The Science of High-Performance Computing Group") at UT Austin